Static Analysis of information flow for Python Case study: Verification of the back-end of an e-voting system

João de Araujo Correia Neto Lopes
Instituto Superior Técnico, Lisboa, Portugal

November 2020

Abstract

With the evolution of technology, software development has highly increased and as so, the need for software developers has never been higher. As that need increases, so does the probability that security of the produced software takes a toll in terms of priority. And in a world where more dependance on software also translates into more methods attackers can use, that is a very dangerous statement. One of the most used programming languages is Python. Python is widely used to implement the server-side applications as several established frameworks have been created for it as well as a high number of third party libraries for the programmer to use. Python is considered a dynamic language, which means that some operations that could be performed before execution are now done so at run-time.

In order to perform an automatic analysis on the code that provides strong security guarantees, we introduce a tool based on information flow that is capable of detecting illegal leaks in terms of both confidentiality and integrity, while only requiring the programmer to type annotate its code as well as providing the security levels that he deems necessary.

The main test case of this work will be the analysis of an Electronic voting system, also known as e-voting. Since e-voting relies on technology, it consequently results in it inheriting all its flaws and problems security-wise. E-voting is the perfect test case since it highly emphasizes on the properties that we analyze, both confidentiality and integrity.

Keywords: Python analysis, static analysis, Information Flow

1. Introduction

1.1. Motivation

With the evolution of technology, software development has highly increased and as so, the need for software developers has never been higher. In the current time, the whole world is depending on software, from small businesses to multinational companies. Among other factors, the fact that the needs and requirements of software are ever-evolving, translates into ensuring that there won't be a shortage of demand for software developers in the foreseeable future. As that need increases, so does the probability that security of the produced software takes a toll in terms of priority. And in a world where more dependance on software also translates into more methods attackers can use, that is a very dangerous statement. It is natural that as software development flourishes, so does the market that wants to exploit it, so the need for security should be getting higher in our priority list, not lower. Even if the developer takes security into consideration, he will most likely use third party libraries for which he may not even have access to the source code.

Python is widely used to implement the server side of applications as several established frameworks have been created for it as well as a high number of third party libraries for the programmer to use. Considering how often Python is used it is definitely important to develop tools that can analyze it. Python is considered a dynamic language, which means that some operations that could be performed before execution are now done so at run-time. One example would be the possibility to change a variable's type as the program in being executed. As a consequence, analyzing such a language in regard to its security can be challenging.

To address the former challenges we propose the usage of information flow analysis. In information flow[DD77] analysis, the variables in a program are usually divided into two security levels, high and low. If we are guaranteeing confidentiality these can correspond to, respectively, secret information, and public information. If on the other hand we are ensuring integrity, these can be represented as tainted and untainted information. If the changes in a variable from one level spread to the other level we may have an illegal flow depending on whether we are ensuring confidentiality

or integrity. If we are evaluating a program's confidentiality changes in high-level variables cannot be visible in low-level variables, on the other hand if we are evaluating the integrity of a program we can't allow flows from low level variables to high level variables.

This work will focus its attention on the TAGUS e-voting system, which was developed in IST(Instituto Superior Tecnico) as a voting solution that is suitable for Universities and other similar organisations. It is based on the H-Belenios protocol[Mar17], that is an adapted version of Belenios[Glo13] that also accounts for the traditional method of voting that involves voting through paper ballots. In order for the e-voting system to be useful it is essential that the security properties and assurances provided by the protocol are verified. Here, we briefly mention some of the most important ones. It is essential that the votes remain anonymous, which means that the voter can't be associated with the corresponding vote. Auditability is also fundamental, the possibility for every voter to check if his vote was processed correctly. Another important property would be Confidentiality, which means that the vote content cannot be inferred until the final count. And last but not least Integrity, only authenticated voters can vote, and they can only do so once. As both confidentiality and integrity are vital for this e-voting system it supplies the perfect test case for our tool.

The aim of this work is to make a static Python analysis tool which is based on information flow. The solution involves the elaboration of a tool that allows for illegal information flows to be detected while only requiring the programmer to type annotate its code as well as providing the security levels that he deems necessary.

1.2. Information Flow Tool

The basis of this tool is information flow. In order to achieve it we would need to allow information to flow in one direction but not in the other. This was done resorting to inheritance since it allows the information to flow from the sub class to the super class but not the other way around. That fact can be used to create a lattice that represents the possible security levels with inheritance connecting the different levels. The security properties are composed of integrity which assures that the information is accurate and trustworthy, and confidentiality that limits the access to information. The two properties work inversely to each other which means the highest level in the lattice in terms of confidentiality will be equivalent to the lowest level in integrity.

The next step would rely on type annotations and a type checker that could identify when there would be type errors. Since the types of the variables are now security levels, each type error would be equivalent to either a confidentiality or integrity leak. Since we now have security level type annotations in the code we need a way to type check them. It involves changing the original code so that the type checker itself can detect leaks and change its output to a more useful output considering what we are using it for. Naturally changing the code relies on going through the abstract syntax tree and changing it so that the type checker can perform the desired information flow. If the security level lattice is solid and the annotations are correctly attributed MYPY can already perform as desired for assignments.

Besides the explicit flows this tool also addresses built-in functions like print and implicit flows originated from cycles and conditional statements. How these scenarios were analyzed will be discussed in depth in both the model and implementation sections but the general idea is to again modify the original code so that the type checker can analyze it directly.

1.3. Objectives

This work addresses the fact that software is exposed to a number of threats from various sources. Its importance in society given the fact that software development is only increasing, ensures the need to make sure that software is secure.

This goal involves more specifically:

- *C*reation of a static analysis tool that enforces information flow for Python.
- *E*nsuring that the server-side of the TAGUS e-voting system is secure.

1.4. Contributions

Created a static analysis tool based on information flow that uses type annotations and a type checker to address most types of illegal flows in regard to two properties, integrity and confidentiality. This tool was originally created specifically for the code related to the server side of an e-voting system but was later extended to be able to apply information flow to any Python program.

2. Background

2.1. Static vs dynamic analysis

The discussion between static and dynamic analysis and when to use one or the other is always vital when considering the security of a program. While dynamic analysis is normally used in an environment where a little bit of overhead is tolerated and the program can be stopped if there is a vulnerability detected, static analysis thrives on the fact that it can detect the errors prior to the execution even though it may introduce false positive results depending on the tool used.

In the context of the protocol being analyzed, both static and dynamic analysis can be applied.

This being said it's essential to check if the implementation of the protocol allows for dynamic analysis to be used on the isolated "threads/processes" that are at risk and can be stopped at runtime without endangering the functioning of the whole server.

When considering the nature of the protocol, its priority is focused not in efficiency but in making sure there are no confidentiality leaks and the integrity of the data remains accurate. Therefore it is of no consequence if a limited overhead is introduced in the program due to dynamic analysis techniques.

2.2. Information Flow

In information flow [DD77] analysis, the variables in a program are usually divided into two security levels, high and low. If we are guaranteeing confidentiality these can correspond to, respectively, secret information, and public information. If on the other hand we are ensuring integrity, these can be represented as tainted and untainted information. If the changes in a variable from one level spread to the other level we may have an illegal flow depending on whether we are ensuring confidentiality or integrity. If we are evaluating a program's confidentiality, changes in high level variables cannot be visible in low level variables, on the other hand if we are evaluating the integrity of a program, we can't allow flows from low level variables to high level variables.

Information flow is usually studied under one of two levels of precision. It can deal exclusively with explicit flows, which as the name implies, happens when a secret is explicitly exposed to a variable that can be seen by the public. And it can also address implicit flows, which happens when the execution arrives at a conditional statement and is divided into two branches, performing publicly observed side effects depending on which branch is taken [RSL10].

Non-Interference is a property that ensures that every low input results in the same low output, no matter the difference in high level variables. Which means there cannot be anything inferred about the high level input through the observable output (low level variables).

Although theoretically this property would have to always be present, in real world applications it can be too restrictive to achieve any kind of useful result. It is therefore important that the mechanism we choose is able to allows us to, in a controlled manner, release as much information as possible without making it possible for the attacker to infer anything about the system. The application of this notion allows for more flexibility despite having to ensure the properties stated before.

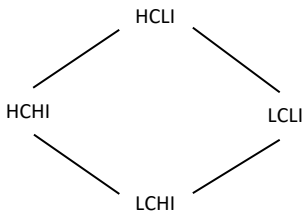

Figure 1: Security Lattice.

In light of the former two paragraphs, the ideal is not proving that a program complies with the Non-Interference property, because in practice that would be too restrictive, but instead proving it complies with a more flexible and more useful version of that property.

3. Model

3.1. Information Flow

Prior to jumping straight into this tool's architecture let us recall one very important decision, why we opted into static analysis. The most important factor was the context in which this analysis would be performed. Since the target of our system is an e-voting system, its availability is of a critical nature.

The basis of this tool is information flow. This was done resorting to inheritance. Inheritance is a very useful mechanism where classes inherit properties and behaviors from their superclass. Thus class would be called a subclass. Its main use lies on the reusability provided by it since there is no need to redefine the already present functionality that it inherited from its superclasses.

3.2. Security Lattice

Since we have now established a way for us to identify illegal flows between two classes we can leverage that along with the concept of multiple inheritance to be able to extend that into multiple security levels and the corresponding relationships between them. We will be able to create a lattice like the one observed in figure 1, that represents the possible security levels with inheritance connecting the different levels.

The security levels previously mentioned will be related to two different security properties. These are composed of integrity which assures that the information is accurate and trustworthy, and confidentiality that limits the access to information. The two properties work inversely to each other which means the highest level in the lattice in terms of confidentiality will be equivalent to the lowest level in integrity.

3.3. Explicit flow

We now have a mechanism that allows us to compare two variables whose types belong to an inheritance hierarchy and check that there are no illegal flows between them. At first, the concept of type annotations in a dynamically typed language like Python may raise some questions. Even though Python only checks the type of the value of a variable during run-time, it may still be beneficial to have a hint of what that type might be to facilitate the comprehension of the code.

In the context of our problem, the type annotations serve as a bridge between the security level lattice and the variables relevant for the programmer. Since the types of the variables are now security levels, each type error would be equivalent to either a confidentiality or integrity leak.

Type annotations can be added to variables in assignments as illustrated in figure 1, to arguments of functions and can even represent the return values of said functions.

```
//Type annotation in assignment
Var = "test" # type:HighConf
//Type annotation in a function
def funct(arg:HighConf):
//Type annotation of the return of the function.
def funct(arg)-> HighConf
```

Listing 1: Possible type annotations

As we could see in listing 1 we have laid the foundation for our tool to work but still lack an essential step, the type checker. The type checker, more specifically MYPY, will check the code for type errors and since the types are now security levels, it will output a warning when an illegal flow is discovered. The reasons to why this type checker was selected among the few that exist are described in the background chapter.

3.4. Implicit flow

We already mentioned the existence of implicit flows but it is also important to understand how they occur in the context of source code. These instances where implicit flows occur include the presence of a conditional statement or a loop.

Resorting to figure 2 let us consider the conditional statement where different computations can be performed based on the conditional expression. To better understand it, imagine the following scenario: a person is trying to figure out if a sandbox has been used by a cat. Yet that person only has access to the information lying on the residing sand. Depending on the state of the sand, for example if it has footprints imprinted on the sand, one could deduce if a cat had been present without ever actually seeing it.

Going back to our example, if both the possible computations contained observable variables,

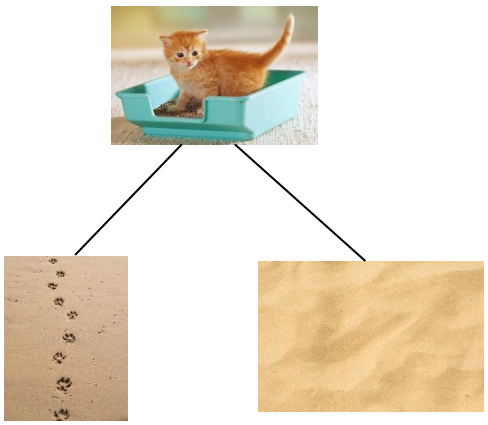

Figure 2: Catbox dillema.

like the sand, and they differed in value, like footsteps in the sand. It would be possible to obtain information about the conditional expression itself without having direct access to it. If the conditional expression contained confidential data, we would be in the presence of a security leak. Depending on which branch was computed, it would be reflected in a change of the observable variable and we would gain knowledge about the conditional expression.

3.5. Built-in functions

Another example of a security leak would be in the presence of built-in functions. In fact most of built-in functions can be considered a liability in certain conditions, for example the use of the built-in function print with its content being a confidential variable would disclose the information.

3.6. Coherence regarding changes

Navigation through the code will be done resorting to the Abstract Syntax Tree which will be, throughout this paper, mentioned as AST. The AST is a tree representation of the abstract syntactic structure of the source code. Ffigure 3 is a graphical representation of the AST. In this example you have the representation of a function call to function with the argument 10 and 11.

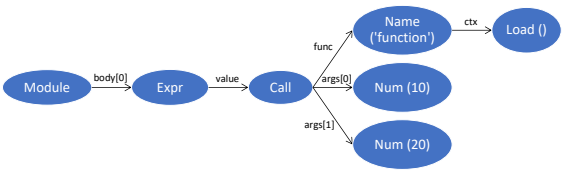

Figure 3: AST representaion

Being able to navigate through the code will make us able to identify the extraordinary cases and add the assignments in the correct places.

Having successfully examined the code and identified the security leaks, the output given by MYPY would still need some adaptation. Given that its purpose is to detect type errors, at this state, the output of MYPY running through our changed program would be the multiple type errors between different security levels as can be observed in listing 2.

```
1: Incompatible types in assignment
(expression has type HCLI,
variable has type LCHI)
```

Listing 2: Native MYPY output

Since the purpose of the output is now informing a programmer of the possible illegal information flows the output could be improved upon. For each warning it would contain the content of the source code, the variable that originated the warning and obviously the line in which the warning occurred.

4. Implementation

4.1. Workflow

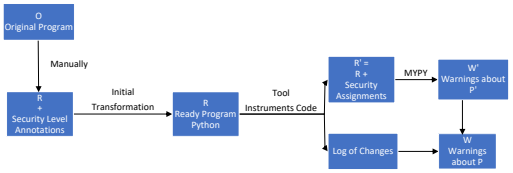

Figure 4: The stages of the tool.

The Workflow of the developed tool consists mainly of four stages as presented in figure 4.

4.1.1 First stage - Initial Transformation

The initial transformation involves changing the python code that is going be subject of the security analysis. However, prior to the execution of this step the programmer will already have performed the following measures. He will already have added the type annotations to the variables.

The annotations are equivalent to security levels of the variables and will be used by the tool to compare them. The programmer is responsible for adding the type annotations so he must have prior knowledge of the program. Apart from adding the type annotations the programmer is also required to provide the tool with his security lattice.

4.1.2 Second stage - Type Annotation

The second stage comprises of changing the variables' type into the type defined by the annotation. As the search in the AST progresses and the assignments are being found, if in the presence of a valid type annotation, the tool will perform its next operation. It will convert the right part of the assignment into an object of the type annotation.

```
// before the conversion
1: testVariable ="anyvalue"
# type:HighConfidentiality
```

Listing 3: Variable original state

```
// after the conversion
1: testVariable = HighConfidentiality
```

Listing 4: Variable transformed

To generate a program ready to be analyzed by our tool that contains the relevant security level annotations the variables would have to effectively belong to the type defined by the programmer, which again translates into its security level. Without this step MYPY wouldn't be able to discern illegal flows since the type annotations alone would not be sufficient for it to generate type errors.

Using a conservative approach, if just one of the variables is of a higher security level, in our example `var1`, the result from the operation should also be of that same higher security level regardless of the other variable's security level.

As to the actual approach, like represented in listing 6, it involves the replacement of the operation by a call to a specific function, `operationHandler`. Through that function we can compare the security types of `var1` and `var3`.

```
// before the changes
1: var1 = HCLI
2: var2 = LCHI
3: var3 = LCHI
4: var2 = var1 + var3
```

Listing 5: Operation original state

```
// after the changes
1: var1 = HCLI
2: var2 = LCHI
3: var3 = LCHI
4: var2 =
operationHandler(var1,
var3)
```

Listing 6: Operation transformed

This approach is also used when providing downgrading. Taking listing 8 as reference, in line 3, a warning will be output. Yet in line 5, the same assignment will no longer result in an output since we applied both declassification as endorsement to it.

```
// before the changes
1: var1 = HCLI
2: var2 = LCHI
3: var2 = var1
4: var1 = ""
#type:de-
#-classification(LCHI)
5: var2 = var1
```

Listing 7: Declassification example

```
// after the changes
1: var1 = HCLI
2: var2 = LCHI
3: var2 = var1
4: var1 = LCHI
5: var2 = var1
```

Listing 8: Declassification transformed

At the end of this stage the code will no longer be executable and will lose some information regarding the values of variables. As you can see in listing 4 we will no longer know that `testVariable` held the value `anyvalue`.

4.1.3 Third stage - Assignment Creation

The third step is possibly the most complicated one. As we have seen in chapter 3, there are multiple scenarios that wouldn't be addressed if our analysis were to stop at this point. The first involves the creation and addition of assignment expressions for which the result will be represented by `R'`. The second is a complimentary method whose purpose is connecting the original code to the increasingly different code that we are changing at each step of the architecture.

Generating R' After the second stage in finished, the following step involves the analysis of not only built-in functions of the language but loops and conditional statements as well.

With the build-in functions dealt with, we still have to address loops and conditional statements. Since the thought process is similar for those two scenarios we will just take the loop construct example and explain the conditional statement more in depth in the next section.

```
// before the changes
1: VariableList= HC
2: for var in VariableList:
3:       a = HC
```

Listing 9: Original For construct

```
// after the changes
1: Variable = HC
2: for var in VariableList:
3:     inVar = LC
4: inVar = VariableList
```

Listing 10: For construct transformed

As we can see in listing 10 an assignment was created and inserted in line 4 of this snippet. If `VariableList` is highly confidential and `inVar` is of a low confidential level, MYPY will be able to generate a warning and we will be able to address implicit flows in loops. The current example is a `for` loop but naturally the rationale can be extended to `while` loops as well.

Log of Changes While the R' state is being generated, the correspondence between the lines before and after the changes to the code is also being recorded. This is being done so that the location of the warning fed to the programmer, which will be part of the end result of this tool, is accurate.

In light of this discovery we are also in need of a log that allows us to convert the line of a certain warning in the current state of the code into the original code. We will call it log of changes from now on. At this stage we have achieved the two requirements to apply the final step, a complete log of changes, and the R' state program.

4.1.4 Fourth stage - Warning Creation

The final step involves running MYPY through the R' state program and resorting to the log of changes, adapting the warnings provided by MYPY. As soon as this process is finished, the end result will come in the form of warnings. These warnings, as you can see in listing 11 provides the line in which the leak has occurred as long as the specific line of code.

```
There is an error in line: 26
The corresponding code is:
public = secret
```

Listing 11: Warning example

4.2. Implementation Details

4.2.1 First Stage - Initial Transformation

Like mentioned in the last section, the initial transformation's purpose involves the cleanse, in the tool's perspective, of the original code. Once the programmer provides us with the security lists we will have to generate a class for each of the levels.

In the figure 1 we can see the resulting lattice from the two security lists.

```
1: class HCLI():
2:                 pass
3: class HCHI(HCLI):
4:                 pass
5: class LCLI(HCLI):
6:                 pass
7: class LCHI(LCLI,HCHI):
8:                 pass
```

Listing 12: Class Inheritance Hierarchy

In the listing 12 we can see the classes that were created in order to address the security require-

ments given by the lattice. In line 1 we can see the highest level confidentiality class which has no superclass and in line 7 we can see the lowest confidentiality class. Note that even though the highest and lowest classes are not directly connected they can still be compared. The classes do not require methods or arguments since their sole purpose is being the foundation for our inheritance-based structure.

The transformation performed in listing 14 prepares the code for the analysis. What results from this transformation will be what we call the original code.

```
// before the changes
1:
2: def exampleFunction():
# This comment will be deleted.
3:
4:      var = "testString"
# type: HCLI()
5:      print(var)
# Prints the variable.
```

Listing 13: Original code

```
// after the changes
1:
2:def exampleFunction():
3: var = "testString"
# type: HCLI()
4: print("Testing")
```

Listing 14: New Original code

4.2.2 Second Stage - Type Annotation

The second stage is done by identifying the initialization of the variable, which is normally an assignment and replacing the right part with the corresponding type regarding its security level. In this listing 16 we can see the two variables being modified as stated before.

```
// original code
1: variable1 = "example"
# type: LCHI()
2: variable2 = 20
# type: LCHI()
```

Listing 15: Variable original state

```
// after the changes
1: variable1 = LCHI
2: variable2 = LCHI
```

Listing 16: Variable transformed

Implementation-wise we are allowed to do this due to the existence of a parameter called `type_comment` that exists in every assignment node and will survive the cleansing performed in Stage 1. As we are traversing the AST besides identifying the assignment of the variable we also have to check in the parameter has a valid type annotation, which in our context will mean that it belongs to the inheritance structure that we mentioned before.

Like exemplified in the listing 17, we compare operands by applying them the Union function. As the name suggests this function groups all distinct arguments in a set.When uniting different types, if they are in different inheritance levels, the output will be of the highest level in the hierarchy.

```
1: def operationHandler(x:Tuple[A,S],
y:Tuple[A,T]) -> Tuple[A,Union[S,T]]:
2:              return
```

Listing 17: Operation Function

In listing 17 we can also see that the security types will be the second element of the tuples residing in function as arguments. If they are compatible, which in the context of security levels would translate of them consisting of the same security type, will generate a tuple with the class A and the type.

4.2.3 Third Stage - Assignment Creation

In this stage we will go deeper into the approaches to deal with both built-in functions and implicit flows.

Built-in functions While traversing the AST when the tool finds a node corresponding to a built-in function, it adds two assignments in the following lines. The first assignment is the creation of a dummy variable.

Conditional constructs We can now handle explicit flows such as assignments and built-in functions. All there is left to cover is how to deal with illegal implicit flows.

With respect to implicit flows, we will only be addressing three possibilities of information leaking. In reality, there are more possibilities, some are external to the code like computational time and are, therefore, outside of the score of this work but others make use of built-in constructs to generate these flows.

However, we are only addressing the three possibilities that are linked to conditional statements and loops. In these three cases information leaks can be reduced to one scenario.If there is a high level variable present in the control statement of a loop, and a low level variable present in the body, we are in the presence of a leak.

However, in terms of implementation the subject is not that simple. Since we operate in the static realm, we don't know which direction the program will take since we don't have access to the value of `secretVar` in this case. Given that our approach is always conservative we will consider both possible computations and issue a warning even if only one of them can actually originate one.

Once we have gathered all the relevant variables we find out where the conditional construct ends and insert the assignments after.

Loop constructs In this case we must make sure that if the control statement has a high security level variable and the body contains a low one MYPY triggers a warning.

Finally we find out where the loop ends and construct and insert the assignments after.

4.2.4 Fourth Stage - Warning Creation

In this final step, MYPY is ran through the changed code which will generate both the useful warnings in the wrong lines for the programmer and warnings that are not useful for this tool.

As stated before, each time an assignment is added to the code through the AST, the useful information is saved in a dictionary structure. Each entry in the dictionary is composed by the line in the changed code, which is the key of the dictionary, the corresponding line in the original code, and the code itself that belongs to the original line.

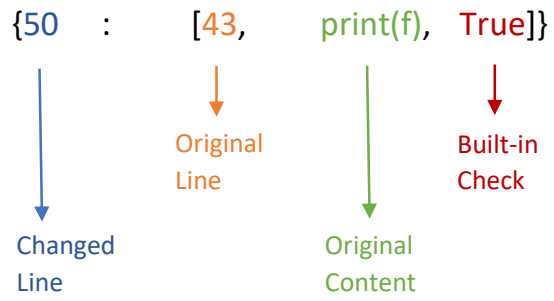

Figure 5: Dictionary cell.

Consequently, direct access to the dictionary will no longer be sufficient to identify all the relevant warnings. After checking the dictionary for direct correspondence, if not found, the tool will check how many lines were added before the line in question. It will do so by counting the entries in dictionary that is ordered by the warnings' original line, before reaching the warning in question.

In listing 18 we have what the final output of the tool looks like for the example mentioned earlier. In includes the line of the warning and its content as well as the variable that generated it.

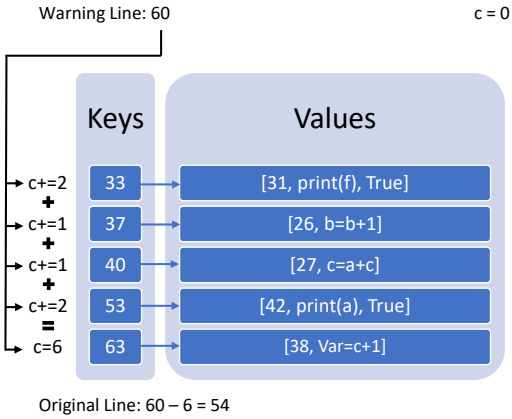

Figure 6: Dictionary.

```
There is an error in line: 26
The corresponding code is: public = secret
```

Listing 18: Final warning output

5. Results & discussion

All of the illegal flows discovered were regarding the security property integrity. As so that would mean that there was no secret information propagated into a public variable, which translates into no confidentiality breaches. This also translates into instances where unreliable information residing in variables was considered reliable, which again resulted in the said leaks.

In order for us to understand why this happened, let us closely analyze the code in listing 19. In a code with this size information will definitely be flowing between variables, like we can observe in lines 131 - 135, but every flow seems to have a common aspect. It always flows from the user input to the server's variables. As we can observe in line 131 as a name of the election is taken from the input coming through the user and put in the field of an object in the server.

```
131: election.name = data['name']
132: election.description = data['description']
133: election.hybrid = data['paperVotes']
134: election.startDate = data['startDate']
135: election.endDate = data['endDate']
```

Listing 19: Function containing information flow

This phenomenon happens due to the nature of the program. Since the e-voting server is responsible for using unreliable and public information, which takes from the user, and create and store

the result in a secret and reliable manner. This will result in no leaks when considering confidentiality but on the other hand will origin integrity related leaks since the unreliable information was passed on to a reliable variable.

However, this test case brought to light another limitation of this tool. It only deals with Python built-in functions, but in a world where frameworks proportionate more and more functionalities through their own built-in functions that represents a gap in our algorithm. Let us consider the following hypothetical scenario, a framework provides an alternative and exclusive method to print a variable. If the user tries to print a high level confidentiality variable by using Python's function "print", we will be in the presence of a leak and we will be able to detect it. But what if the user decides to use the alternative function? If it has the same functionality, or at least contains the original functionality from its Python counterpart, we will be in the presence of a flow that we cannot detect.

6. Conclusions

This thesis presents a static analysis tool that is able to detect illegal information flows in Python. It allows for a high degree of flexibility since the programmer can define multiple security levels regarding confidentiality and integrity, while still remaining easy to use since it resorts to basic type annotations to the code. Even though in a conservative manner, it addresses third party libraries. If having access to the source code of such libraries it allows for the programmer to type annotate them as well which will reduce the possible false positives that we would normally have from taking a conservative approach. This tool addressed both explicit and the more common implicit flows while providing the programmer useful feedback in order to keep his code secure. The flexibility that the programmer has in terms of attributing security annotations at his will comes with a price. He will also require the knowledge in terms of the software to make the most out of this tool.

In the following sections we will display in which specific scenarios can our tool detect a leak as well as what could be improved in future work.

6.1. Achievements

This static information flow tool can address illegal flows in the following scenarios:

- *Explicit flows*
  - *Assignment derived*
- *Built-in functions*
- *Implicit flows*
  - *Loop constructs*
    - *While constructs*
    - *For constructs*
  - *Conditional constructs*:
- *Third party libraries*
  - *With access to source code*: If we have access to the source code we can annotate it as well since thanks to MYPY it can detect leaks across files.
  - *Without access to source code* If we don't have access to the source code we can annotate what comes from the third party libraries. Even though it is less accurate than having analyzed the source code itself.

The current solution can now detect and warn the user of possible information flows in Python software, its performance depending on how accurate are the type annotations made to the target code

6.2. Future Work

Even though the tool covers a number of possible information leaks it could still be improved in multiple aspects:

- *Flexibility regarding functions:* At the current state, the tool does not provide a way for the return value of a function to adapt to the computations performed in it, the programmer can only define a fixed value for the return of the function.
- *Lattice flexibility:* Implement the ability to generate a security lattice given any number of confidentiality and integrity levels, currently the tool requires confidentiality and integrity to have the same number of security levels.
- *Built-in behavior:* The behavior given built-in functions could be more specific given that currently it does not adapt to the behaviour of the function.
- *Increase code exposure:* Even though the tool has been exposed to a high volume of python code, it still wasn't exposed to the gigantic dimension and near infinite scenarios that programs can have.
- *Implicit flow coverage:* Even though a decent number of implicit flows are covered, exceptions are not. The tool would highly benefic from being able to analyze try/catch constructs.

- *Framework limitation:* As it stands the tool can't cope with functions provided by frameworks, being limited to analyzing the ones provided by Python.

In conclusion, this thesis provides not only an implementation of a tool that can address illegal information flows in Python but its algorithm, which may also be applicable to other programming languages.

7. Acknowledgments

I would like to thank my family and friends for being there to provide all the help when I needed the most. I would also like to thank my supervisors Ana Almeida Matos and Jan Cederquist who were always avaiable to provide help and guidance when necessary as well as Instituto Superior Técnico for providing the infrastructure.

References

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[Glo13] Stéphane Glondu. Belenios specification. 2013.

[Mar17] Fernando Marques. E-voting on fenix. 2017.

[RSL10] Alejandro Russo, Andrei Sabelfeld, and Keqin Li. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*, 2010.