Static Code Analysis on Networking Code: Identifying the possibilities of finding implementation flaws using Abstract Syntax Trees

July 26, 2019

Ivar Slotboom
MSc. Security and Network Engineering
University of Amsterdam
ivar.slotboom@os3.nl

Wouter van Dongen
Supervisor
DongIT
wouter.vandongen@dongit.nl

***Abstract*—Static code analysis is a useful technique for both web and software developers to identify bugs and performance issues. Investigating network implementation issues on static code is a relatively new area of research, as analyzers are commonly trained on finding bugs and performance issues instead. Using the technique of Abstract Syntax Trees (AST), the static code could be parsed into a series of nodes to identify network implementation issues through either manual or scripted analysis.**

This research paper explores the technique of using ASTs, whilst it also contains an assessment of the capabilities and limitations. A tool was written to analyze custom crafted sample projects in multiple scenarios, which should simulate real world scenarios. The development of the tool was an iterative process, using purposefully implemented flaws which can then be discovered by analyzing the parsed output of AST. The output shows that ASTs can be effective in analyzing static code. On the other hand, it has also exposed some severe limitations. While it is possible to define network implementation rules in the tool to analyze the static code, every additional rule will require additional human interference, depending on the complexity of the rule. This means that the complexity of code analysis can be varying based on the techniques used in the static code itself. Whenever techniques such as threading are present, the reliability is not always there to perform a confident analysis, as the results may change during run time per machine and per configuration.

Other limitations include the lack of the context in the parsed AST node trees. In Python, a generally higher-level scripting language, a lot of context is lost due to the nature of the language itself. This can create difficult to comprehend analysis reports, since context is usually required to see where implementation flaws occur.

***Keywords*— Static Code Analysis, Networking Implementation, Abstract Syntax Tree, Python, AST**

I. Introduction

Static code analysis is a technique that can used by web and software developers to analyze their work in order to establish a feedback loop. The analyzer is built upon a set of rules and definitions that probes the static code for potential improvement points.[5] As these improvements points are primarily aimed at detecting bugs and unoptimized code, research was done to identify its capabilities on the implementation of networking code in applications.

In this paper, the possibilities of analyzing the implementation of networking code is investigated. This assessment is focused on how implementation issues can be detected, explore its scalability with large codebases and the limitations that come with it.

II. Background

A. Static Code Analysis

Static Code Analysis is mainly used in large codebases comprising millions of lines of code. The analyzer is then usually trained through machine learning to improve its detection consistency and to evolve its pattern recognition.[7] At first it analyzes small parts of the codebase to train itself. When the results are more consistent and correct, larger parts of the codebase will be exposed to the analyzer.[5] This process repeats indefinitely. The analyzer has to be able to analyze the entire codebase, needs to keep up with the latest changes of the programming language itself and find new improvement points that it was not able to identify before.

There are certain pitfalls with static code analysis. In case there is code that can not be analyzed, like e.g., no read permissions, this will not be reported as an error.[1] Other pitfalls include incorrect configuration settings, macro definitions and options during the analysis in lower level languages such as C/C++. Lastly, detected bugs could be misinterpreted by developers, since they can be perceived as false positives. This misjudgment may results in abandoned analyses or potentially affect the training model of the analyzer.

B. *Abstract Syntax Trees*

An Abstract Syntax Tree (AST) is commonly used to recreate the steps the code compiler or interpreter takes to understand the written code. ASTs basically represent the structure of the code through a node tree. The output of an AST is a set of nodes where most of the nodes consist of a parent node and often multiple child nodes. By traversing the nodes, it is possible to figure out the route on how a compiler or interpreter reaches certain functionality or behaviour.[7]

III. Related work

Goseva-Popstojanova et al. have researched the capabilities of static code analysis by detecting flaws in their code implementation in C/C++ and Java using multiple tools.[5] In their research they claim that the state-of-the-art tools are not very effective in detecting security vulnerabilities. The research also mentions that there is still an opportunity where it can be more effective than performing manual inspection. These findings are very valuable for this paper, since it shows that - while improvement points can be identified - they may not always be identified under the same circumstances. It shows that the feedback from the analyzer should be regarded as indicative feedback rather than the rule.

Comparable research by Al Bessey et al.[1] shows that the ideal scenario is to have an analyzer check millions of lines of code:

1) requiring little manual setup;
2) capable of finding the maximum number of serious true errors; and
3) generating a minimum number of false reports.

It shows that the process of static code analysis is iterative, as one needs to improve the analyzer repeatedly in order to get the desired results. It is also important to note that the analyzer needs to have all of the context available to make a proper report. This includes all of the available source code, configuration flags, macro definitions and compilation targets.

In other research done by Tasnim and Rahman, Abstract Syntax Trees were used to analyze static code by creating a syntactic structure.[7] As these ASTs do not describe every single detail of the actual syntax, it is enough to identify patterns and flaws in the code. This is useful as implementation flaws in networking code are usually detected on a higher-level, rather than the necessity to inspect the packets themselves.

IV. Research questions

This paper examines the capabilities of analyzing static code to find implementation flaws. This includes exploring its limitations, scalability and accessibility.

This analysis will therefore focus on the following research question: **"Is it possible to create a tool to analyze static Python code to detect potential network implementation flaws using Abstract Syntax Trees?"** Additionally, two sub-questions have been defined:

1) How can network implementation flaws be detected using Abstract Syntax Trees?
2) What are the limitations of identifying network implementation flaws using Abstract Syntax Trees?

V. Methodology

The methodology consists of two parts: the approach and test setup.

A. *Approach*

The best way to analyze network implementation issues, is by writing a simple networked application in Python 3.6.8 which provides full control over what happens in the application. This way it is possible to purposefully plant implementation flaws to be detected by the analyzer. This way an iterative production line is possible, where improvement points can be suggested one step at the time, since more and more flaws will be implemented for the analyzer to be found. The only disadvantage of this is that every implementation flaw needs to be predefined so that it can be detected, since there is no machine learning to train the machine to find identical patterns or behaviour.

Once the example projects have been set up, the analyzer will first parse the file into an AST using Python's AST library[3] so that it is possible to manually analyze the node tree. In this node tree it is possible to detect the weaknesses that were purposefully implemented. This then enables to add the rule of the detected flaw in the analyzer (referred to as impscan.py in Figure 1). All of the source code is publicly available on GitHub[6].

After implementing our manual detection as a pattern in the impscan.py file, the analyzer can be run, which will return a log with its findings. These findings are then either used to improve the current detection method or to implement a new detection rule instead.

1) Defining network implementation flaws: Implementation flaws are commonly occur on a high level, such as binding an IPv4 address to an IPv6 socket. Because of this, there is little to no risk that AST will lose the required level of detail as mentioned by Tasnim and Rahman.[7] Through manually analyzing the output of the parsed AST, the nodes will already reveal the flaws that should be detected by the analyzer.

B. *Test setup*

In order to develop the analyzer and test projects, a test setup was built that contains four components:

1) a desktop with Python 3.6.8[4] installed;
2) any text editor or integrated development environment (IDE);
3) test projects written with Python 3.6.8[4] to perform analysis on; and
4) an output terminal such as the default Linux terminal to receive the output of the analyzer

As Visual Studio Code (1.35.1)[2] gives feedback on the syntax of code, as well as suggestions while typing, it was used as primary text editor to make both the test projects and the analyzer itself. The setup and flow are shown in Figure 1:

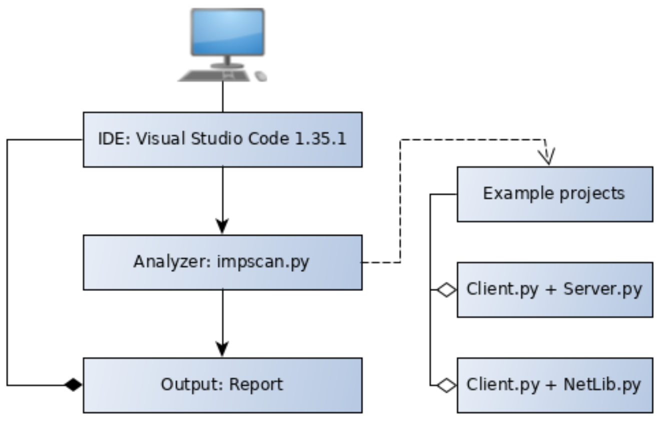

Fig. 1. Test Setup

The analyzer and test projects were made in the IDE. By running the analyzer on a test project, the output will be parsed into a report (as well as debugging information). These reports and debugging information will provide feedback on how to improve the analyzer.

VI. Results

This section contains a description of the results of our research.

A. AST parsing

AST parsing has shown to be an effective method to traverse through the nodes of the Python interpreter. The scenarios that were used to simulate network implementation flaws were detected. After every run, a report is shown on the screen which looks like the output shown below:

```
### Report ###
Errors: 1
        Socket "s" could be sending infinite amount
    of bytes because of its latest buffer assignment
    : sys.stdin.readline
Warnings: 1
        Socket connectivity is not configured for
    IPv6 connections
```

The errors are definitive flaws that must be handled or otherwise there could be serious connectivity or security issues. The warnings are suggestions on how to improve the network connectivity but can be ignored, since the application would likely work as expected without addressing it.

1) Multi-file parsing: As Python deals with imports, it is important to support the importing functionality of other files. The main script of the Client.py + NetLib.py (as shown in Figure 1) did not contain any socket implementation, creating aliases for the functionality it did need. The main file in this scenario looks like this:

```
import sys
import netlib

def OnReceive(aMessage):
  sys.stdout.write("\r{}".format(aMessage)) #
    Already includes a \n
  sys.stdout.write("Say: ")
  sys.stdout.flush()

netlib.Initialize(True, OnReceive)

try:
  while True:
    sys.stdout.write("Say: ")
    sys.stdout.flush()
    str_send = sys.stdin.readline() # Mistake 1:
    Potential for unlimited bandwidth
    netlib.Send(str_send)
except:
  netlib.Exit()
```

The code above has no socket definitions or functionality, since it is abstracted into the netlib.py file that has been imported by the main file. As mentioned in the research by Al Bessey et al., no code results into no bugs.[1] It is important for the analyzer to get all of the code to establish the context in order to perform a full analysis. If the analyzer did not import the netlib.py file in some way, it would yield neither warnings nor errors.

Tests have shown that AST has no issues with reading multiple files as one big file, even though the structure of the file is spread out over multiple places, like e.g.:

1) imports can appear in the middle;
2) functionality can be called before new function definitions; and
3) irregular Python coding patterns that the average developer would expect.

Instead, AST could still successfully perform the analysis with the exact same result as when the client was not abstracted to a netlib.py and client.py file.

B. Limitations

While an Abstract Syntax Tree unveils the process of the application, it is still difficult to handle all possible problems of detecting network implementation flaws for a number of reasons:

1) Threading causes unpredictable behaviour: Threading can cause unpredictable behaviour on run time, since it prevents static code analysis. For example, a thread could be altering a variable that is being used in the network implementation. This can be the buffer that is sent, the socket that is maintaining the connection or the application that is managing the networking itself on a high level.

```
# Semi-Python pseudo code - assume imports and
    sockets were handled
stringToSend = ""

def ThreadFunc1():
    global stringToSend
    while (True):
        time.sleep(3)
        stringToSend = "NewTextAssignment"

def ThreadFunc2():
    global stringToSend
    while (True):
        stringToSend += "1234567890!"
```

```

t1 = Thread(target=ThreadFunc1)
t2 = Thread(target=ThreadFunc2)

t1.start()
t2.start()

time.sleep(5) # Let the threads run for a while
socket.send(stringToSend) # How big is this string?
```

It is impossible to know how big the string ”stringToSend” will be from the example, since it differs on every machine with different architectures, available cores and clock speeds. With this in mind, it is extremely hard to track every socket destructor when threading comes into play. Situations like these can only thoroughly be checked on run time.

This is a serious issue as threading is a big part of networking code, as sockets are blocking on default. It is best practise to spread connections over multiple threads. Without the ability to thoroughly and precisely analyze threaded functionality, the analysis using ASTs could be lacking context, which could affect the overall output.

2) Multi-file projects versus imports: When importing a file using the import functionality in Python, it can cause unpredictable behaviour, since it is either importing directly from a previously installed library, or a file that is in the same directory. Because of this, it is unclear in the AST which functionality exactly is being called, causing a loss in the context during the analytical process.

```
# Import using existing library
import sys # imports the installed sys library
# AST:
#   Import(names=[<_ast.alias object at 0
    x0000020C212CFAC8>])
#   alias(name='sys', asname=None)

# Import using file in the same directory
import netlib # imports the netlib.py in the same
    directory
# AST:
#   Import(names=[<_ast.alias object at 0
    x0000020C212CFB70>])
#   alias(name='netlib', asname=None)
```

Fully parsed AST output and source code can be found on the GitHub page.[6]

In the example above you have to use heuristics and check for the ”netlib.py” file in the same directory in case it exists, but then also hope it is not an installed library that Python will use instead. If - in this particular case - there is a ”sys.py” file in the same directory, there is no way to know for sure if it is referencing to the sys.py file, or the sys library in Python, since the AST output regarding its identification is identical. This shows some limitations that the AST technique by itself is not enough to fully analyze a multi-file project.

An alternative would be to compare the installed libraries with the imports, although that would mean that the machine analyzing the code would need to have the exact same libraries installed with no additional libraries, since this could cause confusion. This could lead to performing workarounds when you want a remote machine to analyze the written code.

3) Complications with defining implementation rules in the analyzer: For every implementation rule, one usually needs the entire context (in AST referred to as the node tree), as well as an implementing of every scenario possible that could go wrong.

For example: When using a variable as a buffer in socket.send(), a rule that the variable cannot be larger than a predefined number of bytes must be implemented. This means that the variable must be checked by finding its latest Assignment() node in the node tree, which could be complicated if not impossible because of issues like e.g. threading.

To find the latest Assignment() node, one needs to traverse the entire node tree up until the socket.send() function. This could lead the re-analyzing the entire node tree multiple times, which causes massive amounts of time lost in analyzing the context. If this is the case, the scenario will most likely scale badly with larger codebases containing millions of lines of code. While time is not considered an important necessity of static code analysis - but rather a convenience to progress faster - this does not raise much of concern.

4) Dead code is harder to detect: In common scenarios, dead code could already be identified by parsing the output of an AST. Functions and variables that have no functionality will have no nodes in the AST representing their use. As networking code can be unpredictable due to the nature of the client-server setup, there is no guarantee if any of the networking-related functionality will be called unless expectations have been set. For instance, a chat client could connect to a server, but there is no guarantee that the chat client will send messages in the chat. As long as a chat message is never sent, the sending and receiving of chat data could be considered dead code. These unpredictable behaviours make it difficult to define what exactly is dead code.

In the case of such dead code being present, the dead code could confuse the analyzer when performing its analysis. For instance, if IPv6 sockets are implemented in a function that is considered dead code, the analyzer could still interpret the code as functional. This could mean that the analyzer things that IPv6 sockets have been implemented, while only IPv4 functionality is present. This could lead to incorrect conclusions when the analyzer shows the report on the screen.

An example of setting expectations would be to write additional code that simulates a test run. The code would run scenarios that is expected by the developer, like e.g. chat messages being sent. Having this code allows the AST to identify the code as being in use, which prevents any confusion for the analyzer regarding it as dead code instead.

The difficulty with dead code is that you need to analyze two projects at once in order to find dead code. If a client has built in functionality to receive a specific network message, but it is never sent on the server, then both the server and client contain dead code. This is a unique scenario, since with static code analysis the norm is to analyze one project per report.

VII. Conclusion

The research question addressed in this paper is: *"Is it possible to create a tool to analyze static Python code to detect potential network implementation flaws using Abstract Syntax Trees?"*

The results show that it is not only possible to detect network implementation issues, but also to create a detailed report based on the node tree generated by the AST. Limitations have been discovered whilst developing the analyzer, using ASTs and in the implications that come with the code that is being analyzed.

The use of ASTs has shown to be effective and flexible for two reasons:

1) Network implementations flaws mostly occur on a high level, making it easy for an AST parser to extract the required bits to perform an analysis.
2) Tests have shown that it can parse multiple files as one file while delivering the same analytical results.

Sub question 1: *"How can network implementation flaws be detected using Abstract Syntax Trees?"*

Since the parsed output of an AST provides a lot of useful information, like e.g. the steps required to get into a specific state of the application, it is not difficult to define a rule set the analytical tool is programmed to detect. The rule set usually requires high level information on the implementation techniques, which very much underpins the quality of the parsed output.

To define the rule sets, manual inspection was required to define what the analytical tool is supposed to detect. By analyzing the parsed output, patterns could be discovered and iteration could be applied to analyze possibilities that the analytical tool has to take into account. An example of this was that the sent buffer could contain infinite amounts of bytes, which could only be found through analyzing all possibilities in the node tree. This eased the tracking of what happened on a higher-level which allowed rule sets to be implemented without too many difficulties.

Sub question 2: *"What are the limitations of identifying network implementation flaws using Abstract Syntax Trees?"*

While the use of ASTs proved beneficial, it also has important limitations. The most important limitation is that network implementation flaws can be difficult to detect when threading is involved. Since threading represents a large part of networking, it is important that such networking flaws will be detected, even if when some of the context is missing.

The complexity of defining dead code is also a limitation, since one would need to set expectations what kind of data is transferred. As using the AST technique itself is not enough to define dead code, it may hinder the analyzer to provide an accurate report.

More complexities with the usage of ASTs entail that in the case of imports, context issues may occur. Parsing the output of the import does not show any differences on whether it is an installed library or not. As this can blur the overall analysis, it is recommended to use different techniques alongside AST to verify whether or not it is an installed library. Recommendations can be heuristics (by checking if the file exists alongside the main file of the project), or by comparing the import with the list of installed libraries (while ensuring the machine has exactly the same libraries installed; no more and no less).

VIII. Future work

Although the concept of detecting network implementation flaws has been researched, there is still a lot of future work ahead. For instance, related research has shown that machine learning is a commonly used technique to find bugs and performance issues. Merging the technique of machine learning with finding network implementation flaws could potentially be a solution to the issues of defining implementation rules. This is because patterns can be defined instead of manually crafted, to which machine learning can attribute.

Another topic for future research would be to make a parser for the parsed output of AST, as currently the nodes have been stripped for keywords for the proof of concept rather than a simpler structure such as JSON objects. Establishing a simpler structure could allow analyzers to process the output with more ease and potentially allow machine learning to be more effective.

Future research can also be done on the complexity of lower level programming languages, such as C/C++. It is possible that the output of an AST will show more detail since lower level programming languages require the context in order to function. Consequently, analyzers can more easily process the static code because there could be less room for confusion, such as the imports in Python.

Finally, alternatives can be sought for the application of ASTs in relation to analyzing networking code, like e.g. dynamic code analysis. The use of dynamic code analysis in combination with ASTs could prove beneficial to negate its weaknesses and limitations, since it is focused on performing run time analysis. The analyzer could likely find dead code with more ease, as it is aware of the network messages - as well as its parameters - that are being utilized.

References

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[2] Visual Studio Code. Visual studio code may 2019. https://code.visualstudio.com/updates/v1_35, 2019. Accessed on July 6th 2019.

[3] Python Software Foundation. ast — abstract syntax trees — python 3.7.4rc1 documentation. https://docs.python.org/3/library/ast.html, 2019. Accessed on July 2nd 2019.

[4] Python Software Foundation. Python release python 3.6.8 — python.org. https://www.python.org/downloads/release/python-368/, 2019. Accessed on July 9th 2019.

[5] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[6] Ivar Slotboom. Github - f13rce/impscan: Analyzes static code to find implementation issues in regards to networking. https://GitHub.com/f13rce/ImpSCAN, 2019. Accessed on July 3rd 2019.

[7] A. Tasnim and M. R. Rahman. Inferring bug patterns for detecting bugs in javascript by analyzing abstract syntax tree. pages 503–507, June 2018.