Improving Performance and Security through Linting

Rose C. Howell Wesley P. Coomber Steven W. Sprecher Jr. Kristen N. Escher
University of Michigan
{rchowell, wcoomber, swsprec, kescher}@umich.edu

Abstract

Linters statically analyze the syntax of code as a programmer is typing it, and can provide immediate feedback on a wide range of bugs, error-prone code, and stylistic violations. This has many benefits, including training the programmer to write better code in the first place, saving time during code review, and preventing errors from propagating through the code before it becomes a bigger problem. In this project, we forked the flake8-eyeo plugin and added new rules to it that improve both performance and security.

Categories and Subject Descriptors

D.2.1 [*Software Engineering*]: Requirements/Specifications; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; I.1.1 [*Symbolic and Algebraic Manipulation*]: Expressions and Their Representation

Keywords

Abstract syntax tree, Linting, Python

1. Introduction

Linting is one of the more convenient and productive methods to catch and fix bugs earlier in software development. This form of static analysis allows programmers to supplement their future test-cases and improve their initial code without diverting effort to premature writing of specific test cases. In this paper, we implement and evaluate a new plug-in for the flake8 linter tool. This plug-in is composed of a number of rules that evaluate three key categories of proper code development: security analysis, dead code elimination, and hoist-able loop invariant detection.

1.1 What is linting?

According to the Unix manual page, "the lint utility attempts to detect features ... that are likely to be bugs, to be non-portable, or to be wasteful," [16]. Most companies who employ software developers have basic rules for coding style and practices that they require all code to conform to. Typically, if your code doesn't follow these rules, then it doesn't get merged into the code-base.


```
> flake8 tests/A208.py
tests/A208.py:4:4: A208 `a == x or a == y` should be `a in {x, y}`
tests/A208.py:8:4: A208 `a == x or a == y` should be `a in {x, y}`
tests/A208.py:12:4: A208 `a == x or a == y` should be `a in {x, y}`
tests/A208.py:16:4: A208 `a == x or a == y` should be `a in {x, y}`
```

Figure 1. Stand-alone flake8

Sometimes these rules are enforced in a code review process, but this can be time-consuming and expensive. Linting is a process that automates some of the checks that would otherwise be done in code review. Linting can also help catch bugs while you're writing your code, even before compiling or running it [9].

1.2 How do you lint?

Linters are very versatile and there are many different ways to integrate a linter into your coding environment. A company is able to write standard linter rules for all of its developers, and then each developer can choose how they would like to implement and interact with the linter. Two of the most commonly used options are as a stand-alone linter, or as a plugin to an IDE (integrated development environment).

1.2.1 Stand Alone Linter

When used as a stand-alone utility, the input will be the file you would like to lint, and the output will be the issues that the linter catches, along with helpful information like the line number, the offending code, and the rule that it breaks [12]. In Figure 1, we are running the stand-alone flake8 linter against a test file. We can see the output shows four bugs, on lines 4, 8, 12, and 16, along with the specific error code, `A208`, and a helpful suggestion for how to fix the issue.

1.2.2 IDE plugin Linter

Figure 2 shows the SublimeLinter plugin, running the same flake8 linter automatically in the Sublime IDE as a developer is writing code. This plugin is configured to put a dot in the gutter on the offending line, underline the offending node, and display the debug information when the mouse hovers over the node, but the plug-in is highly configurable and can be modified to suit many different linting preference styles. Linter plugins are available for most major IDEs, including Apple's Xcode, Microsoft's Visual Studio, and other open source IDEs like NetBeans.

1.3 Linting State of the Art

There are many different options for linters. Pylint and flake8 are two popular choices today.

1.3.1 Pylint

Pylint was released 16 years ago, in 2001, and has been a popular choice for open source projects such as Xen and Mercurial [14].

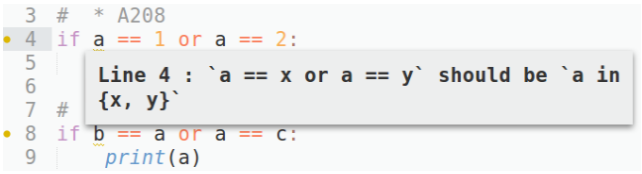

Figure 2. SublimeLinter flake8 plug-in

Table 1. Number of Github open source projects using various programming languages

Language	Number of Projects
Python	45,847
Java	35,672
Ruby	10,993
C	9,171
C++	6,554
Rust	4,267
Rails	4,005

Pylint is not being used very often anymore, except for some older legacy projects, mainly because it is too large, slow, and often produces an unmanageable amount of output. [18].

Pylint is quite robust, and under the hood uses the astroid python module, which is a Pylint specific abstract syntax tree (ast) wrapper. Pylint uses this to parse the code to find errors. The astroid module allows Pylint to gain more context about a certain node in the ast, and can even do something called inference. Inference is where an astroid node can use the extra context it keeps track of to resolve the value of a node without actually running any code.

This robustness makes astroid, and by extension Pylint powerful, but it also makes it bloated.

1.3.2 Flake8

Flake8 is newer and rising in popularity, according to open source developers we interviewed [8]. It is used by Django [3] and eyeo (maker of Adblock Plus), among many others.

Flake8 acts as a framework within which developers can add rules and extensions. By default, flake8 comes with three default ruleset plugins: pycodestyle for enforcing the PEP8 style guidelines, pyflakes for detecting various coding errors, and McCabe for detecting overly complex code. There are many other plugins that can enforce things like ordering your `import` statements correctly, naming your variables better, providing good docstrings, and checking for security vulnerabilities [4]. It is also fairly straightforward to create a novel plugin for specialized rules, as eyeo did with flake8-eyeo.

2. Motivation

2.1 Why focus on Python?

We chose to work with Python for a few reasons. First, it is very widely used in the open source community. In Table 1 we see that it is more popular than all the other languages included in the table by at least 4x (except Java). Also, in our interviews with open source developers, they mentioned that Python was very widely used throughout the open source community [8].

Second, a few of us work in security, and find that Python is an invaluable and ubiquitous language for almost any small task (and many larger ones), and we are interested in improving the quality of Python code produced with regard to safety and optimization.

2.2 Linting is Valuable

2.2.1 Code review is one of the most important tasks in software development.

Studies have found that "developers spend more time understanding code than any other activity" and that "code review is one of the most effective quality assurance techniques in software development," [5]. After speaking with some of the developers at 'eyeo', we found that there was some frustration about finding similar bugs recurring over and over [8]. If these issues could be automatically caught by a linter while a developer is writing code, then they would never make it to the later code review process, and therefore saves the reviewer's time and energy.

2.2.2 Good code is easier to debug and maintain.

Large projects with many contributors can become difficult to maintain, especially if different coding practices and styles are followed in different places. The consistency that linters enforce helps make the project easier to debug, and easier to maintain in the future [10].

2.2.3 Cost benefits of catching errors early.

IBM reported that catching bugs earlier in the software development life-cycle is cheaper by orders of magnitude. They found that the average cost of catching a defect during coding was $25, "rising to $100 at build time, $450 during testing/QA, and $16,000 post-release," [6]. Linters can dramatically reduce costs for companies by helping to catch defects during the coding phase [1].

2.3 Why focus on flake8?

We chose to work with flake8 for this project for a number of reasons. First, it is the current favorite for modern open source projects according to developers we interviewed [8]. Second, we had access to some simple flake8 extensions already written by the developers at eyeo, and permission to contribute. Third, we were interested in the modular plug-in contribution format of flake8 which makes it widely extensible. The modular design of the flake8 linter, allows us to write a novel plug-in that implements new linter rules and warnings, and then release this finished project as an open-source plug-in for flake8. We are extending the plugin with 6 additional new rules.

2.4 Compiled vs Interpreted

Python is an interpreted language, meaning that when a Python file is executed, each line will be converted to byte code (or rather matched to byte code already in the interpreter) and executed. What this saves in overhead, we lose in other areas, especially in optimization.

Compiler optimization is a long studied area of work, and is applied very successfully to well known compiled languages (like C, C++, etc.). Since Python is never compiled, the code often doesn't contain even very basic optimization. Using linting, we hope to bring a few compiler like optimizations right to the programmer, to aid them in creating faster and more effective Python code. We've chosen to implement warnings for dead code and loop invariant code. Additionally, since a few of our authors work in security, and static analysis is a proven-useful tool in secure software development [11], we decided to write a few security warnings.

3. Rules and Implementation

3.1 Dead Code

We implemented three different dead code warnings, for varying complexity of dead code, a summary of which is in Table 2

Table 2. Dead Code Warning Codes and Details

Code	Example	Details
A421	`if(0)`	`if()` w/ constant number equal to 0
A422	`if(false)`	`if()` w/ constant boolean false
A423	`foo=0 ... if(foo)`	`if()` w/ indirect false

3.1.1 Rule A421 and Rule A422

A421 and A422 are the error codes that we use for the constant value dead code checks. For these new rules, we created a new `_check_if()` function that is called whenever the overloaded `visit_If()` linter function is run. In this dead code check, we traverse the AST to identify every `if()` statement, and then verify whether the conditional statement is evaluating a constant integer, float, or complex value. If the AST node is evaluating a constant number, then we run a simple check to see if that constant is equal to zero; because this means that the code executed within the `if` statement is unconditionally unreachable dead code.

For the A422 rule, we implemented additional functionality to the `_check_if()` function. In this new dead code check, we traverse the abstract syntax tree in the same manner as the aforementioned A421 rule, but we instead check whether the `If()` statement node is evaluating a constant Boolean value. If the node is indeed checking a constant Boolean value, then we run a simple python comparison to verify whether the Boolean is False and we raise a linter warning. In the case of an `If(False)` statement, we can see that the block of code dependent on that statement is dead code that is unconditionally unreachable.

3.1.2 Rule A423

This is the error code that we use for a more complex dead code check. This new rule warns the programmer about dead code that is indirectly unconditionally unreachable. For example, in a simple python program, there may be an assignment statement like "`foo = 0`" followed by a conditional statement like "`If(foo)`". One can see that in the case of variable `foo` deterministically and unconditionally having a value of 0 when the `If(foo)` statement is evaluated, then the block of code dependent on the `If(foo)` statement is dead code.

To achieve this functionality, we override the `visit_Assign()` linter function that is called for every assignment node in the AST. For every variable assignment, we create (or update) an entry in a python dictionary that tracks the current value of each variable in the program. This means when we explore the AST to evaluate an `If()` statement node, if we are branching conditionally on a variable `foo`, then we iterate through our dictionary data structure to find the live variable value at the time of comparison. This means we can effectively detect dead code blocks within `If()` statements that are indirectly unconditionally unreachable. This is demonstrated in the 'helloWorldTest.py' test case. There are some limitations to this approach because our linter statically evaluates the code, thus it can only flag and warn the programmer about dead code that is unconditionally unreachable. It also can not evaluate variable values that are dependent on user input. This is not a limitation of our particular implementation but a limitation based on the fundamentals of static code analysis and linters.

3.2 Loop Invariant Assignments

We implemented a rule that recommends hoisting a line of invariant code from a loop. As the Python compiler does not do optimization [7], such redundant assignments must be manually corrected.

3.2.1 Rule A200 - Constant assignment within a loop

This error code indicates that a constant value, either a string or a number, has been assigned to a variable within a loop. This error is only triggered when there are no intervening assignments between loop iterations.

We traverse the AST to identify every `Assign` and `AugAssign` node that is a child of a `For` or `While` node. As `Num` and `Str` types are immutable in Python [2], these variables cannot be changed by passing them as function arguments, so only explicit assignments require analysis. Our recursive function `_check_operands_constant()` is used to check the operands of any assignment to a `Name` target and determines that an assignment is constant if all operands are of type `Num` or `Str`. Our rule accounts for chaining of assignments (e.g. `a = b = 3`), augmented assignments (e.g. `a += 3`), and chained binary operations (e.g. `a = 3 + 7 * 10 ** 2`). To allow analysis of assignments within nested loops, we also record the depth of the loop in which the assignment occurs.

We created a new `_check_hoistable_line()` function that is called after the overloaded `visit_For()` and `visit_While()` linter functions visit a loop's child nodes. An error is generated if an assignment has been made to a variable that is not redefined at the same loop depth.

The error is suppressed if the same variable has been used at a previous line number at the same loop depth. This allows for logic that runs the first iteration of a loop with the variable holding one value, and all subsequent iterations with a different value.

3.3 Security

We implemented two new security checks: one check for insecure hash functions such as MD5 [17], and one check for insecure cipher block modes such as ECB. These are shown in Table 3.

3.3.1 Rule A370 - Insecure hash functions

This rule checks for use of insecure hash functions MD2, MD4, MD5, SHA, and SHA-1. SHA-1 was suspected to be vulnerable for some time, but recently Google found an actual hash collision [15]. To check for these errors, we modified the `TreeVisitor` class that is derived from Python's `ast` module. Each time the `TreeVisitor` visits a function call or evaluates an expression, it will also check if the program is calling one of these insecure hash functions from the Python `hashlib` module (a popular Python hash library).

3.3.2 Rule A371 - Insecure cipher block modes

Most popular encryption algorithms work on blocks of a fixed size. If the data to be encrypted is smaller than the block size, it must be padded. If the data is larger then the block size, then the data is broken into blocks, and the encrypted blocks are either ordered sequentially (ECB) or chained together (CBC) to form larger ciphertexts. But, both of these modes of operation are known to be vulnerable. ECB mode leaks information about the underlying data, and is vulnerable to replay attacks. CBC mode is vulnerable to padding oracle attacks.

The implementation of this rule is similar to that in A370. The `treeVisitor` walks the AST, checking each function call and evaluating each expression for use of these insecure cipher block modes.

4. Results

We collected over 700,000 lines of open source python code from github to test the effectiveness of our linter plugin. As shown in Table 5, we first ran the flake8 linter, without the eyeo plugin on all of our code base, which gave 66,130 warnings. We then ran flake8-eyeo which yielded 97,008 warnings. Finally, we ran flake8-eyeo

Table 3. Insecure Code Warning Codes and Details

Code	Example	Details
A370	`hashlib.md5(1)`	MD5 is a vulnerable hashing algorithm
A371	`AES.new(key, AES.MODE_ECB)`	ECB is a vulnerable cipher block mode

Table 4. Specific Error code hits on data set

Error Code	# Hits	Percentage
A200	77	74.03 %
A370	21	20.19 %
A371	1	0.96 %
A421	0	0.00 %
A422	3	2.88 %
A423	2	1.92 %

```
1081        # without replacement
1082        n_row = 3
1083        for n_col in range(4, 5 + 1):
1084            prob_dist = torch.rand(n_row, n_col)
1085            prob_dist.select(1, n_col - 1).fill_(0)  # index n_col shouldn't be sampled
1086            n_sample = 3
1087            sample_indices = torch.multinomial(prob_dist, n_sample, False)
1088            self.assertEqual(prob_dist.dim(), 2)
1089            self.assertEqual(sample_indices.size(1), n_sample)
1090            for i in range(n_row):
1091                row_samples = {}
1092                for j in range(n_sample):
1093                    sample_idx = sample_indices[i, j]
1094                    self.assertNotEqual(sample_idx, n_col - 1,
1095                                        "sampled an index with zero probability")
1096                    self.assertNotIn(sample_idx, row_samples, "sampled an index twice")
1097                    row_samples[sample_idx] = True
```

Figure 3. A200 on line 1086

```
25 def md5_text(s):
26     if not isinstance(s, compat_str):
27         s = compat_str(s)
28     return hashlib.md5(s.encode('utf-8')).hexdigest()
29
```

Figure 4. A370 on line 28

```
163                 return False
164             if decrypt_info['METHOD'] == 'AES-128':
165                 iv = decrypt_info.get('IV') or compat_struct_pack('>8xq', media_sequence)
166                 decrypt_info['KEY'] = decrypt_info.get('KEY') or self.ydl.urlopen(decrypt_info['URI']).read()
167                 frag_content = AES.new(
168                     decrypt_info['KEY'], AES.MODE_CBC, iv).decrypt(frag_content)
169             self._append_fragment(ctx, frag_content)
170             # We only download the first fragment during the test
171             if test:
172                 break
173             i += 1
174             media_sequence += 1
```

Figure 5. A371 on line 168

with our added rules and we genereated 97,162 warnings. That is 154 more warnings than the flake8-eyeo.

We didn't expect the number of warnings from our plugin to be much larger than observed for a few reasons. Most of the open source code in question is older mature code that has already been through formal code reviews. Additionally the open source community has identified and fixed many bugs in the example programs. This helps cut back on bugs, like those we identify with our rules, but yet still some bugs get through.

A200 was by far our most successful rule as shown in Table 4, in that it warns on loop invariant code that can be hoisted outside of the loop. We see in Figure 3, on line 1086, that `n_sample = 3` is within the for loop. We also see that `n_sample` is never modified in the for loop. Thus, the program is redoing this assignment for each iteration, and would benefit if that line was lifted out of the loop.

From a security perspective, the cryptographic hash function md5 is lacking. It is trivial to generate collisions, and md5 has been called upon to be deprecated for many years now. We see in Figure 4 on line 28, that some open source projects still use insecure functions like md5 either knowingly, or rather more hopefully, unknowingly and that in our test our rule A370 found a non negligible number of errors as shown in Table 4.

```
346             if outlier_label is None:
347                 assert_raises(ValueError, clf.predict, z2)
348             elif False:
349                 assert_array_equal(np.array([1, outlier_label]),
350                                    clf.predict(z2))
```

Figure 6. A422 on line 348 of test_neighbors.py

Additionally from a security perspective, the block cipher mode called cipher block chaining (CBC) can be extremely vulnerable to attacks, such as a padding oracle attack. This would allow an attacker to decrypt any message sent using this scheme in linear time. As we see in Figure 5 on line 168, programmers, who hopefully are just unaware of it's vulnerabilities, still utilize this mode of encryption.

Finally, programmers sometimes write code that is not ever actually executed, as seen in Figure 6 on line 348. The code within that `elif` statement will never run, seeing as `elif False` will always be `False`. Using a light-weight linting tool with our plug-in would warn the developer about this dead code before release.

4.1 Data Accuracy

To test our rules for false positives, we wrote specific tests to check all the edge cases we could predict. There are a myriad number of ways to program, both correctly and incorrectly, and thus we expected to see some false positives when running our plug-in on our data set. We hand-verified all the linter warnings for the A422 and A423 rules. Across the three A422 warnings, we verified three accurately identified dead code bugs in the files: `test_neighbors.py`, `graph_utils.py`, and `nav_utils.py`. So within our limited sample size we observed a high 100% accuracy rate for the A422 dead code rule. For the two A423 warnings, we hand-verified the warnings and observed a single false positive in `tensorflow_backend.py`, along with an accurate positive detection of dead code in `polygon.py`. The A423 rule had a high 50% false positive rate, but because of the extremely limited sample size of 2, this high false positive rate may not be representative of the overall efficacy of the linter rule, and further investigation is warranted. We opted to do some spot checking on the files that our errors triggered on and found that A200, A370, and A371 seemed to not have many (if any) false positives. A421 didn't trigger on any files, which makes sense since the code we've run on has been reviewed before being published. This is because of the difficulty in learning context using the ast available to flake8. Essentially, it made it far too difficult in the scope of this project to attempt to track a variable over conditional statements. We leave that task for future work.

5. Related Work

While this paper focuses on flake8, we are also knowledgeable about another Python linter called Pylint and the astroid project it uses. One of the benefits of flake8 is that it is more lightweight and faster than Pylint, but conversely it has less functionality. While both projects use the Python AST module, astroid has built-in much more context functionality. In the generic AST module, a node

Table 5. Errors / Warnings Caught on 700,000+ lines of open source python

Linter	Version Details	Number of Errors
flake8	3.5.0 (mccabe: 0.6.1, pycodestyle: 2.3.1, pyflakes: 1.6.0)	66,130
flake8-eyeo	3.5.0 (eyeo: 0.1, mccabe: 0.6.1, pycodestyle: 2.3.1, pyflakes: 1.6.0)	97,008
flake8-eyeo + our plugin	3.5.0 (eyeo: 0.1, mccabe: 0.6.1, pycodestyle: 2.3.1, pyflakes: 1.6.0)	97,162

does not know much about its context, for example if it is in a loop or not. Knowing this context allows Pylint developers more freedom to make rules that can use this information. For example, they can safely unroll and evaluate loops and more accurately flag warnings for inefficient loop-invariant code [13]. There has also been previous work into employing lightweight static analysis to improve program security[4]. However, this work was a static analysis tool for the C programming language, and their research focused on different security concerns than our Python language linter plug-in.

6. Conclusions and Future Work

Linting over the most popular open source github projects that use python with our plugin to flake8-eyeo, we see that there are still some optimization and security issues with code in production. As we discussed, the cost of fixing these flaws in their current state is exponentially more expensive than if they were caught during development. Given that logic, one may think that the more linting rules you can make the better, but in reality, it can be difficult to strike the right balance in choosing what to lint for. As with security, if you make the rules too strict, people will grow weary of them, or find ways to subvert them. In security, this is known as "warning fatigue". Future research into how strict linting rules can be before programmers begin to subvert them would be valuable research in this field.

Our work taught us that to implement a good linting rule for python, a developer should understand how to walk the abstract syntax tree and write code that will statically analyze it to check for bad behavior. A good test suite that includes positive and negative test cases, along with potential false positive and false negative test cases should also be included.

6.1 Future Work

We hope that more robust rules for dead code, recursive inference loop invariant code, and more in depth security rules will be implemented, either by us or by the open source community going forward. Because of the limited scope of this class project, we were only able to evaluate our linter rules on a set of mature and already-tested open-source code projects. The fundamental motivation behind linters is to catch bugs early-on in development, so in the future–we hope to continue our work by having field studies with real developers using our linter plug-in while they are writing novel code for new features and projects. We expect that our new linter plug-in would prove to be most helpful at this stage of software development.

7. Contributions

Our team worked well together and we divided the work in a fair and equal manner. Rose and Steven were in charge of implementing the linter plugin's security rules involving insecure hashing algorithms (MD5, MD4, MD2 and SHA) and insecure cipher block chaining modes (ECB and CBC). Wesley wrote the linter rules to identify unconditionally unreachable dead code, and wrote the fundamental design for identification of insecure usage of hashing libraries. Finally, Kristen implemented the hoistable loop-invariant code warning for our new flake8 plugin. For the group presentations and paper-writing, we met up in-person to all collaboratively work on these team-shared deliverables. Overall the group is content with the work distribution and thus our comprehensive project contributions can be broken down into 25%/25%/25%/25% for the members: Rose, Steven, Wesley, and Kristen.

References

[1] BACA, D., CARLSSON, B., AND LUNDBERG, L. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security* (2008), ACM, pp. 79–88.

[2] DAWSON, M. *Python Programming for the Absolute Beginner*. Cengage Learning, 2010.

[3] DJANGO SOFTWARE FOUNDATION. Coding style — django documentation. `https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/`, 2005.

[4] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE software 19*, 1 (2002), 42–51.

[5] FLOYD, B., SANTANDER, T., AND WEIMER, W. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (May 2017), pp. 175–186.

[6] GOUES, C. L., DEWEY-VOGT, M., FORREST, S., AND WEIMER, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *2012 34th International Conference on Software Engineering (ICSE)* (Jun 2012).

[7] GUELTON, S., B. P. A. M. M. A. C. X., AND RAYNAUD, A. Pythran: enabling static optimization of scientific python programs. *Computational Science & Discovery* (2015).

[8] HOWELL, R., NOACK, S., AND KUZNETSOV, V. Linters in industry, Nov 2017.

[9] JOHNSON, S. C. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[10] KNUPP, J. How python linters will save your large python project, Dec 2016.

[11] LI, P., AND CUI, B. A comparative study on software vulnerability static analysis techniques and tools. In *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on* (2010), IEEE, pp. 521–524.

[12] OWEN, R. Python Linters. In *Python in Astronomy 2016* (Mar. 2016), p. 28.

[13] POPA, C. 12 years of pylint (or how i learned to stop worrying about bugs). `https://ep2015.europython.eu/conference/talks/12-years-of-pylint-or-how-i-stopped-worrying-and-love-the-bugs`, Jul 2015.

[14] PYLINT. Misc. projects using pylint. `https://docs.pylint.org/en/pylint-1.4.1/backlinks.html`, 2013.

[15] STEVENS, M., BURSZTEIN, E., KARPMAN, P., ALBERTINI, A., AND MARKOV, Y. The first collision for full sha-1. `https://shattered.it/static/shattered.pdf`, 2017.

[16] THE UNIX AND LINUX FORUMS. Freebsd 11.0 - man page for lint (freebsd section 1). `https://www.unix.com/man-page/FreeBSD/1/lint`, May 2001.

[17] TURNER, S., AND CHEN, L. Updated security considerations for the md5 message-digest and the hmac-md5 algorithms.

[18] TURNER-TRAURING, I. Why pylint is both useful and unusable, and how you can actually use it. `https://codewithoutrules.com/2016/10/19/pylint/`, 2016.