POSTER: Precise Detection of Unprecedented Python Cryptographic Misuses Using On-Demand Analysis

Miles Frantz, Ya Xiao, Tanmoy Sarkar Pias, Danfeng (Daphne) Yao
Computer Science, Virginia Tech, Blacksburg, VA
{frantzme, yax99, tanmoysarkar, danfeng}@vt.edu

***Abstract*—While many research studies target Java cryptographic API misuses, similar issues within the Python landscape are still uncovered. In this work, we provide 1) security guarantees for complex Python cryptographic code through the use of our tool, Cryptolation, and 2) a basis for understanding the practices of Python cryptographic API misuses and detection through a thorough analysis and a state-of-the-art benchmark. Cryptolation is a Static Code Analysis (SCA) tool that conducts an inter-procedural data-flow analysis and successfully handles many Python features through inference and context injection. Our state-of-the-art benchmark, PyCryptoBench, includes 228 basic and advanced insecure cases to evaluate our tool and provide a framework for future evaluation and comparison of competing tools. We evaluate Cryptolation and other state-of-the-art tools Bandit, Licma, Semgrep, and DLint against our benchmark and 1000 open-source projects. Overall, Cryptolation provides more insight when scanning Python projects and our benchmark compares state-of-the-art tools against several programming patterns.**

Keywords*—*static code analysis, cryptographic API misuses, Python, benchmark

I. INTRODUCTION

Many studies have shown how cryptographic API misuses result in security vulnerabilities [1], [2], [3], [4]. These studies motivate a line of research into SCA to find these API misuses [5], [6], [7], [8]. Since most attention is paid to Java or C applications using cryptographic libraries, Python cryptographic APIs have not been thoroughly examined. We focus on 59 Python cryptographic modules, including `PyCrypto` [9] and `PyJWT` [10] that are popular in Python [11]. We mapped 18 security rules to the API misuses that violate them. We leverage Astroid [12] to create our Abstract Syntax Tree (AST) since it makes variable inferences. Astroid attempts to infer the potential values of a variable within the AST. This, in turn, lets us create separate AST to extend our data flow analysis through the different inferences of the variables. Creating the separate AST decreases the performance but increases the Precision and Recall while decreasing the False Negative (FN).

II. CHALLENGES

Shown in Figure 1 is a simple insecure path-sensitive hash example. The developer imported the hashlib module but changed the value based on the conditional at line 2. This snippet creates a simple AST to parse but requires the SCA tool to track the default hash value propagation. SCA tools will have to either operate in a path-insensitive manner or evaluate the conditional on identifying the correct path. Developers may not make use of their imports and we do not create False Positive (FP) alerts on unused imports. We do live-import analysis by identifying uses of the imports to ensure it is a cryptographic misuse and not a dead import. Analyzers need to trace the import propagation and reaching definitions of assigning the imports. Cryptolation uses the inferred values of variables provided by the AST to trace the variable propagation. We also use a path-insensitive approach through the variable inference to ensure complete coverage and lower FN.

```
1 import hashlib
2 if True:
3     _default_hash = hashlib.sha1
4 else:
5     _default_hash = hashlib.sha256
6 print(_default_hash(b”HelloWorld”))
```

Listing 1. An insecure hash Python sample. The default hash method is set to sha 1 at line 3 then the developer calls the default hash on a custom string at line 6. Static analyzers must identify the hashlib import, use path-insensitive flow to identify the vulnerable import at line 3, and identify the use at line 6. This sample is similar to several samples seen through testing.

III. EVALUATION

Cryptolation is framework-agnostic and general to the language itself; thus, we compared against tools that scan Python code. We compared Cryptolation against the cryptographic results of Bandit [13], Semgrep [14], Licma [15], and Dlint [16]. We created the basic and advanced PyCryptoBench benchmark, which evaluates the tools against a standard set of insecure cryptographic practices derived from [8]. The 38 basic cases are basic files testing each rule. The 190 advanced cases include global, inter-procedural, inter-procedural at two-level, path-insensitive, and field-insensitive test files. We also scanned the nine famous Python projects chosen by their maturity and how much of the repository was Python; keras, ansible, scrapy, IntelOwl, requests, core, httpie, Django, and flask. We also used 1008 projects from GitHub if they were a Python repository tagged with either “payments” or “cryptography”. Table II is the breakdown of True Positive (TP) alerts, Precision, Recall, Accuracy, and F1 scores. Cryptolation outperforms all other tools during the benchmark examination, with DLint coming in a close second. DLint outperformed Cryptolation within Tagged Projects by a minimal gap since it identifies the vulnerabilities based on imports.This approach could lead to False Positives if the import is included but unused. Licma focuses on hybrid-based Python projects with a smaller cryptographic scope. Cryptolation has a precision of 99.7% while having more than 6,000 alerts compared to the nearest tool DLint. Due to the massive quantity of files,

TABLE I. The cryptographic vulnerability, attack type, and cryptographic property per vulnerability. The severity levels are denoted H/M/L for High, Medium, and Low risk. The cryptographic properties C/I/A are Confidentiality, Integrity, and Authenticity.

#	Vulnerability	Attack Type	Crypto. Property	Severity
1	Predictable/Constant Cryptographic keys	Predictability	Confidentiality	H
2	Use Wildcard Verifiers to Accept All Hosts	SSL/TLS MitM	C/I/A	H
3	Create Custom String to Trust All Certificates		C/I/A	H
4	Create Unverified HTTPS Context		C/I/A	H
5	Use of HTTP		C/I/A	H
6	Cryptographically Insecure PRNGs	Predictability	Confidentiality	M
7	Static Salts	CPA	Confidentiality	M
8	ECB Mode in Symmetric Ciphers		Confidentiality	M
9	Fewer than 1,000 Iterations for creating Salt	Brute Force	Confidentiality	L
10	Insecure block ciphers (e.g., IDEA, Blowfish)		Confidentiality	L
11	Insecure asymmetric ciphers (e.g, RSA, ECC)		C/A	L
12	Insecure cryptographic hash (e.g., SHA1, MD5)		Integrity	H
13	Not Verifying a Json Web Token	SSL/TLS MitM	I/A	H
14	Using an insecure TLS Version		Confidentiality	H
15	Using an Insecure Protocol		C/I/A	H
16	Using an insecure XML Deserialization	Deserialization	Confidentiality	M
17	Using an insecure YAML Deserialization			M
18	Using an insecure Pickle Deserialization			H
19	Not escaping a regular expression	Brute Force	Integrity	M

the Massive and Tagged project scans do not have ground truth. We reviewed their results by automatically retrieving the code snippets identified by line numbers and programmatically identifying specific libraries.

TABLE II. The breakdown of the precision and recall per tool per scanning type. The benchmark is a custom and open-sourced data set while the Full Tests and Massive Scan Types are live projects pulled from GitHub.

Batch Scan	Tool Name	TP	Precision	Recall	Accuracy	F1
Benchmark	Bandit	86	100%	37.70%	100%	54.8%
	Cryptolation	**108**	100%	**47.40%**	100%	**64.3%**
	DLint	104	100%	45.60%	100%	62.7%
	Licma	10	100%	4.40%	100%	8.4%
	Semgrep	52	100%	22.80%	100%	37.1%
Tagged	Bandit	3001	89.7%	100%	89.7%	94.6%
	Cryptolation	**16471**	99.8%	100%	99.8%	99.9%
	DLint	6482	**99.9%**	100%	**99.9%**	**100%**
	Licma	0	0%	0%	0%	0%
	Semgrep	2213	100%	100%	100%	100%
Massive	Bandit	164	92.7%	100%	92.70%	96.2%
	Cryptolation	288	100%	100%	100%	100%
	DLint	356	100%	100%	100%	100%
	Licma	0	0%	0%	0%	0%
	Semgrep	109	100%	100%	100%	100%

IV. Conclusion and Ongoing Work

We created Cryptolation to examine and discover potential cryptographic misuse for complex programming patterns. Our benchmark PyCryptoBench provides 228 files that cover several complex programming patterns. When evaluating against PyCryptoBench, Cryptolation provides improved Recall and F1 on complex patterns compared to the state-of-the-art tools. We will further evaluate these tools against more projects based on their McCabe Cyclomatic Complexity (MCC) score.

Acknowledgment

This work has been supported by the National Science Foundation under Grant No. CNS-1929701.

References

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 289–305.

[2] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in Java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 372–383.

[3] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 536–547.

[4] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it," in *Proceedings of the 29 th USENIX Security Symposium (USENIX) Security*, vol. 20, 2020.

[5] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[6] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLint," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 519–534.

[7] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "CogniCrypt: supporting developers in using cryptography," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[8] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2455–2472.

[9] "PyCrypto - The Python Cryptography Toolkit," https://www.dlitz.net/software/pycrypto/.

[10] "Welcome to PyJWT," https://pyjwt.readthedocs.io/en/stable/#.

[11] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 154–171.

[12] PyCQA, "GitHub - PyCQA/astroid: A common base representation of python source code for pylint and other projects." [Online]. Available: https://github.com/PyCQA/astroid

[13] Bandit, "Welcome to Bandit's developer documentation! — Bandit documentation." [Online]. Available: https://bandit.readthedocs.io/en/latest/

[14] "Semgrep," Feb 2022, [Online; accessed 8. Feb. 2022]. [Online]. Available: https://semgrep.dev

[15] A.-K. Wickert, L. Baumgärtner, F. Breitfelder, and M. Mezini, "Python Crypto Misuses in the Wild," vol. 21, pp. 1–6, sep 2021. [Online]. Available: http://arxiv.org/abs/2109.01109http://dx.doi.org/10.1145/3475716.3484195

[16] duo labs, "dlint," Feb 2022, [Online; accessed 8. Feb. 2022]. [Online]. Available: https://github.com/duo-labs/dlint

POSTER: Precise Detection of Unprecedented Python Cryptographic Misuses Using On-Demand Analysis

Miles Frantz, Ya Xiao, Tanmoy Sarkar Pias, and Danfeng (Daphne) Yao
Computer Science, Virginia Tech, Blacksburg, VA 24060, USA
{frantzme, yax99, tanmoysarkar, danfeng}@vt.edu

1. Motivation

Python is used by practitioners of various levels of coding experience

The support for static analysis on Python is way behind other common programming languages[1].

Python is difficult for static analysis tools due to its dynamic nature[1]. Only optional hints, import aliasing, and functions as methods are a few issues.

The current cryptographic analysis projects mainly focus on frameworks.

We propose Cryptolation, a static analysis tool that scans Python code in a depth-insensitive and path-insensitive manner with 98% precision.

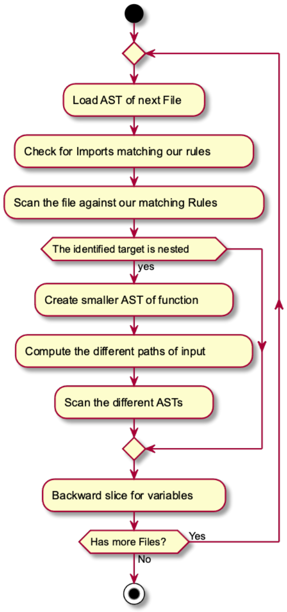
2. Rules

#	Vulnerability	Attack Type	Crypto. Property	Severity
1	Predictable/Constant Cryptographic keys	Predictability	Confidentiality	H
2	Use Wildcard Verifiers to Accept All Hosts	SSL/TLS MitM	C/I/A	H
3	Create Custom String to Trust All Certificates		C/I/A	H
4	Create Unverified HTTPS Context		C/I/A	H
5	Use of HTTP		C/I/A	H
6	Cryptographically Insecure PRNGs	Predictability	Confidentiality	M
7	Static Salts	CPA	Confidentiality	M
8	ECB Mode in Symmetric Ciphers		Confidentiality	M
9	Fewer than 1,000 Iterations for creating Salt	Brute Force	Confidentiality	L
10	Insecure block ciphers (e.g., IDEA, Blowfish)		Confidentiality	L
11	Insecure asymmetric ciphers (e.g, RSA, ECC)		C/A	L
12	Insecure cryptographic hash (e.g., SHA1, MD5)		Integrity	H
13	Not Verifying a Json Web Token	SSL/TLS MitM	I/A	H
14	Using an insecure TLS Version		Confidentiality	H
15	Using an Insecure Protocol		C/I/A	H
16	Using an insecure XML Deserialization	Deserialization	Confidentiality	M
17	Using an insecure Yaml Deserialization			M
18	Using an insecure Pickle Deserialization			H
19	Not escaping a regular expression	Brute Force	Integrity	M

Our 19 rules identify several popular attack vectors used within the OWASP top 10[2].

We allow developers to add their own custom rules as well.

3. Approach



We create and test a path-insensitive, inter-procedural, and depth-insensitive static analysis tool called Cryptolation.

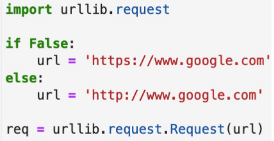
We leverage the AST to create a demand-driven analysis.

We scan the Python file if it contains any cryptographic imports we have rules for.

We create more ASTs if the identified cryptographic method is nested within multiple methods.

We continue slicing through the nested ASTs to determine the cryptographic misuse in a path-insensitive way.

4. Challenges



```
import urllib.request

if False:
    url = 'https://www.google.com'
else:
    url = 'http://www.google.com'

req = urllib.request.Request(url)
```

Python code allows for various types of aliasing, forcing analyzers to trace the variable propagation

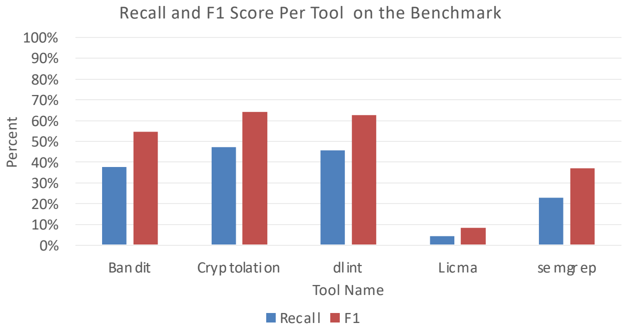
The code is vulnerable and uses HTTP despite being insecure. When code analysis is not path-sensitive, the malicious code is not identified.

Python allows developers to use functions as variables and doesn't require type definitions, restricting normal Static Analysis techniques.

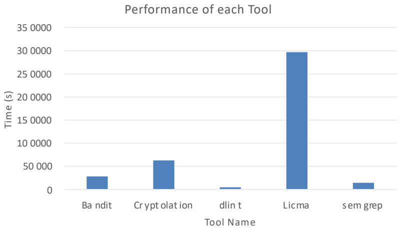
5. Benchmark and Tests

- We created PyCryptoBench to evaluate tools' performance against specific programming patterns. This benchmark is provided to researchers for future evaluation and tool comparison and includes the ground truth:
    - 38 Basic Files
    - 190 Advanced Files, using global, inter-procedural, field-sensitive, and path-sensitive programming patterns
- We also evaluated all the tools against nine major Python Projects such as keras and scrapy and more than 1000 python projects.

6. Preliminary Evaluation



Cryptolation has the highest recall and F1 scores with 47.4% and 64.33 %, respectively. The benchmark results are shown since we have the ground truth.



Licma took the longest since it creates a multilingual AST.

The other tools all took within 75% percentile of time, up to 63,362 seconds.

7. Ongoing Work

We present our tool Cryptolation that successfully scans complex Python code. We also present our open-source benchmark PyCryptoBench to provide samples for further tool evaluation.

[1] Gulabovska, Hristina, and Zoltán Porkoláb. "Survey on Static Analysis Tools of Python Programs." *SQAMIA*. 2019.
[2] "OWASP Top 10:2021." 15 Nov. 2021, owasp.org/Top10.

VirginiaTech
Invent the Future

Yao Group on Cyber Security
http://people.cs.vt.edu/danfeng/

NDSS SYMPOSIUM