Methods and Benchmark for Detecting Cryptographic API Misuses in Python

Miles Frantz, Ya Xiao, Tanmoy Sarkar Pias, Na Meng, Danfeng (Daphne) Yao
Virginia Tech
{frantzme, yax99, tanmoysarkar, nm8247, danfeng}@vt.edu

***Abstract*—Extensive research has been conducted to explore cryptographic API misuse in Java. However, despite the tremendous popularity of the Python language, uncovering similar issues has not been fully explored. The current static code analysis tools for Python are unable to scan the increasing complexity of the source code. This limitation decreases the analysis depth, resulting in more undetected cryptographic misuses. In this research, we propose Cryptolation, a Static Code Analysis (SCA) tool that provides security guarantees for complex Python cryptographic code. Most existing analysis tools for Python solely focus on specific Frameworks such as Django or Flask. However, using a SCA approach, Cryptolation focuses on the language and not any framework. Cryptolation performs an inter-procedural data-flow analysis to handle many Python language features through variable inference (statically predicting what the variable value is) and SCA. Cryptolation covers 59 Python cryptographic modules and can identify 18 potential cryptographic misuses that involve complex language features. In this paper, we also provide a comprehensive analysis and a state-of-the-art benchmark for understanding the Python cryptographic Application Program Interface (API) misuses and their detection. Our state-of-the-art benchmark PyCryptoBench includes 1,836 Python cryptographic test cases that covers both 18 cryptographic rules and five language features. PyCryptoBench also provides a framework for evaluating and comparing different cryptographic scanners for Python. To evaluate the performance of our proposed cryptographic Python scanner, we evaluated Cryptolation against three other state-of-the-art tools: Bandit, Semgrep, and Dlint. We evaluated these four tools using our benchmark PyCryptoBench and manual evaluation of (four Top-Ranked and 939 Un-Ranked) real-world projects. Our results reveal that, overall, Cryptolation achieved the highest precision throughout our testing; and the highest accuracy on our benchmark. Cryptolation had 100% precision on PyCryptoBench, and the highest precision on the real-world projects.**

***Index Terms*—static code analysis, cryptographic API misuses, Python, benchmark**

I. Introduction

Many studies show cryptographic API misuses result in security vulnerabilities [1], [2], [3], [4]. Similar studies showed that many developers often downgrade or reduce security by using cryptographic modules inappropriately. Developers may accidentally do this due to an API blindspot or a misunderstanding of the proper usage of an API [5]. For instance, Brun et al. [6] conducted a user study on API blindspots through programming puzzles and discovered Python developers were less likely to complete a puzzle with blindspots of API. Zhang et al. [7] conducted an empirical study on the effects of Stack Overflow responses on API misuse and discovered cryptographic API to be determined insecure at an average of around 42%. These studies show developers may not use the cryptographic API correctly or securely due to the complexity of the API.

This problem ultimately motivates a line of research to detect potential cryptographic API misuses [8], [9], [10], [11]. Since most of the attention is paid to Java or C applications using cryptographic libraries (e.g., JCA, JSSE, GNUTLS, etc.), Python cryptographic APIs have not been fully investigated. This has only recently been acknowledged since most analysis tools depend on the static type declaration system [12]. Python did not have this by design and only obtained optional type "hints" from Python 3.5+ [13]. These type hints cannot be definitively relied upon since a recent JetBrains Survey [14] shows that 29% of Python developers do not use type hints.

Our paper aims to find out the security practices of Python cryptographic modules and provide a static analyzer to detect them. We focus on the challenges derived from the complexity and flexibility of the Python programming language that makes static analysis imprecise. We encountered several different language features and were able to identify the potential cryptographic misuses using our specification successfully. This specification may include the usage of a vulnerable method or identifying whether a variable is set to a vulnerable value. We also create a benchmark PyCryptoBench[1], comprised of vulnerable and non-vulnerable test cases using different language feature evaluation. We focus on 59 Python cryptographic modules, including `PyCrypto` [15], `PyJWT` [16], that are popular in Python [17]. We create and map 18 security misuse patterns to the fully qualified API that violate it. The vulnerable usage includes using hard-coded or inappropriate values as the arguments of some methods.

Detecting cryptographic API misuse with SCA and Data-Flow Analysis has been previously reported in many studies [8], [11], [9]. However, as a dynamic programming language, Python SCA tools face unique challenges in precisely identifying the program properties. These challenges include but are certainly not limited to:

1) Understanding and identifying that a variable may contain a method, an import, or a value.
2) Setting the attributes of a Python object during run time.
3) Aliasing methods, variables, lambda functions, classes, and imports.
4) The global and local scopes of importing modules.

[1]The GitHub link for the PyCryptoBench will be provided upon acceptance.

5) Python allows its method calls to be determined at run time.

To our knowledge, we are the first work focusing on static analysis-based detection for Python cryptographic code and its capability for complex test cases with different Python language features. Our contributions include:

1) We investigated 59 Python cryptographic modules and the effects different language features had on the precision of SCA tools.
2) We present Cryptolation, an Interprocedural SCA tool that can detect cryptographic API misuses despite complex language features of Python; supported by a SCA supported by variable inference. The variable inference values are statically predicted values of the variable being examined supported by constant propagation. This allows Cryptolation to have increased precision and remain a SCA tool.
3) We provide a comprehensive cryptographic API misuse benchmark PyCryptoBench that covers a non-exhaustive list of five language features in Python. PyCryptoBench is a comprehensive benchmark composed of 1,836 files evaluating different cryptographic misuses to evaluate the precision of each static code analysis tool. PyCryptoBench comprises 1,530 basic test cases and 1,634 advanced test cases.
4) We conducted an experimental evaluation to demonstrate the capability of our tool Cryptolation and state-of-the-art (SOA) SCA tools Bandit, Dlint, and Semgrep in terms of capturing the complex patterns in our benchmark. These results were manually reviewed by two of the authors and prove Cryptolation achieves the highest precision and improvement in recall.
5) We validate Cryptolation's effectiveness of having 100% precision by evaluating with PyCryptoBench and Real-World Projects. The Real-World Projects include one Open-Source Top-Ranked project and 939 Un-Ranked Python Projects we've chosen from GitHub. The Top-Ranked project is composed of Scrapy; while the Un-Ranked Projects are comprised of Open-Sourced Projects from GitHub with the associated topics "payments" or "cryptography".

Our paper is organized as follows. Within Section II, we describe the threat model Cryptolation was created to handle. In Section III, we review the design techniques and analysis used within Cryptolation. Within IV, we go over the benchmarks we introduce for cryptographic tool evaluation. Within V, we go over the results and the methodology of the results.

II. Threat Model and Detection Challenges

In this section, we will cover the different misuse patterns Cryptolation currently supports and Python-specific challenges. Each misuse pattern incorporates different language features, allowing for the simplification of misuse patterns.

A. Misuse Patterns And Samples

We identified and created misuse patterns that encompass 18 different potential cryptographic API misuses. Each potential cryptographic API misuse type requires specific slicing and variable verification. We extrapolate these tasks to potentially misused modules. These misuse patterns are shown in Table I. We chose these misuse patterns since they represent different types of attack and attack surfaces. Identifying potential standard library misuses or taint analysis using methods such as "os.system" or "eval" is not cryptographic misuse and therefore is beyond the scope of this work. Researchers and developers can extend the current misuse patterns by including cryptographic patterns in their specifications.

1) Use a Wildcard to Avoid Verification: HyperText Transfer Protocol (HTTP) creates requests for applications to access web pages. HTTP itself has no security and can be continuously monitored by hackers [18]. HyperText Transfer Protocol Secure (HTTPS) use Secure Socket Layers (SSL) certificates to provide confidentiality to the website pages and are therefore recommended. Several modules allow developers to accept any certificate using a wildcard "*". The usage of a wildcard is synonymous with accepting any input.

2) Creating a Custom String to avoid Verification of Certificates: Developers use SSL to secure their connections and create HTTPS connections. This is similar to the misuse patterns II-A1 by preventing developers from removing certificate verification. By default, the Certificate Authority (CA) that is used to validate certificates is provided by the host machine. Developers can overwrite CA for certain modules [19]. This is highly discouraged, as this may be used to disable verification.

3) Use an unverified context to avoid HTTPS Verification: Like the previous misuse pattern, developers can avoid verification by changing the default verification behavior. By using the SSL library, developers may create an unverified context, therefore, disabling the SSL verification. This is discouraged, and developers are recommended to use the default SSL context.

4) Using HTTP instead of HTTPS: Developers can interact with APIs or web services by using HTTP or HTTPS requests. As explained by the Wildcard Misuse Pattern, HTTP is insecure and does not provide confidentiality. HTTPS provides confidentiality through SSL and is therefore recommended.

5) Using Insecure Random Number Generation: Developers must include cryptographically secure Pseudo Random Number Generator (PRNG) when using encryption methods to ensure integrity. If the developer uses an insecure key, an attacker may obtain the key and break into the encryption. Random numbers in such a security context must be cryptographically strong and secure. The Python Library provides a standard library "Random" that creates random values, but is cryptographically insecure [20]. As of Python 3.6, the standard library includes "Secrets" that is recommended for use, as it is cryptographically secure [21].

6) Using a static and insecure Salt: Certain cryptographic algorithms used for Password Based Encryption (PBE) require the use of a secure and random salt value. If the salt is

TABLE I

THE CRYPTOGRAPHIC POTENTIAL CRYPTOGRAPHIC API MISUSE, ATTACK TYPE, AND CRYPTOGRAPHIC PROPERTY PER POTENTIAL CRYPTOGRAPHIC API MISUSE. THE SEVERITY LEVELS ARE INDICIATED AS H/M/L FOR HIGH, MEDIUM, AND LOW RISK. THE CRYPTOGRAPHIC PROPERTIES C/I/A STAND FOR CONFIDENTIALITY, INTEGRITY, AND AUTHENTICITY RESPECTIVELY. THE ATTACK TYPES SSL/TLS MITM AND CPA STAND FOR SSL/TLS MAN-IN-THE-MIDDLE AND CIPHERTEXT ATTACK RESPECTIVELY. THE REQUIRED ANALYSIS METHODS PER RULE ARE LISTED WITH ↑ REFERRING TO FORWARD SLICING AND ↓ REFERRING TO BACKWARD SLICING. WE MARK THIS WITH AN ASTERISK TO EMPHASIZE THAT THE ANALYSIS REQUIRED IS SPECIFIC TO THE CRYPTOGRAPHIC RULE. EACH CRYPTOGRAPHIC RULE MAY REQUIRE ADDITIONAL ANALYSIS BASED ON THE LANGUAGE FEATURES. WE HAVE ALSO MINIFIED THE CODE SNIPPETS TO REPRESENT EACH POTENTIAL CRYPTOGRAPHIC API MISUSE TYPE. THE CODE SNIPPETS HAVE ABBREVIATED SECTIONS MARKED WITH AN ELLIPSE AND DO NOT INCLUDE IMPORTS DUE TO SIZE RESTRICTIONS. SEVERAL OF THESE RULES WERE INSPIRED BY CRYPTOLATION [11], AND WE HAVE MARKED THEM WITH A **.

#	potential cryptographic API misuse	Attack Type	Property	Severity	Slicing *	Snippet
1	Wildcard Verifiers Accept Hosts	SSL/TLS MitM	C/I/A	H	↑ & ↓	`argument.request(...,verify=False)`
2	String Trusting All Certificates		C/I/A	H**	↑	`os.environment["CURL_CA_BUNDLE"]=""`
3	Unverified HTTPS Context		C/I/A	H**	↑	`... = ssl._create_unverified_context`
4	Use of HTTP		C/I/A	H**	↑	`urllib.request.Request("http://...")`
5	Cryptographically Insecure PRNGs	Predictability	C	M**	None	`random.randint(0, 100)`
6	Static Salts	CPA	C	M**	↑ & ↓	`pbkdf2_hmac(...,b"NotLong")`
7	ECB Mode in Symmetric Ciphers		C	M**	↑	`Cipher(..., modes.ECB())`
8	Fewer than 1,000 Iterations for Salt	Brute Force	C	L**	↑ & ↓	`pbkdf2_hmac(..., 100)`
9	Insecure block ciphers		C	L**	↑	`DES.new(...,DES.MODE_OFB,...)`
10	Insecure asymmetric ciphers		C/A	L**	↑ & ↓	`dsa.*private_key(key_size=1_024)`
11	Insecure cryptographic hash		I	H**	↑	`MD5.new().update(...)`
12	Not Verifying a JSON Web Token	SSL/TLS MitM	I/A	H	↑ & ↓	`jwt.decode(...,verify=False)`
13	Using an insecure TLS Version		C	H	↑ & ↓	`ssl.wrap_socket(ssl.PROTOCOL_SSLv2)`
14	Using an Insecure Protocol		C/I/A	H	↑ & ↓	`ldap.initialize().simple_bind("","")`
15	Insecure XML Deserialization	Deserialization	C	M	None	`xml.dom.minidom.parse(...)`
16	Insecure YAML Deserialization			M	None	`yaml.dump_all(...)`
17	Insecure Pickle Deserialization			H	None	`pickle.loads(...)`
18	Securing Regex From ReDos	Brute Force	I	M	None	`re.search(r"", line, re.M` re.I)

not secure and has a random value, attackers may break the encryption through dictionary attacks. It is recommended to use a secure and random value.

7) Using an Insecure Mode: While developers may choose different algorithms for their encryption, they may choose the Electronic Codebook (ECB) mode. This mode is insecure as it breaks integrity and may leak information. It is recommended to use different modes such as Cipher Block Chaining (CBC).

8) Using less than 1,000 Iterations: Developers using PBE must include an iteration value. When developers use less than 1,000 iterations, their algorithm is cryptographically insecure. If developers use such a low iteration number, they increase the potential of a brute-force attack. Since the beginning of 2022, the official Python team [22] recommended that the value should be at least 1,000.

9) Using an Insecure Block Cipher: When developers use block ciphers, they use symmetric cryptography. Symmetric cryptography uses the same password for encryption and decryption. Using insecure block ciphers leaves the developer open to brute-force attacks. Ciphers such as Data Encryption Standard (DES) are considered insecure. It is recommended that developers use the secure symmetric cipher Advanced Encryption Standard (AES).

10) Using an Insecure Asymmetric Cipher: For developers using asymmetric ciphers, they need to use both public and private keys. The public key is used for encrypting certain contents while the private key is used for decrypting the contents, providing confidentiality and integrity. The asymmetric Rivest-Shamir-Adleman (RSA) cipher is considered secure when using a key size of 2,048 bits or greater [23]. It is recommended that developers use at least 2,048 bits.

11) Using an insecure Hash: Developers can use cryptographic hashes to verify the integrity of messages, files, or contents. These cryptographic hashes provide integrity by creating a specific value or message digest from a given input. A cryptographic hash is considered broken if two forms of content can create the same message digest, breaking the integrity. Broken cryptographic hashes include Secure Hash Algorithm (SHA)1, MD4, MD5, and MD2. Developers are recommended to use SHA256 and SHA512.

12) Not verifying the Json Web Token (JWT): Developers may use JWT to secure connections they make to remote servers JWT. Although JWT is more compact compared to other forms of authentication, they are verified by default. It is recommended that developers use the default behavior of the method and not override the signature verification. Cryptolation will detect when the default behavior is overridden to bypass verification.

13) Using a deprecated or invalid Transport Layer Security (TLS) Version: Python developers may specify the TLS or SSL version they use when connecting to different systems. Certain versions of TLS and SSL are deprecated and are considered insecure [24]. When developers use insecure versions, they risk potential Man in the Middle (MiTM) attacks. It is recommended that developers use the latest versions available.

14) Using an insecure protocol: When connecting to certain systems or computers, developers can use the Lightweight Directory Access Protocol (LDAP) protocol. Developers may connect using LDAP without providing credentials, which allows unauthenticated connections. To ensure a confidential connection, it is recommended to provide credentials.

15) Using an insecure Extensible Markup Language (XML) Deserialization: Developers who use XML objects should use secure methods to protect against Server-Side Request Forgery (SSRF) attacks. SSRF attacks will allow attackers to read the configuration information about the server. It is recommended for developers to disable network access and resolve remote entities.

16) Using an insecure YAML Ain't Markup Language (YAML) Deserialization: YAML modules may be exploited in a similar way to the XML Deserialization Misuse pattern. Unsafe usage of YAML grants attackers access to Remote Code Execution (RCE) by simple deserialization. Each YAML module has a safe deserialization method that is recommended.

17) Using an insecure Pickle Deserialization: Machine Learning developers commonly use the Pickle format to transfer information about their model. Despite high usage, the format is inherently insecure [25] since it supports RCE. There is no safe way to use the Pickle format, but it is recommended to sign the files for integrity.

18) Not properly escaping regular expressions (regex): Developers who use regex should escape the input they pass in to avoid Denial of Service (DoS). If they do not escape regular expressions, the input passed can cause the computer to calculate the expression continuously. It is recommended to escape the input to better sanitize the input passed in the regular expression engine.

B. Detection Challenges Introduced by Python

We chose several language features in addition to examining cryptographic API misuses specifically since these patterns have been used in other similar cryptographic misuse studies [26]. To our knowledge, there has been no research on Python language features used within cryptographic usage or its effects on SCA techniques. These language features should not be considered fully conclusive, but representative patterns that can directly create false positive (FP).

Global

```
1 from yaml import dump
2 x = dump
3 def starting_method():
4  global x
5  x(data, stream=None)
```

Listing 1. This is a sample program where the developer explicitly accesses a global variable. The global variable x stores the unsafe and insecure method dump and can be dynamically used. This itself is assigned Rule 16. Using the keyword global, the developer directly accesses this variable.

Developers may use variables at global scope in a program, making programming more simple, yet making SCA more difficult. Shown in Code Listing 1 a developer attempts to dump data from a certain YAML data. In line 2 the developer sets a global variable to an unsafe method and uses this method in line 5. Although SCA pattern matching tools can easily identify usage, they would not be able to track the propagation of global variables. The "global" declaration at line 4 allows developers to update the variable, and the SCA tools need to track the data-flow analysis.

Inter-Procedural

```
1 import pickle as pkl, os
2 class Klass(object):
3  def __init__(self, arg):
4   self.arg = arg
5  def __reduce__(self):
6   return self.arg.system,('echo "Hello"',)
7 pkl.loads(pkl.dumps(Klass(os)))
```

Listing 2. This code listing shows a custom and malicious Pickle class. Once the class is unmarshalled, it will automatically run the command "echo Hello World". This code execution is inherently within the Pickle structure and cannot be disabled.

SCA may need to slice through methods to trace variable propagation to identify cryptographic misuse. In the code listing 2 is a developer using the Pickle format to create a class and load that same class. This format is commonly used for Machine Learning (ML) applications but is naturally insecure. The Pickle format serialized the whole object, wrote it into bytes, and loaded it without any need for translation. The format allows developers to use hidden methods such as on line 5 to hook into and run custom code samples for deserialization. This overwriting ability creates an opportunity for RCE and cannot be patched or defended against.

The program flow starts at line 7 with the developer serializing and then deserializing the object at line 7. The PickleKlass sample is serialized without problems, but when the object is deserialized, the "magic method" __reduce__ at line 6 will automatically be executed. When the method is called, it currently simply executes an "echo Hello World" on the command line. SCA tools need to go through the methods to identify the program that was executed, and the vulnerable method called.

Double Inter-Procedural

```
1 from Crypto.Hash import MD5
2 def call_method():
3  def starting_method(argument):
4   print(argument().update(b'Hello').hexdigest())
5  return starting_method
6 call_method()(MD5.new)
```

Listing 3. Within this code sample is a method returning another method that creates the hash of a string. By taking the hash Algorithm as an argument, it creates a complex data flow graph that needs to be accounted for.

SCA needs to slice through multiple methods to identify potential cryptographic API misuses in more complicated and realistic programs successfully. This double-nested Inter-Procedural example is shown inside the program Code Listing 3. The first call of the method at line 2 returns the starting_method method from line 3. The "MD5.new" is passed into the second method and used dynamically at line 4. In contrast, pattern matching SCA may be able to identify the MD5 hash signature but not the usage of the code. This would potentially increase the FP if the MD5 were partially passed in as an input but not called.

Path-Sensitive

```
1 import os, sys, jwt
2 if True:
3  if str(input("Accept Path?")).lower() == "yes":
4   jwt.decode("",options={"verify_signature":False}
      )
5  else:
6   print("Didn't accept path")
```

Listing 4. This is a developer decoding a JWT through two conditional statements. The first conditional can be statically determined, while the second conditional depends on user input and cannot be statically determined. A SCA will need to handle the conditions in a path-insensitive manner to identify the potential cryptographic API misuse in Line 4.

SCA tools must decide how to deal with conditionals and paths to avoid path explosion. This happens when code analysis tools try to evaluate too many paths or conditions within the code flow. Shown in Code Listing 4 is the structure of the Path-Insensitive code files. There is a simple conditional on line 2 that always evaluates True, and an input-based conditional at line 3. Common SCA tools can easily bypass the first conditional, but the second depends on the user's input. At line 4 the developer decodes some JWT without verification, which is a cryptographic misuse and is not recommended.

Field-Insensitive

```
1 import ssl
2 class BaseRunner(object):
3  def __init__(self, argument):
4   self.argument = argument
5 obj = BaseRunner(ssl.PROTOCOL_SSLv2)
6 ssl.wrap_socket(ssl_version=obj.argument)
```

Listing 5. Within this code listing is a developer using a custom class to store variables and then uses it to specify the SSL version. This showcases a typical field-sensitive code flow, where the data is passed through the class and then through the method. The SCA needs to follow the data flow from object creation to object use. The Python library only provides an AST parser but does not provide a native data flow graph construction.

Utilizing data-flow graphs is required for SCA if they determine potential cryptographic API misuses based on the flow of the program. To showcase our field-insensitive examples is the Code Listing 5. SCA tools need to first identify the vulnerable SSL protocol used at line 5 and know the variable it is being assigned to. At the cryptographic potential cryptographic API misuse in line 6, the SCA needs to slice backward to identify the constant-propagation-based value of the variable. Pattern matching SCA may identify the signature of an invalid Secure Shell Protocol (SSH) version but not verify the usage. This approach may cause many FPs.

III. Design, Techniques, Analysis

In this section, we go over our design and implementation for Cryptolation. For our analysis, we cover Python projects in an intra-file inter-procedural approach.

A. Design

Our detection streamlines the analysis by only analyzing what is necessary. Different from other SCA tools, we identify the imports that we have misuse patterns for and only scan for those. We call this an *import-driven approach*. Shown in the Algorithm 1 is how Cryptolation examines each repository. The misuse patterns that are initially used include the default misuse patterns and any misuse patterns the developer passed in. We then retrieve the misuse patterns based on the imports from the file at line 10. This involves looking for the shortest matching signature between two strings, which includes more analysis but reduces the amount of false negative (FN). Using an import-driven approach increases performance and does not sacrifice accuracy or precision. Only after we slice through the import usage do we verify if there is a cryptographic misuse.

SCA tools may find and raise alerts if imports were included in the file but not used. This approach does not account for unused or dead code in the program. These potentially vulnerable imports are unused and do not pose an immediate threat. For each module, we find usages of the imports as shown in the Algorithm 4. We start by using forward slicing to identify the usage of the method signature. Then we will continue to slice backward if the misuse patterns depend on the value of a certain variable.

B. Techniques

Within this section, we describe the composition-based static analysis that Cryptolation uses. We use a combination of forward and backward static program slicing [27], and alias analysis [28]. Our analysis can be broken down into several steps about the specific misuse patterns and the module being examined. Each misuse pattern is composed of different modules and each module specification may require a different analysis. The different slicing techniques generally required per misuse patterns can be seen in Table I. Cryptolation uses def-use data-flow analysis [29] to identify variables assigned into the fields from class objects. Def-use data-flow analysis identifies variables assigned to a field from a class object instead of being instantiated from within the class. Instead of identifying the fields from within the class SCA tools need to track newly instantiated fields during execution. Our analysis includes tracking the variable value propagation through internal methods of the class.

Program slicing, as stated by Weiser [30] uses a Slicing Criteria $C = (\nu, p)$, where ν is a variable and p is a statement (e.g., a method call interacting with ν) within the program P to identify their influences or points of influence throughout the program. The slicing criteria are crucial to SCA as the criteria retrieve the set of program statements that may have influenced or been influenced by the variable ν or the statement ρ. Each slice we create through our SCA is executable, meaning each slice can be run separately and independently of the overall program P. For security program analysis, each statement ρ uses fully qualified module signatures. This operation enables the SCA tool to efficiently identify and slice through cryptographic code regions.

Our analysis comprises the following steps: i) Import Alias Analysis, ii) Forward Program Slicing, iii) Backward Program Slicing, iv) and Inter-Procedural Analysis. Cryptolation first passes the file and any user-defined misuse patterns files into

the import alias analysis resulting in the pertinent misuse patterns to be used for slicing criteria. These slicing criteria are then passed into the forward program slicing identifying and returning their usages and any dependent statements. Next, the slicing criteria are also passed into the backward program slicing to identify the statements that affect the specified variables. This helps determine the potential variable inferences. Finally, if there is any method during the program slicing, we use SCA slicing to extend our program slicing to become intra-procedural slicing. Each step of our analysis is misuse pattern dependent and is used as needed.

Cryptolation scans Python projects by scanning each Python file on its own and creating an individualized AST. We chose to create each AST using the popular library Astroid [31], which allowed us to focus on the depth of our analysis. Astroid statically interprets the Python code and determines the values for variables and methods as well. Shown in the Code Listing 7 within the Appendix is an example taken directly from Astroids website[2] showcasing their inference ability. Their value inference ability uses constant propagation to determine the values statically. Astroid takes program loops as a single iteration to avoid the well-known path explosion problem[Astroid While Loop, Astroid For Loop]. Due to this, Astroid supports path-sensitivity; which enables us to support it as well. Astroid leverages constant propagation and def-use chains to determine the potential variable values for backward slicing. We leverage and extend Astroids inference capability directly to support intra-procedural analysis.

Using the inference ability of Astroid allows us to go further with our data-flow analysis and increase the overall precision of Cryptolation. The inference ability we leverage allows us to statically predict what the value of the variable is without executing the code. We indicate whether the potential cryptographic misuse evaluation was made using inference and the variable inferences that led to the identification. We also include the preceding variables and the inferred values to detect potential cryptographic misuse. The inference ability can also handle conditional statements, making it path-insensitive and increasing our precision. This analysis lets us examine assignments or static assignments within the program and infer the values of the variables used within our cryptographic APIs. Astroid also provides the ability to manipulate the AST and run inferences upon it as normal[3]. An example of this is shown in the Code Listing 6 located in the appendix.

We create new copies of the leaf AST to handle the different language features Python provides (as described in Section II). These new AST copies extend our data flow graph by providing the inferred value of each variable. Since we create and modify a copy of the function $prepare\_url$, the original source code is not changed. We create copies of sub ASTs to ensure the changes will not persist within the original AST of the source code. Injecting the inference of a variable into an AST simply creates a new assignment node at the top of the AST before the rest of the method call. Shown in the Code Listing 6 is an example method that creates a URL when the prefix is provided. At line 5 the method is called with a "http://" prefix, resulting in an insecure URL at line 6. We bypass Intraprocedural limitations by taking the argument from line 5 and creating the new assignment node within the method at line 2. Utilizing the modified AST we can decrease our FN and increase our precision by analyzing the data-flow graph with the propagated assignment. Creating this modified AST allows us to slice through the base AST and continue slicing through the extended ASTs.

C. Analysis

1) Import Alias Analysis: Aliasing variables, statements, and imports require SCA tools to incorporate alias analysis to track. The AST module provided by the standard library will identify modules and their aliases in an intra-procedural manner. The variable propagation is created independent of control flow and must be handled by the SCA tool. We handle Import Aliasingto identify the imports stated by the project. Finally, we determine the imports that have misuse patterns associated with them due to our import-driven design.

2) Forward Program Slicing: Several of our misuse patterns require us to identify statements that are influenced by our slicing criteria. We identify these statements by using forward program slices, which retrieve the statements from the program P that is affected by the variable ν or the statement p. An example would be from misuse patterns 14 II-A14 in which the developer initializes an LDAP connection and then uses a simple bind. Since the simple bind is affected by the LDAP initialization, we identify these related instructions.

3) Backward Program Slicing: We leverage the inference capabilities of Astroid for our backward analysis. Their constant-propagation-based algorithm takes each conditional path as a separate potential variable since it's path-sensitive. For each loop, it uses a single iteration approach to avoid path explosion, where there may be potentially infinite code flows. their algorithm is not just limited to variable values but also aliased imports and aliased methods. In this manner, we determine a potential variable value despite the dynamic features of python. An important feature of Astroid's algorithm is limiting the number of inferences made per variable, which directly feeds our intra-procedural analysis; which we let the developers specify. If developers do not specify we use the order of variable inferences provided by Astroid.

4) Intra-Procedural Analysis: As shown in Algorithm 3, this process happens for every argument passed in and every function call. For each set of arguments and their inferred values, we determine the full set of combinations needed to represent each potential AST. While accommodating the language features examined earlier, this algorithm increases the overhead for the worst-case scenario[4]. Creating each AST poses a problem similar to the path explosion problem, and

[2]The website can be found at the following URL (URL)

[3]Astroid provides the ability to modify the in-memory AST. This does not overwrite or alter the source code in the file.

[4]The worst-case scenario being an extremely complex Python file as ranked by Mccabe Cyclomatic Complexity (CC) (MCC) and the language features used.

Algorithm 1 The program workflow of Cryptolation.

Input FilePath ▷ Path to directory or file.
Input Pattern File ▷ User defined patterns .
Output Results

$filesLen \leftarrow$ *length*$(FilePath)$
for $fileNum \leftarrow 1$ *to* $filesLen$ **do** ▷ Loop each file.
  $file : PythonFile \leftarrow files[fileNum]$

  ▷ Get the imports from our rules and the file
  $fileImports : imports \leftarrow$ *getImports*$(files)$
  $ruleImports \leftarrow$ *getImports*$(Base \bigcup UserFile)$

  ▷ Find the union of imports.
  $rule \leftarrow fileImports \bigcup ruleImports$

  $ruleLen \leftarrow$ *length*$(rules)$ ▷ Each rule.
  for $ruleNum \leftarrow 1$ *to* $ruleLen$ **do**
    $rule \leftarrow patterns[ruleNum]$

    ▷ Determine slices that contain the rule imports.
    $slice \leftarrow$ *AnalyzeProgram*$(file, rule)$

    ▷ Determines if the slice breaks the rule.
    $broken \leftarrow$ *verifyPattern*$(rule, slice)$

    if $broken$ **then** ▷ Append the slice to the results.
      $results \leftarrow results + slice$
    end if
  end for
end for

we refer to it as a depth explosion problem. To mitigate the potential risks of the depth-explosion problem, we let the user limit the number of ASTs created. From our real-world evaluation, we tracked timeouts to indicate if there were any problems. While Cryptolation had four timeouts that may indicate a depth explosion, it is at the average number of timeouts per project through the overall evaluation.

The growing number of ASTs may also introduce inaccuracy. The inaccuracy is caused by subsequent ASTs that are created from variable inferences. If a potential cryptographic API misuse is discovered through our Inter-Procedural Analysis, we include the specific program slices leading to the potential cryptographic API misuse . This analysis mitigates the increasing inaccuracy by requiring the developer to manually review the program slices.

IV. PyCryptoBench Benchmark

In this section, we go over the benchmarks we evaluate the tools on. We also cover the methods of creating our own curated benchmark PyCryptoBench, and the breakdown of the different test cases. For the real-world test cases, we explain our methodology and rationale for the projects we chose to include in our evaluation.

We created a Python benchmark called PyCryptoBench to determine the effectiveness of our tool against five cryptographic API misuses. Our benchmark consists of 1,836 test file. Each test file may include one language feature, one potential cryptographic API misuse, and one Test Case Type. There are 1,530 test files that use the following language features: Unenforced Variable Type Definitions, Import Aliasing, Complex Data Type Usage, and Named Variable Placement. Each language feature category consists of 1,530 test files, with 306 test files that do not have language feature . Then, our benchmark consists of different cryptographic API misuses, language features , and finally into three different Test Case Types: Benign test cases, Vulnerable test cases, and then Non-Usage test cases. We limited the program patterns of inter-procedural test cases to single and double inter-procedural test cases since more enhanced test cases would yield the same results. We break the benchmark down first into two major groups, the group using a module and the group without a module. This is similar and was inspired by [26] where the benchmark was broken down by the complexity of the test cases.

A. Vulnerable Test Case Type

We created Vulnerable with one cryptographic API misuse each with one Module import. If the test case included a language feature, the cryptographic API misuse would be incorporated into the language feature. Each file has a minimum number of Lines of Code (LoC) to reduce the time it takes each tool to scan it. There was an overall maximum of 10 LoC for all the test cases.

B. Benign Test Case Type

Each test case Benign file only contains a language feature usage and does not contain any cryptographic API misuse. These test cases focus simply on whether the tools can parse each language feature without creating false alerts. Overall, the tools did not have problems scanning these files, as they did not have cryptographic API misuse and no Module import. SCA Tools that solely searched for the source code "Module import" should be able to correctly identify these files as Benign.

C. Non-Usage Test Case Type

The Non-Usage test cases were created with one Module imported but not used. We used this to indicate whether the tools could identify the difference between dead or unused code and a potential potential cryptographic API misuse. These files also include a minimum number of LoC to reduce the scan time of each file.

V. Evaluation

Within this section, we cover the evaluation. First, we will start with the methodology that we use to determine the different benchmarks and how we reduce the bias of the results. Following this, we will cover the benchmark at the top right and the unranked projects in that order.

A. Methodology

We chose Bandit, Semgrep, and Dlint since they are Python framework-agnostic and utilizes SCA. We used the following tools and their associated versions, Bandit 1.7.0, Semgrep 0.75.0, and Dlint 0.12.0. We used a single Jupyter Notebook that operates the installation, scanning, and aggregation for every project being scanned using each tool. Each program automatically maps its respective misuse pattern values from its respective tools to the equivalent Cryptolation misuse patterns value.

Our evaluation addresses the following concerns.

- Do the tools handle language features in the benchmark?
- How do tools compare when scanning the real-world project?

B. Benchmark Results

TABLE II
THIS IS THE BREAKDOWN OF TRUE POSITIVE (TP),FN, PRECISION, RECALL, AND ACCURACY SCORE PER TOOL ON THE PYCRYPTOBENCH. PYCRYPTOBENCH CONSISTS OF 1,836 TEST CASES; TEST CASES WITH ONE POTENTIAL CRYPTOGRAPHIC API MISUSE AND TEST CASES WITHOUT ANY POTENTIAL CRYPTOGRAPHIC API MISUSES. WE ONLY CREATED VULNERABLE TEST CASES TO FOCUS ON THE PRECISION OF EACH TOOL AND TO PENALIZE TOOLS FOR INCORRECT ALERTS.

Tool	TP	FP	TN	FN	Pre.	Recall.	Acc.
Bandit	54	92	1,548	142	37%	28%	87%
Cryptolation	120	0	1,609	107	**100%**	53%	**94%**
Dlint	80	420	1,224	112	16%	42%	71%
Semgrep	122	384	1,224	106	24%	**54%**	73%

Cryptolation has outperformed the other tools in the benchmarks with an overall Precision of 100% and 0 FP as shown in Table II. The SCA tool Semgrep was the closest in performance to the other tools. We intentionally created the benchmark to use different Python language features that are difficult for SCA to scan. While Semgrep had the highest number of TP alerts, it also had the second-highest number of FP alerts. Most of the tools consistently had 100% of true negative (TN) alerts. While Semgrep had the highest FN alerts, this may be alleviated by consistently updating the engine[5]. Between the time of evaluation and writing, Semgrep has increased several versions.

We created PyCryptoBench independent of Cryptolation, with the intent of stress-testing SCA tools. Each advanced test case used one language feature to create a difficult-to-scan but easy-to-read Python script. We modeled the complexity of the test cases from the language features we listed, from a previous well-known Java SCA benchmark [26], and weaknesses discussed in SCA tools [11], [6], [32], [33], [12]. Our Cryptolation algorithm, which was created to handle more difficult-to-scan Python scripts, has the highest Accuracy at 94%. Bandit, which has the second-highest Accuracy, achieved 87%. Our algorithm shows significant growth toward handling complex Python source code. Every issue we discover with Cryptolation is used to progress research into adapting to these difficulties. Specifically, with Cryptolation, we discovered Field-Sensitive and Interprocedural language features are the two lowest categories. These two categories had an overall average Recall of 32.43% and 10.53% respectively. Our algorithm focuses on variable assignment through function depth. However, the breadth of the variable assignments and improper refinement before using the library Astroid is why Cryptolation suffers in these two language features. To improve Recall with these language features, we will update the algorithm to make refinements and remove unnecessary information for those conditions.

We also broke down the evaluation by language feature as shown in Table III. Cryptolation succeeds in having the highest precision and accuracy scores throughout, corresponding to the overall results. Despite Cryptolation having the highest overall recall, Semgrep had the highest recall for the Global language feature and Dlint had the highest recall for the Double-Interprocedural categories. Cryptolation has a lower recall in these categories due to its focus on precision. The Global and Double-Interprocedural test categories were the most complex language features to identify for Cryptolation. The precision per category was consistently 100% for Cryptolation.

C. Top-Ranked GitHub Projects

We chose four Top-Ranked Python projects specified by their maturity and percent of Python files. We had a ratio of 90% Python files measured by GitHub and chose four Top-Ranked project. These projects are; Ansible, Django, IntelOwl, and Scrapy. We manually reviewed all of these alerts due to the low number of projects.

Due to the maturity of these projects, we did not encounter as many alerts as compared with the Un-Ranked Projects. This is shown in Table IV. Dlint has the highest precision for the Top-Ranked Project, with a 5% difference with Cryptolation. This is due to the regex Cryptolation uses for pseudo-random number generator identification. The pseudo-random number generator has been an important and contested topic of conversation with code analysis tools and one of which we will adequately address. Despite this Cryptolation has a 14% Precision difference from Bandit. From scanning the projects, Bandit raised eight FP due to the identified potential cryptographic API misuses not being used. Their results revealed unused and not real potential cryptographic API misuses. Cryptolation, Dlint, and Semgrep all raised TP alerts, with Cryptolation and Dlint raising more TPs.

D. Un-Ranked GitHub Projects

We ranked GitHub projects by the most popular Python projects with the tag “payments” or “cryptography”. We chose those two tags since they are more likely to use cryptographic imports. After crawling through the two different tags, we retrieved 939 projects. Scanning the 939 projects resulted in more than 20,000 alerts. Due to many alerts, we manually reviewed a sample size of 50 alerts per scanned tool. We reviewed these results by checking the sample size, automatically retrieving the code snippets identified by line numbers,

[5]We used version 0.75.0 of the rules, that we have archived.

TABLE III

IN THIS TABLE IS THE BREAKDOWN OF THE TP, FP, FN, PRECISION, RECALL, AND ACCURACY OF EACH TOOL SCANNING EACH LANGUAGE FEATURE. THESE EVALUATIONS ONLY INCLUDE THE MANUALLY REVIEWED RESULTS AND THE NON-BIASED TEST FILES FROM THE BENCHMARK. CRYPTOLATION HAS THE HIGHEST PRECISION AND ACCURACY PER CATEGORY. WHILE CRYPTOLATION HAS THE HIGHEST RECALL FOR MOST CATEGORIES, DLINT HAS THE HIGHEST FOR THE FIELD-SENSITIVE AND DOUBLE-INTERPROCEDURAL CATEGORIES.

Language Feature	Tool	TP	FP	TN	FN	Precision	Recall	Accuracy
Double-Interprocedural	Bandit	12	16	258	20	42.86%	37.50%	88.24%
	Cryptolation	31	0	268	7	**100.00%**	**81.58%**	**97.71%**
	Dlint	16	70	204	16	18.60%	50.00%	71.90%
	Semgrep	23	64	204	15	26.44%	60.53%	74.18%
Field-Sensitive	Bandit	7	17	258	24	29.17%	22.58%	86.60%
	Cryptolation	12	0	269	25	**100.00%**	32.43%	**91.83%**
	Dlint	14	70	204	18	16.67%	43.75%	71.24%
	Semgrep	20	64	204	18	23.81%	**52.63%**	73.20%
Global	Bandit	5	16	258	27	23.81%	15.62%	85.95%
	Cryptolation	24	0	268	14	**100.00%**	**63.16%**	**95.42%**
	Dlint	9	70	204	23	11.39%	28.12%	69.61%
	Semgrep	17	64	204	21	20.99%	44.74%	72.22%
Interprocedural	Bandit	2	17	258	29	10.53%	6.45%	84.97%
	Cryptolation	4	0	268	34	**100.00%**	10.53%	**88.89%**
	Dlint	10	70	204	22	12.50%	31.25%	69.93%
	Semgrep	16	64	204	22	20.00%	**42.11%**	71.90%
Path-Sensitive	Bandit	15	13	258	20	53.57%	42.86%	89.22%
	Cryptolation	24	0	268	14	**100.00%**	**63.16%**	**95.42%**
	Dlint	16	70	204	16	18.60%	50.00%	71.90%
	Semgrep	24	64	204	14	27.27%	**63.16%**	74.51%

TABLE IV

THIS IS THE BREAKDOWN OF TP,FP, AND PRECISION PER TOOL FOR THE TOP-RANKED PROJECT. THIS IS THE TOTAL NUMBER OF ALERTS FROM THE TOP-RANKED PROJECT.

Toolname	TP	FP	Precision
Bandit	108	15	87.80%
Cryptolation	295	40	**88.06%**
Dlint	331	50	86.88%
Semgrep	130	40	76.47%

and programmatically identifying the specific libraries using Mitosheet. These were then manually verified by two of the authors.

TABLE V

HERE ARE THE TP, FP, AND PRECISION OF 50 RANDOMLY SAMPLED ALERTS FROM THE UN-RANKED GITHUB PROJECT SET.

Toolname	TP	FP	Precision
Bandit	44	6	88%
Cryptolation	50	0	100%
Dlint	50	0	100%
Semgrep	50	0	100%

Due to the number of projects, we did not manually review the alerts. We randomly sampled 50 alerts per tool within the Un-Ranked data set and manually verified this sample. These results can be found in Table V with Cryptolation having 100% precision along with Dlint and Semgrep. After setting these values, we discovered the breakdown of results was similar to that of the Top-Ranked project V-C. Cryptolation has a relatively large number of alerts for cryptographic API misuse rule 11 and 5, while Dlint has the highest number of alerts for cryptographic API misuse rule 5. Most of the alerts were from rule 11, as shown in the appendix chart 4. It should be noted the majority of the alerts were focused on cryptographic API misuse rules from 5 to 15. This could be due to the specificity of rules or the relative usage of each rule. We included a one day timeout for the tools to finish scanning their projects, which all but Cryptolation failed a few projects.

These relative spikes of alerts indicate a high possibility of FN by the other tools or FP by the individual tool. For future work, we will manually review a larger sample of this data set to identify the FN alerts. This included grouping the alerts by broken misuse patterns and checking for keywords.

VI. DISCUSSION AND LIMITATIONS

Within this section, we will go over the discussion and limitations of Cryptolation. This includes the scope of the dynamic language features we chose, the precision and recall that Cryptolation provides, and proper reporting of these alerts.

Include more language features We included five different language features throughout our advanced test cases, but more language features need to be examined. There has recently been a similar project called PyScan [34] which identifies a large number of language features used within Python. The seed language features we targeted gave us many great insights about certain project domains using Python. However, these language features are not exhaustive.

Improving Precision and Recall We discussed and decided against creating a super control flow graph that incorporates all of the files in the project. While this approach would improve the recall and accuracy by identifying potential cryptographic API misuses across files[6], we did not want to decrease performance. We will also correct the pseudo-random number generator and ensure Cryptolation correctly identifies their

[6]This attack scenario is when Python files are treated as libraries or namespaces and are imported as such. Thus the functionality and data flow of a method may cross over different files.

usage. Another way recall can be improved is to enhance how we resolve the fully qualified APIs. Currently, we first attempt to infer the class name of the object however, if that fails, we fall back to using the import name as the prefix.

Further Analysis and Reporting We did not create any GitHub issues for potential cryptographic misuses due to the need for further analysis. There were more than 100,000 alerts within the Un-Ranked Projects benchmark. We do not want to create a huge number of FP GitHub issues. This would be irresponsible and create distrust within the community.

We will continue to review these results and make issues when all of the alerts have been thoroughly reviewed. This work provides a benchmark to scan a large number of Un-Ranked projects, however, the ground truth of these projects are unknown. This imposes a limitation on calculating the recall of the alerts, as it relies on the knowledge of the ground truth. We explored 943 projects. Currently, confidently determining the ground truth of such a large number of projects can take years of effort and is out of the scope of this paper. Future work to automate and facilitate the determination of the ground truth for such Un-Ranked projects would have a valuable impact in the evaluation and development of SCA tools.

VII. Related Work

Many of Python's available static code analysis tools either do not support advanced language-specific features or use simpler analysis algorithms. An exceptional and useful program is CryptoGuard[7]. CryptoGuard uses static code analysis to scan Java and Android projects for potential cryptographic misuses [11]. Similar to Cryptolation, CryptoGuard reduces false positives to achieve precise results.

Code analysis tools that scan Python programs focus on specific libraries or modules. For example, PyXhon is a Dynamic Code Analysis (DCA) program that identifies potential Dynamically Linked Libraries (DLL) issues utilizing Function Oriented Analysis [35]. Pyxhon uses the code execution to identify potential DLL misuse, while Cryptolation statically identifies potential cryptographic module issues.

Both Pythia [36] and DjangoChecker [37] focus on cryptographic misuses within the popular Python web framework Django. The tools both aim to identify Cross Site Scripting (XSS) flaws within the Django code. DjangoChecker utilizes taint tracking and a custom browser to sanitize the input and identify malicious payloads. Pythia uses taint tracking and a Universal Abstract Syntax Tree (UAST) between the Django templates and the Python code to identify malicious payloads and the attack space. Python code analysis tools may also analyze the Python bytecode. Chen et al. [38] created a modified version of CPython to write out more detailed information[8]. The enriched Python bytecode is statically and dynamically analyzed to identify the def-use data and control dependencies. While this creates more information to enhance the code analysis, this depends on the manipulation of the interpreter.

Another similar approach can be related to the work done by Xu et al. [33], where the authors utilize a different kind of preprocessing to handle Python. They created an enhanced Python Control Flow Graph that includes much of the meta information. This meta information creates a tie between the variable and whatever type the value is. Their work is not designed to run intra-procedural analysis to identify ties between different methods. Another way to create developer-friendly security tools is to ensure the future and extensibility of your tools. While they didn't fully go over their analysis and methods, they provided many different test cases. They also enable developers to use different analyses such as data flow control flow lexical semantics.

VIII. Future Work

Further Source Code Evaluation In the future, we will include a bigger breadth of Python topics. The topics we used were likely associated with Cryptographic module usage. There are likely other topics that may indicate a strong need for Cryptographic module usages, such as "system", "repository", "revision", "Internet of Things (IoT)". We will also look into the top-rated Python projects by GitHub, PyPiStats, and other ranking sites.

Closed Source Comparison We deliberately chose only to compare Cryptolation to other open-source programs. Open-source programs allow us to look closely at their internal algorithms and allowed us to script the scanning and their test results. In the future, we will compare Cryptolation against other tools, such as Synopsys and Snyk. Both tools are highly rated and well known throughout the field and have been recognized several times by Gartner Reports [39].

Performance Our approach may not potentially scale well with increasingly more difficult Python programs. With unbounded limitations, our algorithm will use an increasingly larger amount of memory and time. We could improve the algorithm by allowing the developer to prioritize different cryptographic rules.

Refinement Our algorithm currently supports enhanced analysis by continuously using more memory. During our analysis, we depend on the Astroid library for identifying the variable inferences based on the context or variable state of the method and the variables. One way we could decrease the amount of FPs Cryptolation incurs from the Top-Ranked Project is to filter out irrelevant, or un-influential variables; similar to the work done by Parfait [40]. This type of reduction shown by this refinement reduced the amount of FPs by an overall total of 50%. Our reduction would not directly affect our analysis, but indirectly by removing the number of variables Astroid will need to take into account. This not only could improve our FP but the amount of memory we use by lowering the number of variables that are unimportant for our use.

General Framework Our algorithm in Cryptolation currently targets potential cryptographic misuses. However, the constant-

[7]https://github.com/CryptoGuardOSS/cryptoguard

[8]This is done in a concept similar to adding debugging flags to a compiler.

propagation-powered intra-procedural analysis could be extended to other uses, such as policies, code regulations, code smells, and taint analysis. We may generalize the algorithm to allow more tailored solutions that require our procedures.

IX. Conclusion

We created Cryptolation to examine and discover potential cryptographic misuse for source code using complex language features. Our benchmark PyCryptoBench provides 1,836 files that cover complex language features, FP test cases, and various Cryptographic modules. We only reported the results of all of the tools for comparison after removing files that certain tools would not cover. We also compared all of the tools against 1,017 different projects and manually reviewed a random sampling of the results. Cryptolation provides improved precision on complex modules compared to the state-of-the-art tools when scanning the real-world source code and our benchmark.

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)*, 2016, pp. 289–305.

[2] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 372–383. [Online]. Available: https://doi.org/10.1145/3180155.3180201

[3] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How Reliable is the Crowdsourced Knowledge of Security Implementation?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 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 *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 109–126. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

[5] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, Y. Brun, and N. C. Ebner, "API Blindspots: Why Experienced Developers Write Vulnerable Code," in *Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security*, ser. SOUPS '18. USA: USENIX Association, 2018, p. 315–328.

[6] Y. Brun, T. Lin, J. E. Somerville, E. M. Myers, and N. C. Ebner, "Blindspots in Python and Java APIs Result in Vulnerable Code," *ACM Trans. Softw. Eng. Methodol.*, nov 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3571850

[7] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, ""Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow"," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 886–896. [Online]. Available: https://doi.org/10.1145/3180155.3180260

[8] 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"*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–84. [Online]. Available: https://doi.org/10.1145/2508859.2516693

[9] 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*, 2015, pp. 519–534.

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

[11] 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*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2455–2472. [Online]. Available: https://doi.org/10.1145/3319535.3345659

[12] H. Gulabovska and Z. Porkoláb, "Survey on Static Analysis Tools of Python Programs." in *SQAMIA*, 2019.

[13] "PEP 484 – Type Hints," Jan 2022, [Online; accessed 4. Jan. 2022]. [Online]. Available: https://www.python.org/dev/peps/pep-0484

[14] "JetBrains: Developer Tools for Professionals and Teams," Jan 2022, [Online; accessed 4. Jan. 2022]. [Online]. Available: https://www.jetbrains.com/lp/python-developers-survey-2020

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

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

[17] 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)*, 2017, pp. 154–171.

[18] C. A. Barton, G. A. Clarke, and S. Crowe, "Transferring data via a secure network connection," Aug. 15 2006, uS Patent 7,093,121.

[19] "Advanced Usage — Requests 2.26.0 documentation," Sep 2021, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python-requests.org/en/master/user/advanced

[20] "random — Generate pseudo-random numbers — Python 3.10.2 documentation," Feb 2022, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python.org/3/library/random.html

[21] "secrets — Generate secure random numbers for managing secrets — Python 3.10.2 documentation," Feb 2022, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python.org/3/library/secrets.html

[22] "hashlib — Secure hashes and message digests — Python 3.10.2 documentation," Feb 2022, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python.org/3/library/hashlib.html

[23] E. Barker, L. Chen, A. Roginsky, A. Vassilev, R. Davis, and S. Simon, "Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography," National Institute of Standards and Technology, Tech. Rep., 2018.

[24] "ssl — TLS/SSL wrapper for socket objects — Python 3.10.2 documentation," Feb 2022, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python.org/3/library/ssl.html

[25] "pickle — Python object serialization — Python 3.10.2 documentation," Feb 2022, [Online; accessed 1. Feb. 2022]. [Online]. Available: https://docs.python.org/3/library/pickle.html

[26] S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller, and D. Yao, "Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 485–497, 2023.

[27] R. P. Lippmann and R. K. Cunningham, "Improving intrusion detection performance using keyword selection and neural networks," *Computer Networks*, vol. 34, no. 4, pp. 597–603, 2000, recent Advances in Intrusion Detection Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128600001407

[28] S. Debray, R. Muth, and M. Weippert, "Alias Analysis of Executable Code," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 12–24. [Online]. Available: https://doi.org/10.1145/268946.268948

[29] H. Y. Yang, E. Tempero, and H. Melton, "An Empirical Study into Use of Dependency Injection in Java," in *19th Australian Conference on Software Engineering (aswec 2008)*, 2008, pp. 239–247.

[30] M. D. "WEISER, "PROGRAM SLICES: FORMAL, PSYCHOLOGICAL, AND PRACTICAL INVESTIGATIONS OF AN AUTOMATIC PROGRAM ABSTRACTION METHOD," Ph.D. dissertation, 1979, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual

underlying works; Last updated - 2023-02-23. [Online]. Available: http://login.ezproxy.lib.vt.edu/login?url=https://www.proquest.com/dissertations-theses/program-slices-formal-psychological-practical/docview/302949655/se-2

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

[32] D. Nikolić, D. Stefanović, D. Dakić, S. Sladojević, and S. Ristić, "Analysis of the Tools for Static Code Analysis," in *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2021, pp. 1–6.

[33] Z. Xu, J. Qian, L. Chen, Z. Chen, and B. Xu, "Static Slicing for Python First-Class Objects," in *2013 13th International Conference on Quality Software*, 2013, pp. 117–124.

[34] Y. Peng, Y. Zhang, and M. Hu, "An Empirical Study for Common Language Features Used in Python Projects," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 24–35.

[35] M. Sun, D. Gu, J. Li, and B. Li, "PyXhon: Dynamic detection of security vulnerabilities in Python extensions," in *2012 IEEE International Conference on Information Science and Technology*, 2012, pp. 461–466.

[36] L. Giannopoulos, E. Degkleri, P. Tsanakas, and D. Mitropoulos, "Pythia: Identifying Dangerous Data-Flows in Django-Based Applications," in *Proceedings of the 12th European Workshop on Systems Security*, ser. EuroSec '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3301417.3312497

[37] A. Steinhauser and P. Tůma, "DjangoChecker: Applying extended taint tracking and server side parsing for detection of context-sensitive XSS flaws," *Software: Practice and Experience*, vol. 49, no. 1, pp. 130–148, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2649

[38] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. C. Chu, and B. Xu, "Dynamic Slicing of Python Programs," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, 2014, pp. 219–228.

[39] "Gartner 2022 Magic Quadrant for Application Security Testing | Micro Focus," Sep. 2022, [Online; accessed 6. Sep. 2022]. [Online]. Available: https://www.microfocus.com/en-us/assets/cyberres/magic-quadrant-for-application-security-testing

[40] Y. Xiao, Y. Zhao, N. Allen, N. Keynes, D. D. Yao, and C. Cifuentes, "Industrial Experience of Finding Cryptographic Vulnerabilities in Large-Scale Codebases," *Digital Threats*, vol. 4, no. 1, mar 2023. [Online]. Available: https://doi.org/10.1145/3507682

APPENDIX A
FORMULAS USED

We use the following formula to measure the Precision of our results.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

We used the following formula to measure the Recall for our results.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

We used the following formula to measure the Accuracy of our results.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3}$$

APPENDIX B
RULE MAPPING

Within this section, we go over the specification we used to map different rule numbers between the tools. Each SCA tool had its rule specification. We provide this to ensure the results are reproducible.

A. Bandit misuse patterns mapping

Listed right below in table VI is the translation table we used to convert Bandit alerts to Cryptolation alerts. We aligned these rules based on the rule message and the modules each tool identifies. This should be adjusted whenever each tool is updated.

TABLE VI
SHOWN HERE IS THE MAPPING OF RULES FROM BANDIT TO CRYPTOLATION. WE ONLY MAPPED THE RELEVANT CRYPTOGRAPHIC RULES.

Bandit ID	Cryptolation ID
B303	11
B304	10
B305	10
B309	4
B310	4
B311	5
B312	14
B321	14
B323	3
B324	11
B401	14
B402	14
B412	4
B413	11
B501	3
B502	13
B503	1
B504	13
B505	1
B507	3

B. Dlint misuse patterns mapping

Listed right below in table VII is the translation table we used to convert Dlint alerts to Cryptolation alerts. We aligned these rules based on the rule message and the modules each tool identifies. This should be adjusted whenever each tool is updated.

TABLE VII
SHOWN HERE IS THE MAPPING OF RULES FROM DLINT TO CRYPTOLATION. WE ONLY MAPPED THE RELEVANT CRYPTOGRAPHIC RULES.

Dlint ID	Cryptolation ID
DUO102	5
DUO103	17
DUO107	15
DUO109	16
DUO122	1
DUO123	1
DUO130	8
DUO131	14
DUO132	14
DUO133	9
DUO134	9
DUO138	18
DUO120	17
DUO135	15

C. Semgrep misuse patterns mapping

Listed below in table VIII is the translation table we used to convert Semgrep alerts to Cryptolation alerts. We aligned these rules based on the rule message and the modules each tool identifies. This should be adjusted whenever each tool is updated.

TABLE VIII
SHOWN HERE IS THE MAPPING OF RULES FROM SEMGREP TO CRYPTOLATION. WE ONLY MAPPED THE RELEVANT CRYPTOGRAPHIC RULES.

Semgrep ID	Cryptolation ID
A1	20
A2	4
A3	13
A4	15
A5	21
A6	2
B303	11
B304	9
B305	7

APPENDIX C
INFERENCE CODE DESIGNS

Within this section, we provide more detailed algorithms of our design for Cryptolation. These extremely high-level algorithms simply describe how Cryptolation operates from beginning to end.

Algorithm 2 showcases the program flow of Cryptolation at the highest level. First, Cryptolation gets the user Python file and all the modules rules to search for. The modules rules include our pre-defined rules and any rules defined by the user. Next, Cryptolation iterates through each import and iterates over each usage of the modules that is found. The usage will be added to the broken_rules if the usage is not verified, and the broken_rules will be returned at the end of the function.

Algorithm 3 showcases how Cryptolation creates a function filled in with variable inferences. First, Cryptolation creates a copy of the method and iterates through each variable passed into the function. Next, we obtain the variable instance at the call site and gather the inferences through the use of

Algorithm 2 Cryptolation's High Level Procedure

Input $pyfile$ ▷ The variable under investigation.
Input $uport$ ▷ The name of the function call.
Output mtd ▷ The method with assignments.

procedure ANALYZE(pyfile, uport)
  $broken\_rules \leftarrow []$
  $len\_imp \leftarrow length(imports(pyfile, uport))$
  for $port \leftarrow 1$ to len_imp **do**
    $len\_forward \leftarrow len(forward(port, pyfile))$
    for $usage \leftarrow 1$ to $len\_forward$ **do**
      if $!verify\_usage(usage)$ **then**
        $broken\_rules \leftarrow broken\_rules$ + $usage$
      end if
    end for
  end for
return $broken\_rules$ ▷ Returns the Broken Rules.
end procedure

Algorithm 3 Cryptolation's inter-procedural analysis.

Input ν_n ▷ The variable under investigation.
Input p ▷ The name or AST of the function call.
Output mtd ▷ A copy of the method with additional assignments.

procedure EXPAND(p, ν_n)
  $mtd : method \leftarrow$ *getMethodCall*(p)
  $astCopy : method \leftarrow$ *createCopy*(mtd)
  ▷ Duplication.

  $lenVar \leftarrow$ *length*(ν_n)
  for $varNum \leftarrow 1$ *to* $lenVar$ **do** ▷ Loop through var.
    $var \leftarrow \nu_n[varNum]$
    $vName : Name \leftarrow getArgName(p, var)$

    $varCall \leftarrow$ *getVarAtCall*(var, p) ▷ Get the var.
    $varCallInf \leftarrow$ *getInferences*$(varCall)$ ▷ infers.

    $varCallInfLen \leftarrow$ *length*$(varCallInf)$
    for $infNum \leftarrow 1$ *to* $varCallInfLen$ **do** ▷ infer.
    ▷ variable inference.
      $varInf \leftarrow varCallInf[infNum]$

    ▷ Insert assignment statement using infer.
      $result \leftarrow inject(astCopy, p, vName, varInf)$

    end for
  end for
return mtd ▷ Returns the function with the assignments for arguments.
end procedure

the Astroid library. For each variable inference, we create an assignment statement within the function assigning the inference value to the variable at the top of the function. This mimics the behavior of the variable passed in within the current value. Finally, Cryptolation will return the copy of the method with the inferences inside the copy of the function. Using this modular basis, we can continuously create inference-supporting functions for each internal function. Due to this, we allow the user to specify the depth or the depth of the functions we will expand.

Algorithm 4 The program analysis of Cryptolation.

Input patterns ▷ The Misuse Patterns.
Input file ▷ The program file.
Output Results

procedure $iterative\_expand(patterns,\ file)$
  $pattern \leftarrow patterns[patternNum]$
  $slicing \leftarrow$ *forward*$(file)$

  if pattern depends on variable **then**
    if InferVal depends on method call **then**
    ▷ Create ASTs with variable inferences injected.
      $subAsts \leftarrow$ *expand*$(astTree, inferVal)$

      $astsLen \leftarrow$ *length*$(subAsts)$
      for $astNum \leftarrow 1$ *to* $subAsts$ **do** ▷ Loop ASTs.
        $curAst \leftarrow subAsts[astNum]$

      ▷ Determine the value inferences.
        $slicing \leftarrow slicing\ +$ *backward*$(curAst)$
      end for
    else ▷ inferVal does not depend on the method call.
      $inferred \leftarrow$ *infer*$(variable)$ ▷ Get inferences.

      $inferLen \leftarrow$ *length*$(inferred)$
      for $inferNum \leftarrow 1$ *to* $inferLen$ **do** ▷ Loop infers.
        $inferVal \leftarrow inferred[inferNum]$

      ▷ Append Results.
        $slicing \leftarrow slicing +$ *backward*$(inferVal)$
      end for
    end if
  else ▷ Pattern does not depend on the variable.
    $slicing \leftarrow$ *backward*$(inferVal)$ ▷ Append Results.
  end if
return $slicing$ ▷ Returns the specified function slice with a number of copies based on the inferences.
end procedure

Algorithm 4 showcases how Cryptolation can continuously infer variable inferences, limited only by depth. First, Cryptolation will obtain the pattern and the forward slices based on the current state. If the current pattern or potential cryptographic misuse does not depend on any variable, the function will append the current slices. Otherwise, the function will gather all of the ASTs supporting the number of inferences we are allowed to look into if the inference value depends on the method call. For each AST, the function will append the inferred backward slicing state into the current slicing capabilities. If the current pattern does not depend on the method call, then the function will immediately add on the backward slice that covers the specific inference value. Finally, the backward slicing within each step is provided by the library Astroid, and we then slice through the results. We nickname this process of creating copies of the AST that expand upon the variable inferences "context injection".

```
1 >>> def prepare_url(prefix=None):
2     #prefix = "http://" >Injected into memory
3     internal_url = "url.com"
4     return prefix + internal_url #> http:// + url.com
5 >>> overhead = top_call("http://")
6 #> overhead = http://url.com
```

Listing 6. This is an example of how we use context injection to enhance the AST. We extend our inference ability to create an instantiation of the potential variable value within the AST. This allows us to follow the code-flow in an inter-procedural manner. This is not a known code sample but is representative of several scanned code samples.

Shown in C is a full example of how the context injection process that fills the variable values to precisely identify potential cryptographic API misuse. The function *prepare_url* simply takes a string prefix and returns the prefix combined with the string "url.com". Normally, SCA tools may not be able to determine the return result of the function call at line 5. Cryptolation uses the context injection process to insert the prefix "http://" from the method call at line 5, into the function at line 2. Since we create and modify a copy of the function *prepare_url*, the original source code is not changed. Finally, Cryptolation determines that the variable *overhead* will likely return the result "http://url.com" at line 6.

A. Extracting nodes via Astroid

```
1 >>> import astroid
2 >>> name_node = astroid.extract_node('''
3 a = 1
4 b = 2
5 c = a + b
6 c
7 ''')
8 >>> name_node
9 <Name.c l.5 at 0x7fc0fb6a1790>
10 >>> inferred = next(name_node.infer())
11 >>> inferred.value
12 3
```

Listing 7. An example inference taken directly from Astroid's website. This example shows how the astroid AST can generate variable inferences from the specified node.

Shown in figure C-A is an example of how Astroid can generate variable inferences for specific variables. Astroid internally uses constant propagation to determine the potential dependent values of a certain variable and attempt to determine the certain result of the logic. Specifically, at line 5, the variable c depends on the variables a and b, and will attempt to determine the combination of their values. Simply looking at it, we know the result would be three if the script were run. Using the Astroid library, line 10 returns a list of variable inferences for the specified value. The first and only inference value available at line 11 returns a successful inference value of 3. It is important to note variable inferencing may not be successful, and Cryptolation will not look further into that function or variable.

Appendix D
Benchmark Insights

Shown within this section is the breakdown of files measured from GitHub. In the breakdowns, we cover the sum and average count of metrics for each category.

TABLE IX
WITHIN THIS TABLE, WE SHOW THE STATISTICS ABOUT ALL THE PROJECTS WE ANALYZED FROM GITHUB. WE GROUPED THESE METRICS BY COUNT, SUM, AND AVERAGE. THE COUNT SHOWS THE NUMBER OF PROJECTS FOR TOP-RANKED AND UN-RANKED, THE SUM SHOWS THE SUMMATION OF THE TYPES OF METRICS, AND THE AVERAGE SHOWS THE AVERAGE OF EACH TYPE OF METRIC. THE TYPE OF METRICS INCLUDES THE COUNT OF FILES, PROGRAM OR TOTAL NUMBER OF LINES, AND THE CYCLOMATIC COMPLEXITY.

Stat.	Type of Metric	Un-Ranked	Top-Ranked
Count	# of Projects	936	4
Sum	# of Files	62,0 60	4,757
	# of Program Lines	6,995,932	366,325
	# of Total Lines	13,397,498	701,0 18
	# of CC	317,715	19,850
Average	# of Files Per Project	66.3	1,189.25
	# of Program Lines Per File	112.73	77.01
	# of Total Lines Per File	215.88	147.37
	# of CC Per File	5.12	4.17

Shown within the table IX, we summarize the Un-Ranked and Top-Ranked Projects that were used within our evaluation. We showcase the total and average Program Lines, Total Lines, and Cyclomatic Complexity per project or file. The Cyclomatic Complexity is included as a commonly used objective method to approximate the difficulty of a source code file. We did not explore the impact of the CC on each tool; however, future work should continue this examination.

TABLE X
SHOWN WITHIN THIS TABLE IS THE NUMBER OF PROGRAM LOC, TOTAL LOC, AND NUMBER OF FILES PER TOP-RANKED PROJECT. THE TOP-RANKED PROJECTS ARE RANKED IN ASCENDING ORDER OF EACH VALUE.

Top-Ranked Project	Program LoC	Total LoC	Files
Django	210,167	383,558	2,724
Ansible	116,605	252,180	1,509
Scrapy	29,409	50,983	329
Intelowl	10,144	14,297	195

The number of files, program LoC, and total LoC for each top-ranked project is shown in the table X. We calculate the program LoC with a simple function by removing any comments or empty lines from our count. It should be noted that many of the Python files may be empty; since an empty "__init__.py" is used to denote a Python directory. The file counts only include Python files with the extension ".py", any potential compiled Python files.

APPENDIX E
ADDITIONAL METRICS

Within this section, we include complimentary metrics and charts, that could be beneficial for clarifying specific points for the reader, but we were unable to include in the main body of the paper.

We have included several venn diagrams that display the shared alerts the tools raised while scanning the Un-Ranked Projects.To identify the shared alerts, we used the unique fully qualified path name of the python file that included each alert as their identifier. Figure 1 shows all the alerts per tool and their linkages to other tools, indicating the number of alerts each tool has in common with the others. It is important to note, as mentioned previously, since we do not have access to the ground truth, these alerts were not reviewed for correctness.

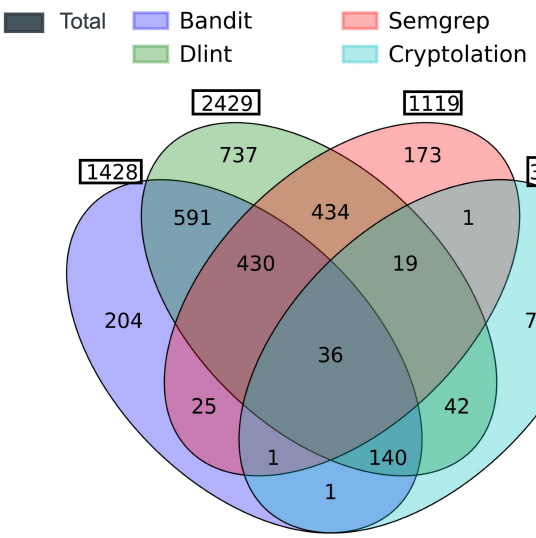

Fig. 1. Venn diagram indicating the number of shared and the number of exclusive files of Un-Ranked Project alerts.

We further broke down the sampled Un-Ranked alertsreviewed in Figure 1 using the fully qualified path names. The venn diagram of this new break-down, shown in Figure 2, consists of 50 alerts per tool that were randomly sampled for review, adding up to 200 alerts in total. The total values per tool do not add up to 200 in this diagram, since the same file can include multiple alerts.

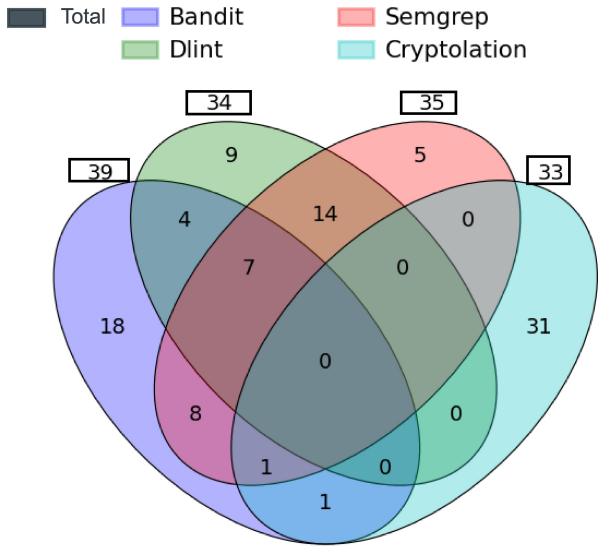

Fig. 2. Venn diagram indicating the number of shared and the number of exclusive files of our sample of peer-reviewed Un-Ranked alerts.

Figure 3 shows our sample of peer-reviewed Un-Ranked alerts from Figure 2, using the project names as the unique identifier for alerts.

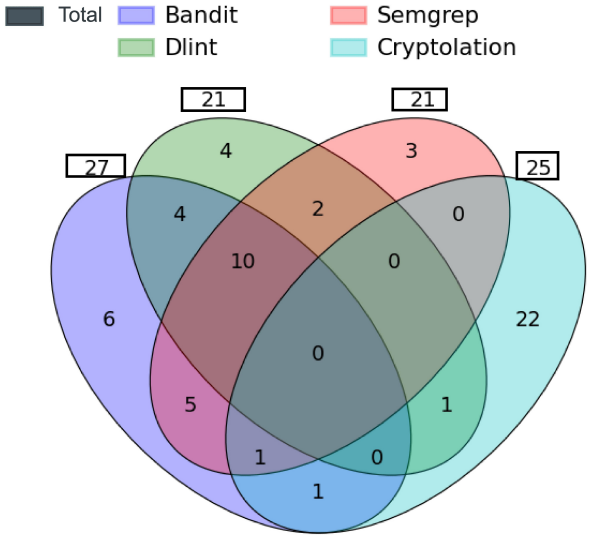

Fig. 3. Venn diagram indicating the number of shared and the number of exclusive projects of our sample of peer-reviewed Un-Ranked alerts.

TABLE XI
SHOWN WITHIN THIS TABLE IS THE PERCENTAGE OF ALERTS RAISED BY EACH TOOL PER RULE NUMBER.

Rule Number	Bandit	Cryptolation	Dlint	Semgrep
1	0.0%	0.13%	2.65%	0.16%
2	0.01%	0.0%	0.01%	0.08%
3	0.1%	0.0%	0.0%	0.0%
4	2.04%	0.28%	0.0%	0.01%
5	7.99%	3.3%	11.24%	0.0%
6	0.0%	0.0%	0.0%	0.0%
7	0.22%	0.07%	0.24%	0.22%
8	0.0%	0.0%	7.31%	0.0%
9	0.15%	0.8%	11.01%	0.37%
10	0.17%	0.5%	0.0%	0.0%
11	14.31%	1.89%	0.05%	6.86%
12	0.0%	0.0%	0.0%	0.0%
13	0.77%	0.0%	0.02%	2.46%
14	0.02%	0.0%	0.05%	0.0%
15	0.0%	5.91%	5.88%	6.53%
16	0.0%	0.0%	0.32%	0.0%
17	0.0%	0.11%	2.53%	0.0%
18	0.0%	0.02%	3.24%	0.0%

Within the table XI we show the rounded percents of alerts per tool broken down by rule over the entire number of Un-Ranked alerts. Many tools have 0 or less then 1% of alerts. Bandit has the overall highest percent of alerts with Rule 11 at 14%. Please note, the majority of these alerts are not reviewed for correctness. This chart only portrays the quantity of alerts per tool.

TABLE XII
SHOWN WITHIN THIS TABLE ARE THE METRICS OF THE TOOLS ALERTS PER THE BENCHMARK CATEGORY TYPES VULNERABLE AND NON-USAGE. THE HIGHEST RANKING PRECISION, RECALL, AND ACCURACY PER TOOL WITHIN EACH CATEGORY ARE INDICATED IN BOLD.

Type	Tool	TP	FP	FN	Pre.	Rec.	Acc.
Vuln.	Bandit	54	32	142	62.79%	27.55%	23.68%
	Crypto.	120	0	107	**100%**	52.86%	**53.07%**
	Dlint	80	36	112	68.97%	41.67%	35.09%
	Sem.	122	0	106	**100%**	**53.51%**	53.51%
Non-Usage	Bandit	0	60	0	0.00%	0.00%	92.54%
	Crypto.	0	0	0	0.00%	0.00%	**100%**
	Dlint	0	384	0	0.00%	0.00%	52.24%
	Sem.	0	384	0	0.00%	0.00%	52.24%

Within the table XII, we showcase the tools Precision, Recall, and Accuracy per general test category type. The Precision and Recall for the Benchmark types Benign and Non-Usage are zero due to there being no True Positive alerts. We did not list the benign test cases since all of the tools had 100% Accuracy recorded. While Cryptolation has the highest overall Precision and Accuracy overall, Semgrep has the highest recall for Vulnerable test cases by less then 1%.

TABLE XIII
SHOWN WITHIN THIS TABLE IS A FULL BREAKDOWN OF THE GROUPINGS WITHIN PYCRYPTOBENCH. WE BREAK OUR TESTS DOWN BY GENERAL CATEGORY, LANGUAGE FEATURE, AND RULE NUMBER. EACH GROUPING OF TEST TYPES WILL ADD UP TO 1,836.

Test Grouping	Sub Grouping	Number of tests
Category	Non-Usage	804
	Safe	804
	Regular	228
Language Feature	Global	306
	Dbl. Inter-Procedural	306
	Path-Sensitive	306
	Inter-Procedural	306
	Field-Sensitive	306
Rule Number	Rule 01	132
	Rule 02	48
	Rule 03	36
	Rule 04	576
	Rule 05	24
	Rule 06	60
	Rule 07	36
	Rule 08	24
	Rule 09	252
	Rule 10	84
	Rule 11	144
	Rule 12	72
	Rule 13	96
	Rule 14	84
	Rule 15	84
	Rule 16	36
	Rule 17	24
	Rule 18	24

We breakdown the benchmark in all three categories in table XIII. While each specific category adds up to 1,836, the categories do not get added to each other for the total. Each category will use the same set of 1,836 files and overlap with one another.

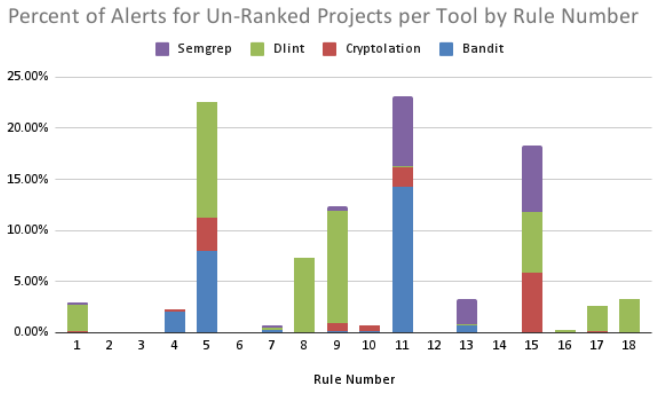

Fig. 4. Shown within this picture is the Percent of alerts generated by tool that were mapped to our rules. Rule 11, five, and 15 have the highest percentage of alerts in ranking. Dlint appears to have a wide breadth of alerts, with the most percentage across all of the rules. It should be noted we did not check the majority of these alerts for validity.

TABLE XIV
SHOWN WITHIN THIS TABLE IS THE PRECISION, RECALL, AND ACCURACY PER TOOL SCANNING PYCRYPTOBENCH BROKEN DOWN BY RULE NUMBER.

Rule	Tool	Precision	Recall	Accuracy
1	Bandit	100.00%	4.17%	82.58%
	Cryptolation	100.00%	33.33%	87.88%
	Dlint	28.00%	29.17%	73.48%
	Semgrep	37.93%	45.83%	76.52%
2	Bandit	0.00%	0.00%	75.00%
	Cryptolation	100.00%	25.00%	81.25%
	Dlint	0.00%	0.00%	75.00%
	Semgrep	100.00%	83.33%	95.83%
3	Bandit	63.64%	87.50%	86.11%
	Cryptolation	100.00%	75.00%	91.67%
	Dlint	100.00%	91.67%	97.22%
	Semgrep	100.00%	91.67%	97.22%
4	Bandit	80.00%	80.00%	99.31%
	Cryptolation	100.00%	66.67%	99.31%
	Dlint	0.00%	0.00%	90.62%
	Semgrep	20.75%	91.67%	92.53%
5	Bandit	100.00%	66.67%	83.33%
	Cryptolation	100.00%	83.33%	91.67%
	Dlint	100.00%	83.33%	91.67%
	Semgrep	0.00%	0.00%	50.00%
6	Bandit	0.00%	0.00%	80.00%
	Cryptolation	100.00%	63.64%	93.33%
	Dlint	0.00%	0.00%	60.00%
	Semgrep	0.00%	0.00%	60.00%
7	Bandit	100.00%	66.67%	88.89%
	Cryptolation	100.00%	75.00%	91.67%
	Dlint	45.45%	83.33%	61.11%
	Semgrep	45.45%	83.33%	61.11%
8	Bandit	0.00%	0.00%	50.00%
	Cryptolation	100.00%	33.33%	66.67%
	Dlint	0.00%	0.00%	50.00%
	Semgrep	0.00%	0.00%	50.00%
9	Bandit	0.00%	0.00%	80.95%
	Cryptolation	100.00%	100.00%	100.00%
	Dlint	0.00%	0.00%	47.62%
	Semgrep	9.09%	100.00%	52.38%
10	Bandit	100.00%	50.00%	92.86%
	Cryptolation	100.00%	66.67%	95.24%
	Dlint	0.00%	0.00%	42.86%
	Semgrep	14.29%	50.00%	50.00%
11	Bandit	8.33%	100.00%	84.72%
	Cryptolation	100.00%	41.67%	95.14%
	Dlint	0.00%	0.00%	58.33%
	Semgrep	9.43%	41.67%	61.81%
12	Bandit	0.00%	0.00%	83.33%
	Cryptolation	100.00%	25.00%	87.50%
	Dlint	0.00%	0.00%	83.33%
	Semgrep	100.00%	83.33%	97.22%
13	Bandit	66.67%	100.00%	95.83%
	Cryptolation	100.00%	33.33%	91.67%
	Dlint	21.05%	66.67%	64.58%
	Semgrep	28.57%	100.00%	68.75%
14	Bandit	0.00%	0.00%	71.43%
	Cryptolation	0.00%	0.00%	85.71%
	Dlint	0.00%	0.00%	50.00%
	Semgrep	0.00%	0.00%	50.00%
15	Bandit	0.00%	0.00%	85.71%
	Cryptolation	100.00%	66.67%	95.24%
	Dlint	0.00%	0.00%	50.00%
	Semgrep	28.57%	100.00%	64.29%
16	Bandit	0.00%	0.00%	66.67%
	Cryptolation	0.00%	0.00%	66.67%
	Dlint	66.67%	100.00%	83.33%
	Semgrep	0.00%	0.00%	50.00%
17	Bandit	0.00%	0.00%	50.00%
	Cryptolation	100.00%	100.00%	100.00%
	Dlint	100.00%	100.00%	100.00%
	Semgrep	100.00%	100.00%	100.00%
18	Bandit	0.00%	0.00%	50.00%
	Cryptolation	0.00%	0.00%	50.00%
	Dlint	0.00%	0.00%	50.00%
	Semgrep	0.00%	0.00%	50.00%

Within the table XIV, we break down the Precision, Recall, and Accuracy of each tool on each rule on the benchmark. Throughout each rule, Cryptolation either has the highest Precision or is tied with the highest Precision.