A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI

Jukka Ruohonen
University of Turku, Finland
Email: juanruo@utu.fi

Kalle Hjerppe
University of Turku, Finland
Email: kphjer@utu.fi

Kalle Rindell
University of Turku, Finland
Email: kakrind@utu.fi

2021 18th International Conference on Privacy, Security and Trust (PST) | 978-1-6654-0184-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/PST52912.2021.9647791

***Abstract*—Different security issues are a common problem for open source packages archived to and delivered through software ecosystems. These often manifest themselves as software weaknesses that may lead to concrete software vulnerabilities. This paper examines various security issues in Python packages with static analysis. The dataset is based on a snapshot of all packages stored to the Python Package Index (PyPI). In total, over 197 thousand packages and over 749 thousand security issues are covered. Even under the constraints imposed by static analysis, (a) the results indicate prevalence of security issues; at least one issue is present for about 46% of the Python packages. In terms of the issue types, (b) exception handling and different code injections have been the most common issues. The `subprocess` module stands out in this regard. Reflecting the generally small size of the packages, (c) software size metrics do not predict well the amount of issues revealed through static analysis. With these results and the accompanying discussion, the paper contributes to the field of large-scale empirical studies for better understanding security problems in software ecosystems.**

***Index Terms*—Bug, defect, issue, smell, vulnerability, weakness, repository, ecosystem, static analysis, linting, Bandit, PyPI**

I. Introduction

Python is currently among the most popular programming languages. Like many of today's poplar programming languages, Python also provides its own ecosystem for archiving and maintaining open source software packages written with the language. This ecosystem is known as the Python Package Index. Recently, this index and related language-specific repositories have been extensively studied from different angles. Maintenance, release engineering, and dependencies have provided typical motivations for the studies. Security has provided a further motivation. All motivations intersect with methodological questions in empirical software engineering.

This intersection also provides the reference point for the present work; the paper continues the recent large-scale empirical analyses on software security issues in software ecosystems [22], [25], [80]. As is elaborated in the opening Section II, the paper also belongs to a specific corner in this research branch: the packages within PyPI are analyzed in isolation of each other; the ecosystem concept is understood in statistical terms as a population rather than in technical terms as a collection of more or less interlinked packages. As is further elaborated in the section, the paper can be further framed with respect to the method of detecting (but not verifying) different security "issues", which may, or may not, equate to actual security bugs. Often, the term "smell" is used as an alternative.

In other words, the paper is closely tied to static analysis, which has long been used as an alternative to other means to discover security issues during software developing, including security-related code reviews (see [58] for a comprehensive review of the history and associated literature). However, neither static analysis nor code reviews are sufficient alone; both tend to miss many security issues [13]. In addition to discussing this point in more detail, the opening section notes the framing toward the Python programming language itself. Afterwards, the structure is fairly straightforward: the large-scale empirical approach is outlined Section III together with the statistical methods used; results are presented in Section IV and further discussed in V in conjunction with their limitations; and a brief conclusion follows in the final Section VI.

II. Research Design

A. Research Questions

The following three research questions (RQs) are examined:

- RQ_1: *How common and severe are security issues in PyPI packages given constraints of static program analysis?*
- RQ_2: *What types of security issues are typical to PyPI packages given again the constraints of static analysis?*
- RQ_3: *What is the average number of security-related issues when the size of the packages is controlled for?*

These questions are easy to briefly justify. Given the ecosystem-wide focus soon described, RQ_1 is worth asking because the empirical dataset covers the whole PyPI. Bug severity is also a classical topic in software security research [56], [60]. It is further related to the types of security issues detected (RQ_2), which are always interesting and relevant for making practical improvements. Although all three questions are constrained by the static analysis tool used, these constraints are balanced by the large-scale analysis. That is, generalizability is sought toward all packages stored to PyPI but not toward all static analysis tools—let alone all security issues. This framing of the paper's scope is a typical "pick one" choice often encountered in practical software engineering: not all generalizability types can be achieved in a single study under reasonable time and resource constraints.

The final RQ_3 can be motivated with existing literature on the relation between software vulnerabilities and software metrics. A basic finding in this literature is that different software metrics (such as those related to quality and design) correlate with the amount of vulnerabilities—yet these metrics correlate also with the size of software [2], [10]. These correlations make interpretation less straightforward. Although numerical indicators on software quality correlate with the amount of vulnerabilities discovered (and, to some extent, can even predict these), it may be that the underlying theoretical dimension is merely software size, or that software size is a confounding factor [15]. Therefore, it has been argued that when predicting bugs, vulnerabilities, weaknesses, smells, or other software quality attributes, software size metrics are mainly useful as covariates (a.k.a. control variables) [21]. To this end, concepts such as vulnerability density have also been introduced to gain more robust empirical measurements.

Although scaling by software size (such as the lines of code) has been common (see [55] and references therein), scaling can be done also in terms of testing coverage and continuous integration traces [17]. Static analysis coverage is used in the present work; RQ_3 is answered by using the number of lines and files of the Python code scanned by the static analysis tool used. The gist behind RQ_3 is simple: if the results indicate that the average number of issues is similar with and without conditioning by these software size metrics, there is a little rationale to use vulnerability densities or related scaling. This reasoning depends on the context; Python and the PyPI software ecosystem, static analysis, and security issues.

B. Related Work

Instead of enumerating individual papers, the paper's focus is better elaborated by considering four large branches of related research. These branches allow to also further frame the paper's scope. The branches can be summarized as follows.

1) Software Ecosystems: Software ecosystems cover a large research branch in software engineering, information systems research, and related fields. According to a famous definition, a "software ecosystem is a collection of software projects which are developed and evolve together in the same environment" [26, p. 265]. On the one hand, this definition underlines the presence of a common environment, which is typically orchestrated by a company or a community. Thus, the definition includes commercial software ecosystems (such as Google Play) and ecosystems that focus on providing supplementary functionality to a software framework, such as the WordPress plugin ecosystem [55]. That said, software package repositories (such as *npm* for JavaScript, CPAN for Perl, or CRAN for R) are likely the most studied software ecosystems in empirical research. Also PyPI has been studied recently [22], [54], [76]. In light of the definition, the Python Package Index indeed provides a common environment, although the actual development mainly occurs on GitHub, another software ecosystem. On the other hand, the definition underlines common software evolution, which, in turn, pinpoints toward dependencies between packages and longitudinal analysis. For framing the present work, it is important to underline that neither apply: the empirical analysis is based on a snapshot and the packages are analyzed in isolation from each other. Finally, it should be remarked that only publicly available open source packages are considered; PyPI is the theoretical population.

2) Static Analysis: Static analysis covers a huge research branch in software engineering and computer science in general. Typical application domains relate to portability, coding style enforcement, reliability, and maintainability [42]. According to this high-level framing, security-oriented static analysis is a subset of the reliability domain. Within this subset, static analysis is typically used to either explicitly detect software vulnerabilities, or to provide warnings to developers about unsafe programming practices that may lead to different security issues, including software weaknesses that may manifest themselves as concrete software vulnerabilities. The warning-related use case applies to the present work; none of the issues in the empirical analysis have been explicitly verified to have security implications, although some implicit confidence exists that some of the issues truly are security issues. In terms of static analysis research, further framing can be done with a distinction to technical and social research topics. In general, the former deal with tools for specific programming languages, while the latter studies the question of how these tools are used with particular languages [4], [38], [77]. The social topics include also the further question of how developers use static analysis tools to diagnose and fix weaknesses and vulnerabilities [44], [59]. Although the paper does not present a new tool, the topic is still technical rather than social in the sense that nothing is said about the development and developers of the Python packages.

3) Security Issues: The third related research branch covers the detection of different security issues during software testing. The branch is again large—to say the least. Even when limiting the scope to static analysis, numerous relevant questions are present: which issues are detected; how tools differ for a given programming language; how detection differs between languages; how fault-injection compares to signature-based approaches and how these differ from mutation analysis and search-based analysis; and so on [58], [1]. These technical questions can be again augmented by social ones; how detection based on static analysis fits into software testing in general; how effective is such detection in terms of time and resource constraints; and so forth. In terms of security issues themselves, typically either known vulnerabilities, usually based on the Common Vulnerabilities and Exposures (CVEs), or the abstract weaknesses behind these, often based on the corresponding Common Weakness Enumeration (CWE) framework, are used on both sides of the socio-technical research paradigm [44], [5]. Roughly, CVEs are more useful on the technical side and CWEs on the social side of research. For instance, CWEs have been used to better understand typical programming mistakes in Python packages [54], to provide dynamic information sources for software developers using static analysis tools [59], [27], and so forth. Despite the advantages of such approaches also for systematic empirical

analysis, the paper's approach is based on the issue categories provided by the static analysis tool used. Although the issues caught by the tool can be mapped to CWEs [52], such mapping yields less fine-grained categories and must be done manually.

4) Python: The Python programming language itself constitutes the fourth and final branch of related research. This branch intersects with all previous three branches. As already remarked, PyPI has been studied in previous work, but there are also many Python-specific studies on security issues and static analysis. Regarding the latter, it is necessary to point out the obvious: Python is a dynamically typed language, which, on one hand, makes static analysis tools less straightforward to implement. On the other, the typing system also makes static analysis tools particularly valuable for Python and related languages [4]. Against this backdrop, it is no surprise that many static analysis tools have been developed for Python in both academia and industry. These range from simple linter-like checkers (such as the one used in the present work) to formal verification methods and actual static type systems [16]. In addition, many related datasets and empirical studies have been conducted, including studies focusing on traditional software metrics (such as those related to code complexity and object-orientation) [43], code smells [79], call graphs [23], and the Python's type system [78]. Within this empirical research domain, there exists also one previous study that shares the same motivation and uses the same static analysis tool: [52]. As will be elaborated in the subsequent section, the datasets, their operationalization, and methods are still very different.

III. Materials and Methods

A. Data

The dataset is based on a simple index file provided in the Python Package Index [74]. In total, $224,651$ packages were listed in the index at the time of retrieving it. Given these packages, the most recent releases were downloaded from PyPI with the `pip` package manager using the command line arguments `download --no-deps`. Because of the dynamic index file, (a) it should be remarked that some packages were no longer available for download. Furthermore, (b) only those packages were included that were delivered as well-known archive files (namely, as *tar*, *gzip*, *bzip2*, *xz*, ZIP, or RAR files). After extracting the archives, the contents were fed to the Bandit [45] static analysis tool using Python 3.6.10. This feeding is illustrated in Fig. 1. Given the ongoing painful transition from Python 2 to Python 3 [28], (c) those packages had to be excluded that were compatible only with Python 2.7. Finally, (d) those packages were also excluded for which Bandit reported having not scanned a single line or a file.

Fig. 1: Sample Construction in a Nutshell

Despite the necessary exclusions discussed, the dataset constructed contains the static analysis results for as many as $n = 197,726$ packages. Thus, the size of the dataset is very similar to other recent large-scale studies (for instance, $192,666$ Python packages were retrieved from PyPI in [22]). A brief remark is also in order about the tool providing the results in the present work. This static analysis tool was originally developed by Hewlett-Packard and associates for use in the OpenStack project. It has been used also in previous research [52]. Given existing taxonomies for static analysis tools [9], [19], Bandit exploits the conventional abstract syntax tree but operates mainly at the local level; statements, functions, and parameters to functions are parsed without considering the flow and semantics between these. The tool's approach can be further elaborated by considering the issues.

B. Issues

Bandit contains several individual detectors that can be enabled or disabled according to a project's specific needs. Given the PyPI-wide focus, all of these were included in the sample construction except a detector for the use of assertions (B101), which was prone to false positives according to preliminary tests. The individual issue types detected are enumerated in Table I. The three first columns in the table display identification codes for the detectors, their mnemonic names, and brief descriptions for the issues detected. The fourth and fifth columns contain Bandit's metrics for the *severity* (S) of the issues detected and the *confidence* (C) of detection for a given issue. Both are important in practice. Static analysis tools are highly prone to false negatives (bugs not caught) and false positives (issues that are not bugs) [9], [13]. Including detection confidence indicators is one way for addressing the problem [38]. Analogously, according to recent surveys, issue severity is the most important factor for software developers in their prioritization tasks [77]. The severity and confidence metrics take three values: *low* (L), *medium* (M), and *high* (H). Although there is a long-standing debate about ranking the severity of security-related issues, including actual vulnerabilities in particular [60], [61], the values provided in Bandit are taken for granted in order to ensure replicability. Finally, the table's last column provides references to further information about the issues. When available, these references again conform with those given in the Bandit's source code.

The issues can be grouped into seven categories. These are:

1) The first category contains detectors for *generic* issues that are well-known to be risky. Particularly noteworthy are the heuristic detectors for the presence passwords hard-coded to the source code. Although Bandit's detection confidence is not high for these issues, hard-coded credentials have been behind many recent high-profile data breaches in cloud computing environments [6].
2) The second category includes a single detector for running a particular web application in *debug mode.*
3) The third category addresses various *function calls* that may lead to security issues with varying degree of severity. The examples range from the Python's "pickling" functionality via insecure cryptography to well-known but unsafe functions in the language's standard library.

TABLE I: Issue Types Detected by Bandit (excluding assertions)

Code	Mnemonic name	Description	S	C	Ref.
B102	exec_used	Use of the exec function	M	H	[48]
B103	set_bad_file_permissions	Insecure permissions for files	M, H	M, H	[72]
B104	hardcoded_bind_all_interfaces	Binding a socket to all network interfaces	M	M	[40]
B105	hardcoded_password_string	Use of hard-coded passwords in non-function contexts	L	L	[66]
B106	hardcoded_password_funcarg	Use of hard-coded passwords in function arguments	L	M	[66]
B107	hardcoded_password_default	Use of hard-coded passwords in default function arguments	L	M	[66]
B108	hardcoded_tmp_directory	Use of hard-coded temporary directories	M	M	[68]
B110	try_except_pass	Using pass as a catch-all-style exception handling	L	H	[36]
B112	try_except_continue	Using continue as a catch-all-style exception handling	L	H	[36]
B201	flask_debug_true	Running a Flask web application in debug mode	H	H	[53]
B301	pickle	Use of insecure deserialization	M	H	[31]
B302	marshal	Use of insecure deserialization	M	H	[31]
B303	md5	Use of MD2, MD4, MD5, or SHA1 hash functions	M	H	[30]
B304	ciphers	Use of insecure ciphers such as DES	H	H	[30]
B305	cipher_modes	Use of insecure cipher modes	M	H	[30]
B306	mktemp_q	Use of the insecure mktemp function	M	H	[51]
B307	eval	Use of the possibly insecure eval function	M	H	[37]
B308	mark_safe	Use of the possibly insecure mark_safe function	M	H	[11]
B309	httpsconnection	Use of the insecure HTTPSConnection with some Python versions	M	H	[67]
B310	urllib_urlopen	Use of a file scheme in urlopen with some Python versions	M	H	[41]
B311	random	Use of pseudo-random generators for cryptography/security tasks	L	H	[35]
B312	telnetlib	Use of the insecure Telnet protocol	H	H	[32]
B313	xml_bad_cElementTree	Use of possibly insecure Extensible Markup Language (XML) parsing	M	H	[18]
B314	xml_bad_ElementTree	Use of possibly insecure XML parsing	M	H	[18]
B315	xml_bad_expatreader	Use of possibly insecure XML parsing	M	H	[18]
B316	xml_bad_expatbuilder	Use of possibly insecure XML parsing	M	H	[18]
B317	xml_bad_sax	Use of possibly insecure XML parsing	M	H	[18]
B318	xml_bad_minidom	Use of possibly insecure XML parsing	M	H	[18]
B319	xml_bad_pulldom	Use of possibly insecure XML parsing	M	H	[18]
B320	xml_bad_etree	Use of possibly insecure XML parsing	M	H	[18]
B321	ftplib	Use of the clear-text sign-in File Transfer Protocol (FTP)	H	H	[32]
B322	input	Use of the insecure input function (with Python 2)	H	H	[37]
B323	unverified_context	Explicitly bypassing default certificate validation	M	H	[47]
B324	hashlib_new_insecure_functions	Use of MD2, MD4, MD5, or SHA1 hash functions with hashlib	M	H	[30]
B325	tempnam	Use of the insecure and deprecated tempnam or tmpnam functions	M	H	[49]
B401	import_telnetlib	Import of a Telnet library	H	H	[32]
B402	import_ftplib	Import of a FTP library	H	H	[32]
B403	import_pickle	Import of a library for deserialization	L	H	[31]
B404	import_subprocess	Import of the possibly insecure subprocess library	L	H	[50]
B405	import_xml_etree	Import of a possibly insecure XML parsing library	L	H	[18]
B406	import_xml_sax	Import of a possibly insecure XML parsing library	L	H	[18]
B407	import_xml_expat	Import of a possibly insecure XML parsing library	L	H	[18]
B408	import_xml_minidom	Import of a possibly insecure XML parsing library	L	H	[18]
B409	import_xml_pulldom	Import of a possibly insecure XML parsing library	L	H	[18]
B410	import_lxml	Import of a possibly insecure XML parsing library	L	H	[18]
B411	import_xmlrpclib	Import of a possibly insecure XML parsing library	H	H	[18]
B412	import_httpoxy	Exposition to the so-called “httpoxy” vulnerabilities	H	H	[57]
B413	import_pycrypto	Use of the deprecated Python Cryptography Toolkit (pycrypto)	H	H	[24]
B501	request_with_no_cert_validation	Ignoring the validation of certificates	H	H	[71]
B502	ssl_with_bad_version	Use of old and insecure Transport Layer Security (TLS) versions	H	H	[34]
B503	ssl_with_bad_defaults	Use of default parameter values that may yield insecure TLS transports	M	M	[34]
B504	ssl_with_no_version	Use of default parameter values that permit insecure TLS transports	L	M	[34]
B505	weak_cryptographic_key	Using inadequate key length for a cipher	H	H	[30]
B506	yaml_load	Insecure use of the load function from the PyYAML library	M	H	[39]
B507	ssh_no_host_key_verification	Not verifying keys with a Python library for Secure Shell (SSH)	H	M	[33]
B601	paramiko_calls	Exposure to command injection via a Python library for SSH	M	M	[63]
B602	subprocess_popen_with_shell_equals_true	Spawning a subprocess using a command shell	L, M, H	H, H, H	[69]
B603	subprocess_without_shell_equals_true	Spawning a subprocess without a shell but without input validation	L	H	[70]
B604	any_other_function_with_shell_equals_true	Using a wrapper method with a command shell (excluding B602)	M	H	[63]
B605	start_process_with_a_shell	Exposure to command injection with a shell (excluding B602 and B604)	L	M	[69]
B606	start_process_with_no_shell	Exposure to command injection without a shell (excluding B603)	L	M	[63]
B607	start_process_with_partial_path	Spawning a process without absolute path	L	H	[63]
B608	hardcoded_sql_expressions	Exposure to Structured Query Language (SQL) injection	M	L	[65]
B609	linux_commands_wildcard_injection	Exposure to command injection via Unix wildcards	H	M	[20]
B610	django_extra_used	Exposure to SQL injection via the Django framework	M	M	[65]
B611	django_rawsql_used	Exposure to SQL injection via the Django framework	M	M	[65]
B701	jinja2_autoescape_false	Exposure to Cross-Site Scripting (XSS) via a Python library	H, H	H, H	[64]
B702	use_of_mako_templates	Exposure to XSS via a Python library	M	H	[64]
B703	django_mark_safe	Exposure to XSS via a Python library	M	H	[64]

4) The fourth category is similar to the third, but instead of detection based on individual function calls, additional checks are present based on simpler *import statements*.
5) The fifth category (from B501 to B507) contains detectors for issues that relate to insecure *network protocols*. The examples include many well-known security-related issues related to TLS, validation, and authentication.
6) The sixth category contains various checks for different *code injections*. These range from conventional SQL injections to code injections via a command line shell.

7) The seventh category contains three simple checks for cross-site scripting, which has been a widespread security issue for web applications written in Python [54].

Although the seven categories cover plenty of specific and important issues, the list is hardly exhaustive. For instance, issues related to synchronization, unused code, risky numerical values, and resource management are not present. The limited coverage is unfortunate because resource management issues and buffer-related bugs have been even surprisingly common in Python applications [54]. The lack of coverage for numerical issues is also a known problem for many tools [44]. Though, it is practically impossible to gain a full coverage of the myriad of different security issues in any given tool, whether based on static analysis or something else. There is also a trade-off: covering the various nuts and bolts of a programming language tends to increase false positives and noise in general. Against this backdrop, it is worth remarking that the popular general-purpose `pylint` [46] was too noisy for the present purposes. The reason relates to the software ecosystem focus: although the various Python Enhancement Proposals (PEPs) cover different best practices, there are literally tens of thousands of individual coding styles used in the hundreds of thousands of packages in PyPI, and so on. All in all, the issues covered in Bandit reflect the particular security requirements and practices in the OpenStack project. Therefore, there are also some special cases (such as B201) that may not be relevant for Python packages in general. Some further limitations are discussed later on in Subsection V-A.

C. Methods

Descriptive statistics are used for answering to the first and second research questions. Following related work [56], [79], the well-documented [12] negative binomial (NB) regression is used to answer to RQ_3. Two NB equations are examined:

$$\text{E}(\textit{Issues}_i \mid \ln[\textit{Files}_i]) = \exp(\alpha_1 + \beta_1 \ln[\textit{Files}_i]) = \mu_i \quad (1)$$

and

$$\text{E}(\textit{Issues}_i \mid \ln[\textit{Lines}_i]) = \exp(\alpha_2 + \beta_2 \ln[\textit{Lines}_i]) = \gamma_i, \quad (2)$$

where $\text{E}(\cdot)$ denotes the expected value, $\textit{Issues}_i$ is the number of issues detected for the ith package, α_1 and α_2 are constants, β_1 and β_2 are coefficients, $\textit{Files}_i$ and $\textit{Lines}_i$ are the number of files and lines of code scanned for the ith package, μ_i and γ_i are shorthand notations for the conditional means, and $i = 1, \ldots, n$. Given a coefficient φ_j, the conditional variances are given by $\text{Var}(\textit{Issues}_i \mid \ln[\textit{Files}_i]) = \mu_i + \varphi_1 \mu_i^2$ and $\text{Var}(\textit{Issues}_i \mid \ln[\textit{Lines}_i]) = \gamma_i + \varphi_2 \gamma_i^2$. Compared to the Poisson regression, overdispersion is taken into account with the terms $\varphi_1 \mu_i^2$ and $\varphi_2 \gamma_i^2$. Overdispersion is an obvious concern in the present context—by assumption, there should be plenty of packages for which no security-related issues were detected. Stated differently, a mixing distribution is used:

$$\textit{Issues}_i \mid \lambda_i \sim \text{Pois}(\lambda_i) \text{ and } \lambda_i \sim \text{G}(\delta_i, \varphi_j), \quad (3)$$

where $\text{Pois}(\cdot)$ denotes the Poisson distribution, $\text{G}(\cdot)$ refers to the Gamma distribution, and $\delta_i \in \{\mu_i, \gamma_i\}$. With these two distributional assumptions, it can be further shown that the number of issues follows the negative binomial distribution.

The two software size metrics cannot be included in the same model because these are expectedly highly correlated with each other. Nevertheless, the basic expectation is simple: the estimated $\hat{\beta}_1$ and $\hat{\beta}_2$ should be both positive. Because the amount of code scanned increases more with additional files than with lines of code, it should also hold that $\hat{\beta}_1 > \hat{\beta}_2$. Given that a logarithm is used for $\textit{Files}_i$ and $\textit{Lines}_i$, the difference should not be substantial, however. Otherwise the coefficients are not straightforward to interpret because these are multiplicative with $\hat{\alpha}_1$ and $\hat{\alpha}_2$. For summarizing the results regarding RQ_3, averaging is used for the conditional means:

$$\overline{\hat{\mu}} = \frac{1}{n} \sum_{i=1}^{n} e^{\hat{\alpha}_1} \textit{Files}_i^{\hat{\beta}_1} \quad \text{and} \quad \overline{\hat{\gamma}} = \frac{1}{n} \sum_{i=1}^{n} e^{\hat{\alpha}_2} \textit{Lines}_i^{\hat{\beta}_2}. \quad (4)$$

The equations in (1) and (2) are estimated for (a) all issues detected, (b) low severity issues, (c) medium severity issues, and (d) high severity issues. Although the estimates for the subset models (b), (c), and (d) are not directly comparable because the sample sizes vary in these severity subsets, the expectations regarding $\hat{\beta}_1$, $\hat{\beta}_2$, and their effects still hold.

Finally, the results are also reported with bootstrapped estimates. Analogously to cross-validation, bootstrapping allows to infer about the accuracy and stability of the regressions (for technical details see [14], [62]). To some extent, bootstrapping can be further used to implicitly deduce about whether there is some generalizability toward the larger theoretical population of all open source Python packages. As for computation, the bootstrapping is implemented by drawing $1,000$ random samples (with replacement) containing $10,000$ observations of the dependent and independent variables. For each random sample, the equations (1) and (2) are then fitted for all four scenarios (a), (b), (c), and (d). The means of the averaged conditional means in (4) are used for reporting the results.

IV. Results

A. Issues (RQ_1)

In total, as many as $m = 749,864$ issues were detected for the n packages. Thus, on average, roughly about four security-related issues were detected across the Python packages. However, these issues seem to concentrate to particular packages since no issues were detected for a little below the half of the packages. As can be seen from Fig. 2, the majority of the issues have a low severity. Medium and high severity issues together account for about 41% of the m issues detected. This breakdown according to severity is not a superficial artifact from the tool used: according to Table I, the tool is not biased toward low-severity issues in general. Moreover, all four distributions visualized in Fig. 2 have a similar shape. That is, there are many packages with no issues, some packages with a few issues, and a minority with many issues. Of the 46% of all packages with at least one issue, the median number of issues is three and the 75th quartile is 7.

At the very end of the tail, there are five packages with more than a thousand detected issues: `PyGGI`, `appengine-sdk`,

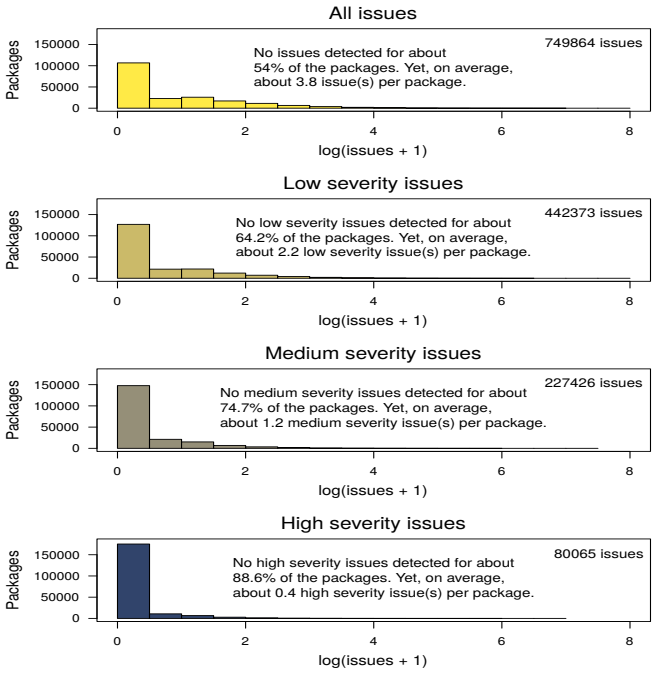

Fig. 2: Issues Detected

`genie.libs.ops`, `pbcore`, and `genie.libs.parser`, respectively. All have large code bases, which partially explains the large number of issues detected for these packages. However, interestingly, there exists variance across the types of issues detected. Of the $2,589$ issues detected for `PyGGI`, all are about the "try-except-pass" construct (B110), which, at least without further validation, may be more of a code smell. However: of the $2,356$ issues detected for `appengine-sdk`, only 395 belong to the generic category; there are also 351 issues belonging to the injection category, 500 issues that are about potential cross-site scripting, seven issues about potentially insecure use of network protocols, and so forth and so on. Even the single debug mode issue (B201) is present. Although these observations are partially explained by the fact that `appengine-sdk` embeds a large amount of third-party libraries directly to its code base, a closer look reveals numerous problematic and potentially insecure coding practices. These observations motivate to continue toward examining the issue types more generally.

B. Types (RQ_2)

The frequencies of the issues detected across the individual types and the seven categories are shown in Figs. 3 and 4. Analogous results are shown in Figs. 5 and 6 for the tool's detection confidence. In general, generic issues (including particularly the already noted B110) and different injections (including particularly those related to the `subprocess` module) have been the most common ones. Together the generic and injection categories account for over a half (52%) of all issues detected. However, these are mostly low severity issues. Although the tool's overall self-reported detection

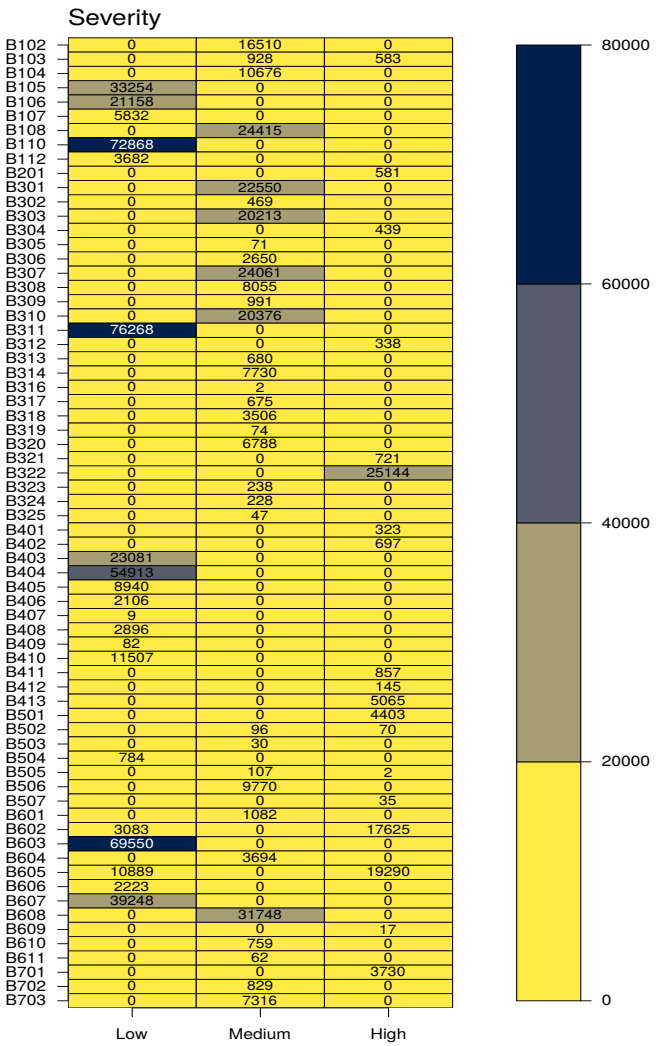

Fig. 3: Severity of the Issues (frequencies)

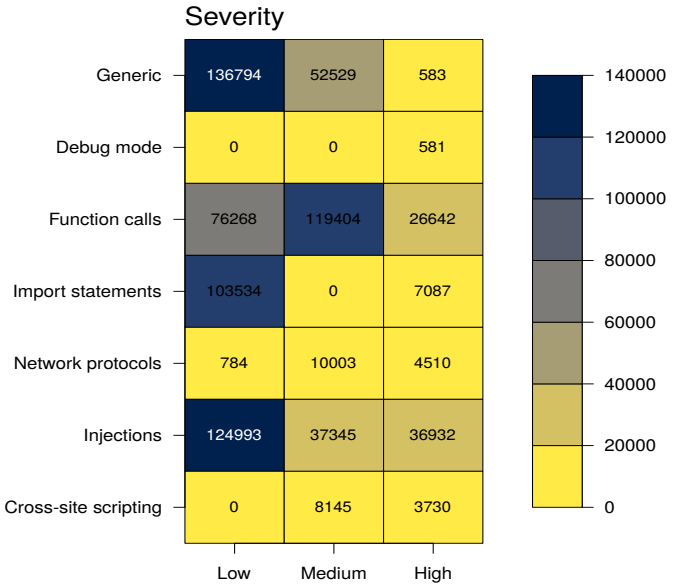

Fig. 4: Severity of the Issues Across Groups (frequencies)

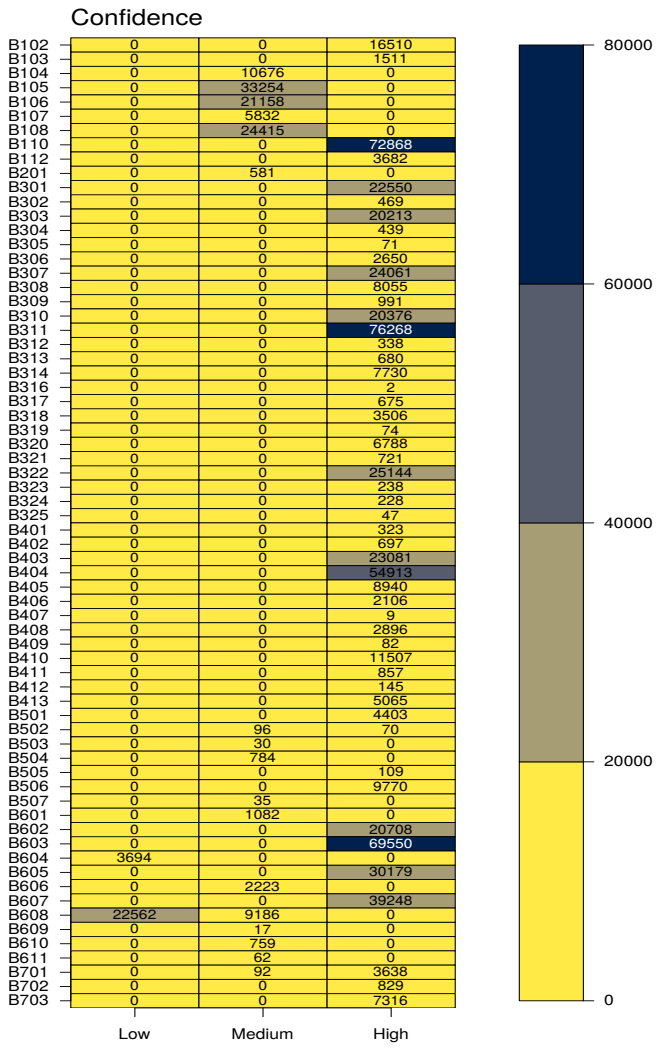

Fig. 5: Detection Confidence (frequencies)

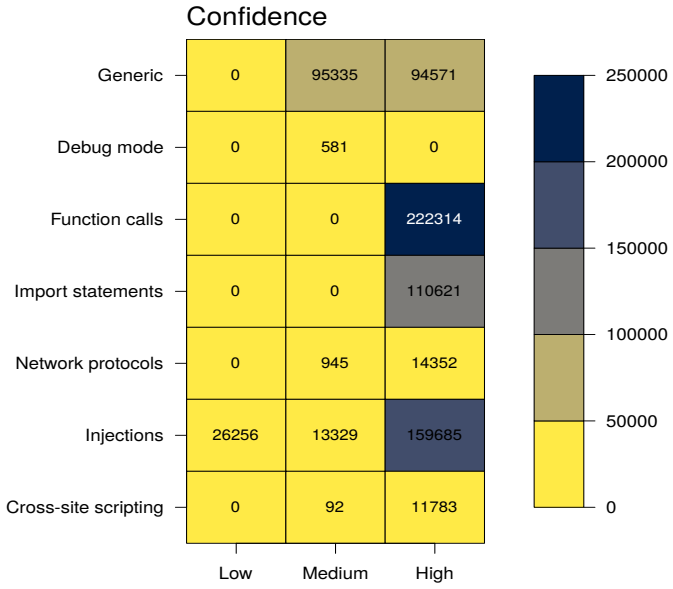

Fig. 6: Detection Confidence Across Groups (frequencies)

confidence is high, the generic and injection categories also gather most of the low and medium detection confidence cases.

These results align well with existing observations; command line injections and generally inadequate input validation have been typical to Python applications [54], [52]. In fact, input validation was already one of the kingdoms in the famous "seven pernicious kingdoms" of software security issues [75]. Some of the results may relate also to the age of the packages in PyPI. The over twenty thousand issues about insecure hash functions (B303) would be a good example in this regard. Nevertheless, the prevalence of different injections indicate that also Python's standard library is prone to misuse. While all of these issues are noted in the official documentation of the standard library, the issues still frequently appear in PyPI.

C. Software Size (RQ_3)

The Python packages in PyPI are generally small. This observation can be seen from Fig. 7, which shows the logarithms of the files and lines scanned. In terms of the files scanned, the largest package is `alipay-sdk-python` (9,801 files); with respect to lines, `huBarcode` is the largest with its about about 5.9 million lines of code scanned by Bandit. These two packages are clear outliers, however. For most of the packages, only about three to fifty Python files were scanned. Indirectly, these results indicate that some existing datasets for Python code (such as those used in [79] and [78]) are biased upwards toward large projects. In any case, the smallness of the PyPI packages and the shapes of the two distributions in Fig. 7 lead to expect that the conditional means are close to the unconditional means (cf. RQ_3). Before examining this prior expectation further, a few points are in order about the two negative binomial regressions (see Subsection III-C) estimated.

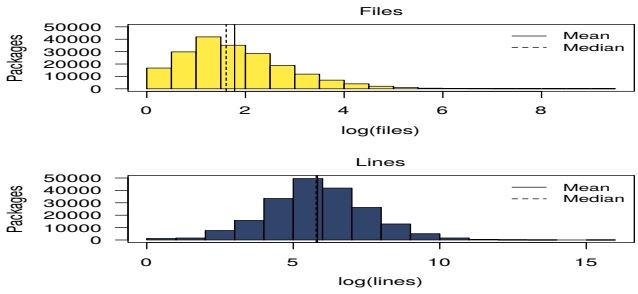

Fig. 7: Files and Lines of Code

The two NB equations (1) and (2) were estimated with the standard R (MASS) function `glm.nb`. The estimated dispersion parameters are nowhere near zero ($\hat{\varphi}_1 \approx 2.999$ and $\hat{\varphi}_2 \approx 2.381$), which indicates that plain Poisson regressions would have been inadequate. Although both regression coefficients are positive and $\hat{\beta}_1 > \hat{\beta}_2$, as was expected in Subsection III-C, the magnitudes of the coefficients are small: $\hat{\beta}_1 \approx 0.859$ and $\hat{\beta}_2 \approx 0.748$. Already due to the large n, these estimates are expectedly both statistically significant and accurate. The 95%-level confidence intervals are $[0.852, 0.866]$ for $\hat{\beta}_1$ and $[0.743, 0.754]$ for the second coefficient estimated.

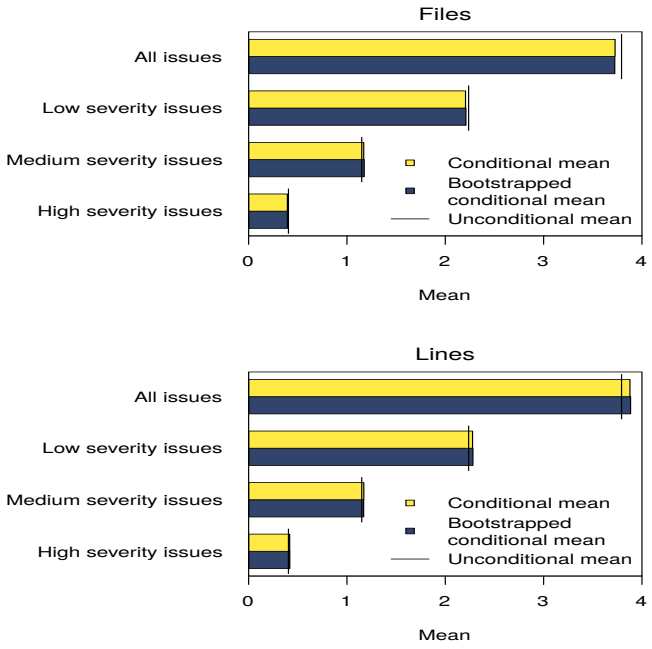

Fig. 8: Regression Results

The small magnitudes of the coefficients indicate relatively modest performance in general. Indeed, for instance, McFadden's [29] pseudo-R^2 values are only 0.09 and 0.14 for the two models including files and lines, respectively. These numbers are implicitly reflected in the final results shown in Fig. 8. As can be seen, the unconditional means are very close to the averaged conditional means computed via (4). The estimates are also highly accurate; the bootstrapped estimates are almost equal with the sample estimates in all eight setups.

V. Discussion

A. Threats to Validity

Recently, particular attention has been paid to different sampling strategies in empirical software engineering research [3]. The dataset used covers practically the whole PyPI, excluding the few packages that had to be excluded due to practical data processing reasons. Of course, there are commercial Python software, other ecosystems for Python packages (such as GitHub), and so forth. However, no attempts were made to infer beyond PyPI. Generalizability (a.k.a. *external validity*) is thus not a particular concern for the present work.

But there are concerns regarding the question whether the metrics used truly measure what is intended to be measured (a.k.a. *construct validity*). All metrics are extremely simple. There are neither interpretation problems nor theoretical assumptions. Instead, the potential issues are technical. Even with the detection confidence metric provided by Bandit, (a) there are obviously numerous false positives and negatives in the dataset. One one hand, the paper's large-scale approach makes it difficult to speculate about the extent of these. On the other hand, if even 0.01% of the issues were truly vulnerabilities, the amount would still be above seven thousand. To patch this tool-specific limitation in further work, Bandit's detection accuracy should be evaluated and compared against other static analysis tools. Given the security-oriented context, the question is not only about straightforward programming issues. For instance, many static analysis tools are known to make many mistakes with respect to cryptography [8]. A few further issues are also worth point out about construct validity.

Unlike in some previous studies [43], no attempts were made to distinguish different Python source code files; (b) the results are based also on code that many not be executed in production use. Test cases are a good example. Though, it is not clear whether the additional code constitutes a real problem; code quality and security issues apply also to test cases. The reverse also applies: (c) not all code is covered for some packages. For instance, there are packages that contain additional archives that were not extracted. Data files embedded directly to Python code are a good example. As is well-known [1], [54], packages are often written with multiple languages; yet, (d) the results do not cover non-Python code that may be particularly relevant in terms of security. In general and to some extent, these construct validity issues are lessened by framing the issues as code smells [52], [79].

Finally, the assumptions concerning cause and effect (a.k.a. *internal validity*) should be briefly contemplated. As was noted already in Subsection II-A, the two simple equations (1) and (2) do not posit meaningful causal relations. Though, the results presented allow to also question whether software size is even a meaningful confounding factor in the context of Python packages. Despite this probable assumption, (e) only a snapshot was used; the results are limited to one point in time. The omitted dynamics are a potential problem for the regression analysis, but the lack of attention paid to software evolution can be seen also as a problem of external validity.

B. Further Work

Some fruitful paths for further work can be mentioned. Patching some of the limitations discussed is the obvious starting points: further large-scale studies are needed to compare different static analysis tools for Python; further software metrics could be collected to make the simple predictive approach more interesting theoretically; longitudinal analysis is required for assessing the robustness of the effects reported; and so forth. The last point about longitudinal research setups is also important for knowing whether things are improving or getting worse. Although many individual projects continuously fix issues in their source code, it may be that such fixing does not improve the situation in the whole ecosystem. To this end, it would be interesting to examine whether automated solutions based on static analysis could be incorporated into the PyPI infrastructure. After all, automation and integration to continuous integration or other related software systems have often been seen important for making practical advances with static analysis [4], [9]. A simple option would be to scan a package via a static analysis tool each time the package is

uploaded to PyPI, and then, upon request, display the tool's output for a user, possibly via the `pip` package manager.

As for further empirical research, it would be interesting to examine whether the results differ between different types of Python packages. To this end, there exists previous work regarding the so-called "GitHub stars" [7]. In PyPI there are also "classifiers" [73] that could be used for the comparison.

VI. Conclusion

This paper examined a snapshot of almost all packages archived to PyPI. Three research questions were asked. To briefly summarize the answers to these, security issues are common in PyPI packages. At least one issue was detected for a little below half of the packages in the dataset; though, the majority of the issues observed are of low severity (RQ_1). The issue types show no big surprises: different code injections and generally inadequate input validation characterize most of the issues (RQ_2). Finally, the generally small size of the Python packages in PyPI implies that basic code size metrics do not predict well the per-package amount of issues detected (RQ_3).

References

[1] B. Aloraini and M. Nagappan. Evaluating State-of-the-Art Free and Open Source Static Analysis Tools Against Buffer Errors in Android Apps. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, pages 295–306, Shanghai, 2017. IEEE.

[2] H. Alves, B. Fonseca, and N. Antunes. Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. In *Proceedings of the 12th European Dependable Computing Conference (EDCC 2016)*, pages 37–44, Gothenburg, 2016. IEEE.

[3] S. Baltes and P. Ralph. Sampling in Software Engineering Research: A Critical Review and Guidelines. Archived manuscript, available online in April 2020: https://arxiv.org/abs/2002.07764, 2020.

[4] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, pages 470–481, Suita, 2016. IEEE.

[5] R. Benabidallah, S. Sadou, B. L. Trionnaire, and I. Borne. Designing a Code Vulnerability Meta-Scanner. In *Proceedings of the 15th International Conference on Information Security Practice and Experience (ISPEC 2019)*, pages 194–210, Kuala Lumpur, 2019. Springer.

[6] P. N. Borazjani. Security Issues in Cloud Computing. In M. H. A. Au, A. Castiglione, K.-K. R. Choo, F. Palmieri, and K.-C. Li, editors, *Proceedings of the 12th International Conference on Green, Pervasive, and Cloud Computing (GPC 2017), Lecture Notes in Computer Science (Volume 10232)*, pages 800–811, Cetara, 2017. Springer.

[7] H. Borges and M. T. Valente. What's in a Github Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, 2018.

[8] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira. Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software. *IEEE Transactions on Reliability*, 68(4):1384–1403, 2019.

[9] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[10] I. Chowdhury and M. Zulkernine. Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[11] Django Software Foundation. Security in Django. Available online in March 2020: https://docs.djangoproject.com/en/3.0/topics/security/, 2020.

[12] P. K. Dunn and G. K. Smyth. *Generalized Linear Models With Examples in R*. Springer, New York, 2018.

[13] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An Empirical Study on the Effectiveness of Security Code Review. In *Proceedings of the 5th International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*, pages 197–212, Paris, 2013. Springer.

[14] B. Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[15] N. E. Fenton and M. Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[16] A. Fromherz, A. Ouadjaout, and A. Miné. Static Value Analysis of Python Programs by Abstract Interpretation. In *Proceedings of the 10th NASA Formal Methods International Symposium (NFM 2018)*, pages 185–202, Newport News, 2018. Springer.

[17] A. Gkortzis, D. Mitropoulos, and D. Spinellis. VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018)*, pages 18–21, Gothenburg, 2018. ACM.

[18] C. Heime. defusedxml. Available online in March 2020: https://pypi.org/project/defusedxml/, 2020.

[19] K. Hjerppe, J. Ruohonen, and V. Leppänen. Annotation-Based Static Analysis for Personal Data Protection. In *Privacy and Identity Management. Data for Better Living: AI and Privacy. Proceedings of the 14th IFIP WG 9.2, 9.6/11.7, 11.6/SIG 9.2.2 International Summer School*, pages 343–358, Windisch, 2019. Springer.

[20] L. Juranic. Back to the Future: Unix Wildcards Gone Wild. Unpublished note, available online in March 2020: http://www.defensecode.com/public/DefenseCode_Unix_WildCards_Gone_Wild.txt, 2014.

[21] B. Kitchenham. What's Up With Software Metrics? – A Preliminary Mapping Study. *Journal of Systems and Software*, 83(1):37–51, 2010.

[22] G. Korkmaz, C. Kelling, C. Robbins, and S. Keller. Modeling the Impact of Python and R Packages Using Dependency and Contributor Networks. *Social Network Analysis and Mining*, 10(7), 2020.

[23] Y. Li. Empirical Study of Python Call Graph. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, pages 1274–1276, San Diego, 2019. IEEE.

[24] D. C. Litzenberge. Python Cryptography Toolkit (pycrypto). Available online in March 2020: https://pypi.org/project/pycrypto/, 2013.

[25] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang. A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*, Seoul, 2020. ACM.

[26] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing Software Ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[27] D. Malcolm. Static Analysis in GCC 10. Red Hat Developer Blog, available online in April: https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10/, 2020.

[28] B. A. Malloy and J. F. Power. An Empirical Analysis of the Transition from Python 2 to Python 3. *Empirical Software Engineering*, 24:751–778, 2019.

[29] D. McFadden. Conditional Logit Analysis of Qualitative Choice Behavior. In P. Zarembka, editor, *Frontiers in Econometrics*, pages 105–142. Academic Press, New York, 1974.

[30] MITRE. CWE-327: Use of a Broken or Risky Cryptographic Algorithm. Available online in March 2020: https://cwe.mitre.org/data/definitions/327.html.

[31] MITRE. CWE-502: Deserialization of Untrusted Data. Available online in March 2020: https://cwe.mitre.org/data/definitions/502.html.

[32] MITRE. CWE-319: Cleartext Transmission of Sensitive Information. Available online in March 2020: https://cwe.mitre.org/data/definitions/319.html, 2020.

[33] MITRE. CWE-322: Key Exchange without Entity Authentication. Available online in March 2020: http://cwe.mitre.org/data/definitions/322.html, 2020.

[34] MITRE. CWE-326: Inadequate Encryption Strength. Available online in March 2020: https://cwe.mitre.org/data/definitions/326.html, 2020.

[35] MITRE. CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG). Available online in March 2020: https://cwe.mitre.org/data/definitions/338.html, 2020.

[36] MITRE. CWE-755: Improper Handling of Exceptional Conditions. Available online in March 2020: https://cwe.mitre.org/data/definitions/755.html, 2020.

[37] MITRE. CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection'). Available online in March 2020: https://cwe.mitre.org/data/definitions/95.html, 2020.

[38] M. Nachtigall, L. N. Q. Do, and E. Bodden. Explaining Static Analysis – A Perspective. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW 2019)*, pages 29–32, San Diego, 2019. IEEE.

[39] National Institute of Standards and Technology (NIST). CVE-2017-18342 Detail. Available online in March 2020: https://nvd.nist.gov/vuln/detail/CVE-2017-18342, 2017.

[40] National Institute of Standards and Technology (NIST). CVE-2018-1281 Detail. Available online in March 2020: https://nvd.nist.gov/vuln/detail/CVE-2018-1281, 2018.

[41] National Institute of Standards and Technology (NIST). CVE-2019-9948 Detail. Available online in March 2020: https://nvd.nist.gov/vuln/detail/CVE-2019-9948, 2019.

[42] H. Ogasawara, M. Aizawa, and A. Yamada. Experiences with Program Static Analysis. In *Proceedings of the Fifth International Software Metrics Symposium (Cat. No.98TB100262)*, pages 109–112, Bethesda, 1998. IEEE.

[43] M. Orrú, E. Tempero, M. Marchesi, R. Tonelli, and G. Destefanis. A Curated Benchmark Collection of Python Systems for Empirical Studies on Software Engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2015)*, pages 1–4, Beijing, 2015. ACM.

[44] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Proceedings of the 19th Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2018)*, pages 86–103, Porto, 2018. Springer.

[45] Python Code Quality Authority (PyCQA) and contributors. Bandit. Available online in March: https://github.com/PyCQA/bandit, 2020.

[46] Python Code Quality Authority (PyCQA) and contributors. pylint. Available online in March: https://github.com/pycqa/pylint, 2020.

[47] Python Software Foundation. PEP 476 – Enabling Certificate Verification by Default for stdlib HTTP Clients. Available online in March 2020: https://www.python.org/dev/peps/pep-0476/, 2014.

[48] Python Software Foundation. PEP 551 – Security Transparency in the Python Runtime. Available online in March 2020: https://www.python.org/dev/peps/pep-0551/, 2017.

[49] Python Software Foundation. Miscellaneous Operating System Interfaces. Available online in March 2020: https://docs.python.org/2.7/library/os.html, 2020.

[50] Python Software Foundation. subprocess – Subprocess Management. Available online in March 2020: https://docs.python.org/3/library/subprocess.html, 2020.

[51] Python Software Foundation. tempfile – Generate Temporary Files and Directories. Available online in March 2020: https://docs.python.org/3/library/tempfile.html, 2020.

[52] M. R. Rahman, A. Rahman, and L. Williams. Share, But be Aware: Security Smells in Python Gists. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)*, pages 536–540, Cleveland, 2019. IEEE.

[53] A. Ronacher et al. Quickstart. Available online in March 2020: https://flask.palletsprojects.com/en/1.1.x/quickstart/, 2020.

[54] J. Ruohonen. An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications. In *Proceedings of the 9th International Workshop on Empirical Software Engineering in Practice (IWESEP 2018)*, pages 25–30, Nara, 2018. IEEE.

[55] J. Ruohonen. A Demand-Side Viewpoint to Software Vulnerabilities in WordPress Plugins. In *Proceedings of the 23rd Conference on the Evaluation and Assessment in Software Engineering (EASE 2019)*, pages 222–228, Copenhagen, 2019. ACM.

[56] J. Ruohonen. A Look at the Time Delays in CVSS Vulnerability Scoring. *Applied Computing and Informatics*, 15(2):129–135, 2019.

[57] D. Scheirlinck et al. httpoxy: A CGI Application Vulnerability. Available online in March 2020: https://httpoxy.org/, 2017.

[58] H. Shahriar and M. Zulkernine. Mitigating Program Security Vulnerabilities: Approaches and Challenges. *ACM Computing Surveys*, 44(3):11:1 – 11:46, 2012.

[59] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering*, 45(9):877–897, 2019.

[60] G. Spanos and L. Angelis. A Multi-Target Approach to Estimate Software Vulnerability Characteristics and Severity Scores. *Journal of Systems and Software*, 146:152–166, 2018.

[61] J. Spring, E. Hatleback, A. D. Householder, A. Manion, and D. Shick. Towards Improving CVSS. White Paper REV-03.18.2016.0, Carnegie Mellon University, Software Engineering Institute (SEI), available online in March 2020: https://resources.sei.cmu.edu/asset_files/WhitePaper/2018_019_001_538372.pdf, 2018.

[62] R. Stine. Introduction to Bootstrap Methods: Examples and Ideas. *Sociological Methods and Research*, 18(2–3):243–291, 1989.

[63] The Open Web Application Security Project (OWASP). Command Injection. Available online in March 2020: https://owasp.org/www-community/attacks/Command_Injection, 2020.

[64] The Open Web Application Security Project (OWASP). Cross Site Scripting (XSS). Available online in March 2020: https://owasp.org/www-community/attacks/xss/, 2020.

[65] The Open Web Application Security Project (OWASP). SQL Injection. Available online in Mach 2020: https://owasp.org/www-community/attacks/SQL_Injection, 2020.

[66] The Open Web Application Security Project (OWASP). Use of Hard-Coded Password. Available online in March 2020: https://owasp.org/www-community/vulnerabilities/Use_of_hard-coded_password, 2020.

[67] The OpenStack Project. OSSN/OSSN-0033: Some SSL-Enabled Connections Fail to Perform Basic Certificate Checks. Available online in March 2020: https://wiki.openstack.org/wiki/OSSN/OSSN-0033, 2013.

[68] The OpenStack Project. Create, Use, and Remove Temporary Files Securely. Available online in March 2020: https://security.openstack.org/guidelines/dg_using-temporary-files-securely.html, 2015.

[69] The OpenStack Project. Python Pipes to Avoid Shells. Available online in April 2020: https://security.openstack.org/guidelines/dg_avoid-shell-true.html, 2015.

[70] The OpenStack Project. Use subprocess Securely. Available online in March 2020: https://security.openstack.org/guidelines/dg_use-subprocess-securely.html, 2015.

[71] The OpenStack Project. Validate Certificates on HTTPS Connections to Avoid Man-in-the-Middle Attacks. Available online in March 2020: https://security.openstack.org/guidelines/dg_validate-certificates.html, 2015.

[72] The OpenStack Project. Apply Restrictive File Permissions. Available online in March 2020: https://security.openstack.org/guidelines/dg_apply-restrictive-file-permissions.html, 2016.

[73] The Python Package Index (PyPI). Classifiers. Available online in April 2020: https://pypi.org/classifiers/, 2020.

[74] The Python Package Index (PyPI). Simple Index. Data retrieved in 28 March from: https://pypi.org/simple/, 2020.

[75] K. Tsipenyuk, B. Chess, and G. McGraw. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[76] M. Valiev, B. Vasilescu, and J. Herbsleb. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, pages 644–655, Lake Buena Vista, 2018. ACM.

[77] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How Developers Engage with Static Analysis Tools in Different Contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[78] X. Xia, X. He, Y. Yan, L. Xu, and B. Xu. An Empirical Study of Dynamic Types for Python Projects. In *Proceedings of the 8th International Conference on Software Analysis, Testing, and Evolution (SATE 2018)*, pages 85–10. Springer, 2018.

[79] C. Zhifei, C. Lin, M. Wanwangying, Z. Xiaoyu, and X. Baowen. Understanding Metric-Based Detectable Smells in Python Software: A Comparative Study. *Information and Software Technology*, 94:14–29, 2018.

[80] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, pages 995–1010, Santa Clara, 2019. USENIX.