The Journal of Systems & Software 202 (2023) 111698

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Simple stupid insecure practices and GitHub's code search: A looming threat?☆

Ken Russel Go [a], Sruthi Soundarapandian [a], Aparupa Mitra [a], Melina Vidoni [b,*], Nicolás E. Díaz Ferreyra [c]

[a] RMIT University, School of Computing Technologies, Australia
[b] Australian National University, CECS School of Computing, Australia
[c] Hamburg University of Technology, Institute of Software Security, Germany

ARTICLE INFO

Article history:
Received 31 July 2022
Received in revised form 27 March 2023
Accepted 3 April 2023
Available online 6 April 2023

Keywords:
Python
GitHub code search
Simple stupid insecure practices

ABSTRACT

Insecure coding practices are a known, long-standing problem in open-source development, which takes on a new dimension with the current capabilities for mining open-source software repositories through version control systems. Although most insecure practices require a sequence of interlinked behaviour, prior work also determined that simpler, one-liner coding practices can introduce vulnerabilities in the code. Such *simple stupid insecure practices (SSIPs)* can have severe security implications for package-based software systems, as they are easily spread over version-control systems. Moreover, GitHub is piloting regular-expression-based code searches *across public repositories* through its "Code Search Technology", potentially simplifying unearthing SSIPs. As an exploratory case study, we focused on popular PyPi packages and analysed their source code using regular expressions (as done by GitHub's incoming search engine). The goal was to explore how *detectable* these simple vulnerabilities are and how exploitable "Code Search" technology is. Results show that packages on lower versions are more vulnerable, that "code injection" is the most scattered issue, and that about 20% of the scouted packages have at least one vulnerability. Most concerningly, malicious use of this engine was straightforward, raising severe concerns about the implications of a publicly available "Code Search".

1. Introduction

Software vulnerabilities can make a software-intensive system an easy target for cyber-attacks such as information loss, disclosure of secrets, manipulation, and system failure. Vulnerabilities are more impactful in open-source software, where libraries and packages are openly shared through version control systems such as GitHub (Alfadel et al., 2021).

Since December 2021, GitHub has been openly working on an improved advanced search to allow perusing source code across all public repositories by using *regular expressions* (regex).[1] This proposal is concerning because GitHub (and its standard API and search) has already been considered 'perilous' in terms of exposing vulnerabilities (Lazarine et al., 2020). In particular, GitHub's incoming "Code Search" can enhance the exposure of software vulnerabilities and insecure practices—in turn, a malicious user could identify *insecure practices* through GitHub's regex search, find 'suspicious code', and analyse it to 'narrow down' software systems susceptible to attacks.

Scenarios like the above can be riskier for those insecure practices that are extremely simple, often consisting of a single line of code, which scatter into multiple GitHub repositories. (e.g., calling `hashlib.md5()` makes that line susceptible to collision attacks Rahman et al., 2019b). Therefore, inspired by Karampatsis and Sutton (2020) analyses on software bugs, we define *simple stupid insecure practices* (SSIP) as "insecure practices that appear on a single function call, and whose corresponding fix implies removing that call or replacing it with a secure alternative". Note that an *insecure practices* has potential to become a vulnerability under given circumstances, but may not currently be a vulnerability.

In this work, we investigated how exploitable GitHub's "Code Search Technology" is with regards to SSIPs. We did this through a case study of commonly-installed Python packages and leveraging a previously developed list of Python insecure practices (Rahman et al., 2019b). We chose Python because (A) it is the currently most used programming language (as per the IEEE Spectrum Ranking,[2]) (B) our goal was to evaluate the 'search process' and

☆ Editor: Neil Ernst.
* Corresponding author.
E-mail addresses: melina.vidoni@anu.edu.au (M. Vidoni), nicolas.diaz-ferreyra@tuhh.de (N.E.D. Ferreyra).

[1] https://github.blog/2021-12-08-improving-github-code-search/

[2] https://spectrum.ieee.org/top-programming-languages-2022

https://doi.org/10.1016/j.jss.2023.111698
0164-1212/

not a coding environment, and (C) Python is a package-based environment, which enhances SSIPs' risk. Prior works demonstrated that the openness and scale of package-based environments (such as R's CRAN, Python's PyPi and Node.js' npm) "lead to the spread of vulnerabilities through the package network, making the vulnerability discovery much more difficult, given the heavy dependence on such packages and their potential security problems" (Alfadel et al., 2021).

Overall, we aimed to answer the following research questions (RQs): **RQ1**, how many SSIPs can be found in Python packages by using regexes? **RQ2** How frequent and scattered are those SSIPs? (namely, if this is an issue worth considering), and **RQ3** Which package type and SSIP are the most affected? Answering these three questions would contribute to our main goal: *how exploitable could be GitHub's new regex-based "Code Search" if made public?*

Our findings reveal that packages on lower versions (such as version zero) are more prone to include SSIPs, and that "code injection" is the most scattered SSIP. About 20% of the scouted packages had at least one vulnerability, with others representing concerning outliers with high numbers of vulnerabilities. Nevertheless, through a basic combination of regexes and a public list of insecure practices, we highlight that the malicious use of this incoming "Code Search" engine was straightforward.

2. Related work

MSR-based privacy studies. Valiev et al. (2018) analysed the sustainability of over 46k Python projects in PyPi through a mixed-methods study and determined that project ties and position in the dependency network impact sustained project activity. Bommarito and Bommarito (2019) summarised almost 179k PyPi packages, determining most packages have a large variability on imports (totalling over 150 million), thus being highly affected by transitive issues. Finally, Abdalkareem et al. (2020) compared Javascript's npm and Python's PyPi and determined trivial packages are perceived to be well implemented and tested; however, only 49% of PyPi packages have tests, and about 3% of trivial packages had over 20 dependencies.

Regarding security, Alfadel et al. (2021) assessed 550 vulnerability reports for 252 Python packages in PyPi; when analysing the propagation and lifespan of these packages, they determined that only half are fixed after publication. Kaplan and Qian (2021) compared npm and PyPi security vulnerability reports and determined that around 134k-610k packages depend on other vulnerable packages and that the ecosystems' encouragement to reuse code accelerates the growth rate of highly vulnerable packages. Vu et al. (2020) used code repositories to detect injections in a package's distributed artefacts, concluding the technique was suitable for lightweight analysis only. In contrast, Ohm et al. (2020) analysed malicious packages used in attacks and available in PyPi, npm and RubyGems to provide a dataset for future studies. Finally, Rahman et al. (2019b) demonstrated that about 31% of Python Gists (i.e., code snippets) shared in GitHub have at least one security smell (including hard-coded secrets),

Malicious search engines. Regarding search engine's malicious use, Fischer et al. (2021) demonstrated that there is a 22.8% chance that one out of the top three Google Search results leads to insecure code on Stack Overflow. Likewise, Rahman and Roy (2014) created a code-recommender system by exploiting GitHub's standard advanced search API; although their search was done regarding exception handling, the issues with GitHub's search exploitability are evident. Finally, other studies demonstrated how straightforward it is to find insecure code (but not specific vulnerabilities or insecure practices) through commonly available search engines (Acar et al., 2017).

Code search tools. Over the years, many code search tools have been proposed to aid developers. Gu et al. (2018) proposed a deep neural network named `CODEnn` implemented on a proof-of-concept tool trained only in Java; this allowed to write textual descriptions akin to summarised comments (e.g., `sort a map by values`), returning a list of relevant methods. However, using this tool required training the models first, and it was only tested on a statically-typed language (i.e., Java).

Likewise, `CodeBERT` is a bimodal pre-trained model for programming and natural languages that learns general-purpose method coding (Feng et al., 2020), while `CuBERT` (Kanade et al., 2020) was specifically trained on 7.4M Python files using the existing docstrings (namely, Python's documentation). Both models were trained based on the natural language comments or documentation and their nearby comment. Although tools as `CodeBERT` and `CuBERT` have simplified code inspection, they are based on what the developers document, which may be misleading or missing entirely (e.g., a SSIP is unlikely to be explicitly documented beyond a `TODO` comment).

Lastly, Pearce et al. (2022) investigated GitHub's CoPilot[3] and found that about 40% of what it produces is vulnerable; however, although CoPilot was trained using code search, their end-users (i.e., developers) use it during *programming tasks*, and not when searching across GitHub's OSS projects.

3. Case study methodology

We followed existing methodologies for mining software repositories (MSR) (Vidoni, 2022). Fig. 1 summarises the entire process for the case study, from repository selection to code inspection.

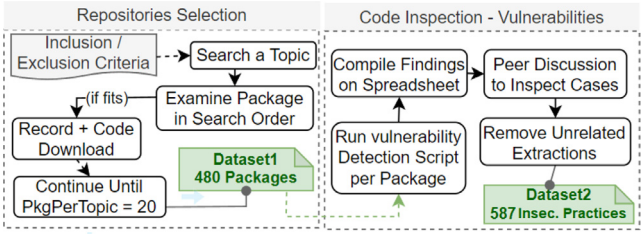

Fig. 1. Process followed to mine the repositories and data from code.

3.1. Repositories selection

Our goal was to test GitHub's regex-based "Code Search" on Python packages. Python was selected due to being the most used programming language, according to IEEE Spectrum[2].

Sources selection. PyPi[4] is the Python Package Index—namely, the official third-party software repository for Python. In particular, PyPi increased from about 118k packages in 2017 (Abdalkareem et al., 2020) to >345k by December 2021. Because PyPi is official, it allows packages to be easily installed. Although many Python packages are hosted publicly as open-source software (OSS) in GitHub (or similar version control systems, like GitLab or BitBucket), their GitHub repository remains a host for the code and does not provide any assistance for quick installations as PyPi does.

Therefore, we used PyPi to determine the most popular and actually installed Python packages. For this, we selected the packages with the largest number of users, as these are more likely to be the target of malicious individuals (Garcia et al., 2014).

[3] https://github.com/features/copilot
[4] https://pypi.org

PyPi organises available packages by *topic*—non-exclusive, developer-assigned themes to classify a package's work area. When mining, PyPi had 24 available topics selected by the developers when registering a package; thus, a package may have multiple topics, although it is uncommon. This provided us with an additional layer of categories to assess our results, as Simmons et al. (2020) has determined that each programming language has *idioms*—namely, that every discipline and/or domain main code differently.

Package sampling. Given the large number of packages available in PyPi, we calculated a sample size (with 95% confidence, 5% error) for the total number of packages; this resulted in 385 packages. Nevertheless, this type of sampling did not ensure representation across topics as it would yield different proportions of packages-per-topic. However, calculating per-topic samples with the same configuration (95% confidence, 5% error) resulted in over 3700 packages. This was also problematic given that some topics had few packages and would be studied entirely or would 'fall short' of the sample size, while others would have only a percentage.

Therefore, we decided to collect a flat number of packages per topic, **20 packages per topic, resulting in a final sample of 480 Python packages**, equivalent to a representative sample of 95% confidence and 4.5% error.

Selection criteria. The search was conducted manually through PyPi's website *first*, and not through GitHub. For each PyPi topic (unrelated to GitHub's topics), we selected the top-20 packages sorted by 'trending' *on PyPi*; this order ensured we obtained the *currently most-installed packages*. Note that a package 'trending' on PyPi does not mean that the same package will be 'trending' as well on GitHub. This difference happens because PyPi and GitHub are separate, unrelated platforms—because of this, PyPi's 'installations' *do not equate* to any of GitHub's own statistics (e.g., forks, stars, watches). Namely, PyPi considers 'trending' by installations, while GitHub does so by undisclosed rules.

Therefore, we chose to leverage *Pypi's trending criteria* to obtain the 'currently most installed' packages, since out purpose was to analyse vulnerabilities in packages that are being frequently installed and used. Nevertheless, because we aimed to assess GitHub's code search, we added the following *inclusion criteria*: a package listed in PyPi had to include a link to a working, public GitHub repository.

Three authors conducted the search in rotating pairs to mitigate biases, completing the following steps: (1) a topic was searched using the criteria, (2) every package was read in search order (top to bottom, from the first result at the top), (3) manually reviewed the package's site (to find a code repository in GitHub), and (4) if it was considered a fit to the inclusion criteria, it was added to a spreadsheet where additional data was manually recorded.

If during (3) the package did not fit the criteria or was already included in another topic, it was skipped. When a package had multiple topics, it was discussed with another author to determine the most relevant topic (in terms of crucial functionalities). This process continued until 20 unique packages per topic were selected.

The extraction spreadsheet D_1 included data related to the package itself: package ID and name, PyPi and GitHub URL, latest version (number and date), search date and position, the retrievers' names, and the package's source code identified by package ID.

3.2. SSIP detection

This section presents GitHub's "Code Search" process and the steps we took for its replication and assessment.

Fig. 2. Screenshot of GitHub's Code Search Suggestions.

3.2.1. GitHub's regexes

The goal of searching by regex was to mimic GitHub's potential behaviour; so far, little is known of the tool except that it allows for a scoped search (similar to the regular engine), file paths (e.g., `path:/.*shrc$/`), exact matches, boolean operations, and regular expressions (e.g., `/<Route(b|>)/`). Fig. 2 presents a screenshot of GitHub's "Code Search" where they show example regexes; as seen, the search process is simple and straightforward, given the availability of online tools to craft and test regexes.[5] Once inputted on the search bar, GitHub's engine peruses existing OSS repositories, and returns a snippet of the files where the match was found (see Fig. 3).

Fig. 3. Results from a regex search on GitHub's "Code Search" Preview.

3.2.2. Process replication

Rahman et al. (2019a) analysed OpenStack's catalogue of Python insecure coding practices systematically, mixing source code analysis (of over 529k code blocks) and StackOverflow discussions (almost 45k posts). Through that process, they determined six generic groups of insecure practices that may appear through different functions, each with single-call to a 'coding

[5] For example, https://regexr.com.

Table 1
Python's SSIPs, classified by Rahman et al. (2019a), including impact and example function call.

Category	Explanation	Example Coding Practices
Code Injection	Coding patterns susceptible to arbitrary code or command injection. *For example, `eval` is susceptible to executing harmful commands without validation.*	`exec`, `input`, `eval`
Cross-Site Scripting	(XSS) Enable injection of client-side scripts into web pages. In particular, this is considered among OWASP's vulnerabilities for web applications[a].	`django.utils.safestring.mark_safe`
Insecure Cipher	Weak cryptography algorithms or predictable random number generators. These practices rely on two Python functions that use weak cryptography algorithms. *For example, the `hashlib.md5()` method makes that line susceptible to collision attacks (Rahman et al., 2019a).*	MD2 (e.g., `Crypto.Hash.MD2.new`), MD4 (e.g., `Crypto.Hash.MD4.new`), MD5 (e.g., `hashlib.md5`, `cryptography.hazmat.primitives.hashes.MD5`), Random generation (e.g., uniform, `randint`, choice, triangular)
Insecure Connection	Using hypertext transfer protocol (HTTP) and the file transfer protocol (FTP) to create and open connections. This is vulnerable to sniffing, spoofing, and brute force, among other basic attacks, and may enable OWASP vulnerabilities of insecure communication[c].	`httpslib`, `urllib`
Race Condition	Using coding patterns that enable the creation of temporary files with predictable paths, increasing the possibility of time of check, time of use attacks	`mktemp`
Untrusted Serialisation	Using data serialisation-related coding patterns without validating the authentication of the source. This is considered a vulnerability among the CWE (Common Weakness Enumeration), already detected in Python[b]	`pickle.loads`, `marshal.loads`

[a] https://owasp.org/www-community/attacks/Code_Injection.
[b] https://cwe.mitre.org/data/definitions/502.html.
[c] https://owasp.org/www-project-mobile-top-10/2016-risks/m3-insecure-communication.

pattern' (summarised in Table 1). Because the insecure practices proposed by Rahman et al. (2019a) fit our definition of SSIP, we decided to search for these single-call functions in the same way that GitHub's new "Code Search Technology" works (see Section 3.2.1) to assess how exploitable GitHub's new search engine may be.

We first crafted the regexes as the function name defined by Rahman et al. (2019a), followed by an opening bracket, and allowed any subsequent characters; for example, `exec(*`.

Then, we created a Python script to read packages' files as plain text (including nested sub-directories) and used the regexes above to search for the SSIPs. In particular, we selected the simplest approach to imitate the behaviour "Code Search" (seen in Fig. 3)–to be selected as a result, a line of code had to contain the regex, regardless of whether it was at the start, middle or end of the line.

Thereby, our parsing script retrieved *all* occurrences per file but only scanned files of the format `*.py`. For those cases in which Rahman et al. (2019a) only provided the packages to import, we: (1) identified files containing the import, and (2) searched for key function names in the identified file.

Because the script worked only with simple, language-agnostic regexes, it collected (a) 'active' code lines, (b) 'commented-out' code lines (i.e., 'dead code'), (c) source code comments mentioning a function by name, (d) markdown text in notebooks justifying a decision and mentioning the insecure function, or (e) code lines with a similar syntax that were not insecure practices. An example for (e) is that, when searching for `exec()`, sometimes we found developer-made functions with the name 'exec' that were not related (nor behaved like) the vulnerability uncovered by Rahman et al. (2019a); e.g., `msg.exec()`.

After each assessment, we transferred the results to a spreadsheet (D_2), which included: package ID and Name, SSIP, path (folder sequence where it was found), filename (containing the SSIP), location (row-column tuple), and code-line (the extracted code snippet).

When D_2 was completed, three authors discussed each detected code-line and conducted a manual data cleaning to remove invalid results. Cases of (e; similar name) required revising the entire code snippet by using the 'location' given by the script. Therefore, through the manual analysis, we filtered cases (c, code comments), (d, markdown) and (e, jupyter text). Cases (b, dead code) were removed as they are an instance of 'dead code' (namely, fragments of code no longer used), which are acknowledged as poor coding practices (Romano et al., 2020). Thus, although we identified these cases through the regexes, they were not taken into account as part of the vulnerabilities counts.

At the end of the filtering, we uncovered **586 code vulnerabilities across packages and topics.**

4. Results

The following subsections assess RQ1–RQ3, and how they contribute to answering our main question: *how exploitable could be this new regex-based code search?*

4.1. RQ1. Detecting SSIPs

Through a simple algorithm, we detected 586 SSIPs, after removing *23 false positives*. This means that from the total extracted (609 lines), less than 4% were not SSIPs, but cases (b) dead code, (d) markdown text in notebooks, and (e) developer-made functions, as per the process defined in Section 3.2. The manual 'clean up' ensured that we were only assessing potential SSIP, while the low number of discarded lines indicates that basic regexes can be effective in detecting SSIPs; however, evaluating the degree of effectiveness was not a goal of this study.

After this, we determined that **20% of the selected packages** had at least one SSIP (exactly 95 packages) detected through basic regexes. The most common SSIPs and their detected frequencies are summarised in Table 2; note that `*parse(` (the last row), was searched as both `xml.dom.minidom.parse(`, and as the combination of import plus function call.

"Code Injection" was the most frequent SSIP category, as the three assessed function calls (i.e., `exec`, `input`, `eval`) were found in large numbers across the projects. This is problematic since, as determined by OWASP,[6] "Code Injection" is a basic vulnerability

[6] https://owasp.org/www-community/attacks/Code_Injection.

Table 2
Number of occurrences of detected SSIP.

Category	SSIP Regex	#
Code Injection	`input(`	159
Code Injection	`exec(`	104
Race Condition	`mkstemp(`	89
Code Injection	`eval(`	84
Untrusted Data Serialisation	`pickle(`	76
InsecureCipher	`md5(`	61
Untrusted Data Serialisation	`DOM(`	6
XSS	`mark_safe(`	5
Untrusted Data Serialisation	`xml.dom.minidom.parse(`	2

that manifests through an SSIP and that can be found in almost every programming language. Thus, being able to detect it so easily through regexes presents a risk to the vulnerable code.

These results align with other authors' findings regarding how widespread simple vulnerabilities are Kaplan and Qian (2021), Alfadel et al. (2021), and the exploitability of search engines (Acar et al., 2017; Fischer et al., 2021). The large number of detected SSIPs is an alarming finding with regards to how simple it was–purposefully, the regexes had no intricate combination of characters and were searched as contained inside a line–this indicates that if GitHub releases its "Code Search Technology" regex-based search, it could potentially increase the exposure of SSIPs to malicious users.

Note that at the moment of writing this manuscript, GitHub's "Code Search Technology" remains a preview and can only be used after requesting access[1]; likewise, this technology is not yet available through GitHub's API, commonly used for mining software repositories (Vidoni, 2022). Nevertheless, the possibility of combining the advanced search with a similar script remains a real risk.

4.2. RQ2. SSIPs' spread

The `__init__.py` is the most vulnerable file across packages with 47 SSIPs, followed by files with `*setup.py` (custom files, not necessarily the original setup), which grouped 21 SSIPs. In the particular case study of Python packages, these results indicate that the setup/initialisation moment could prone to security vulnerabilities.

Overall, the 586 SSIPs were spread across 290 files–meaning that some packages clustered more than one SSIP per file. On a descriptive level, 63.4% of the files (exactly 184) had a single SSIP, but 17.5% (exactly 51) files had two SSIPs. About 5% of the files had four SSIPs in the same file; a similar proportion of files had either three SSIPs or six SSIPs. Exactly eight files had between 7–17 SSIPs *per file*. This is visible in Fig. 4; intuitively, files containing more than one vulnerability are more at risk of being uncovered through a regex search, as they have more than one function call that could expose them.

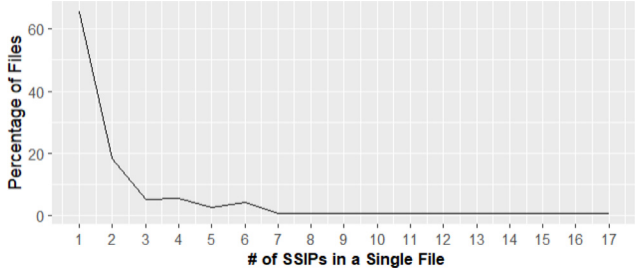

Fig. 4. Percentage of files with each sum of SSIPs.

Although there are some concerning outliers, **most files (among those detected with the regexes) have a single SSIP**, and few have more than one. Table 3 presents how many affected files per package were found (i.e., number of packages for each number of SSIP-affected files). As can be seen, 56.8% of the packages had a single SSIP-affected file, while 18.9% had two affected files and overall, *almost a fifth of the scouted packages were SSIP-affected.*

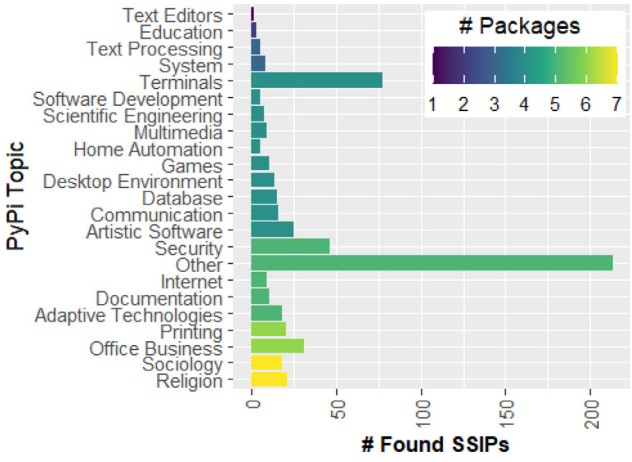

Fig. 5. Compromised PyPi topics per number of package and found SSIPs.

Table 3
Number of SSIP-affected files per package.

# Affected Files	1	2	3–4	5–7	9–11	34	63
# of Packages	54	18	14	4	3	1	1

From this, it is visible that the scattering of SSIPs is not dense but instead found as a few vulnerabilities per package. Although this reduces the impact of SSIPs as security threats, the vulnerabilities exist and can be easily spotted with minimal effort.

4.3. RQ3. Packages' characteristics

Regarding compromised PyPi's topics, Fig. 5 showcases how many SSIPs were found (x-axis) per topic (y-axis) and how many packages (colour) contributed to that amount. Note that we mined twenty packages per topic (see Section 3.1).

The topic 'Other' was the most affected, with only five packages (a quarter of the topic) contributing 214 SSIPs; from which 191 belonged to a single package. 'Terminals' was the second most affected topic, with four packages contributing to 77 SSIPs; from those, 71 belonged to a single package. Looking at individual packages, the third most vulnerable package (with 26 SSIPs) was categorised in the topic 'Security' and the fourth in 'Office Business', accounting for 17 SSIPs. This simple demographic shows that although vulnerable outliers have a high chance of being detected, even an inconspicuous single SSIP can be unearthed through a regex search, thus highlighting the inherent risks of GitHub's new Code Search.

Fig. 6 presents topics (y-axis) and their SSIP count (x-axis), breaking down the vulnerabilities by category name (as defined in Table 1), and colour indicating the amount of similarly-affected packages. Overall, "Code Injection" had 347 SSIPs detected (out of 587 SSIPs total), followed by "Race Condition" with 89 SSIPs. There are PyPi topics (such as 'Games', 'Communication', 'Home Automation') that are exclusively affected by "Code Injections"–meaning that "Code Injections" represent all of its packages' SSIPs, while in other topics (such as 'Other', 'Terminals', 'Security', 'Documentation') the vast majority of SSIPs are "Code Injections".

The above finding matches prior results by other authors (Rahman et al., 2019b) and is concerning given that "Code Injection" is often considered one of the most dangerous vulnerabilities (Gautam et al., 2018). Moreover, although we cannot disclose the

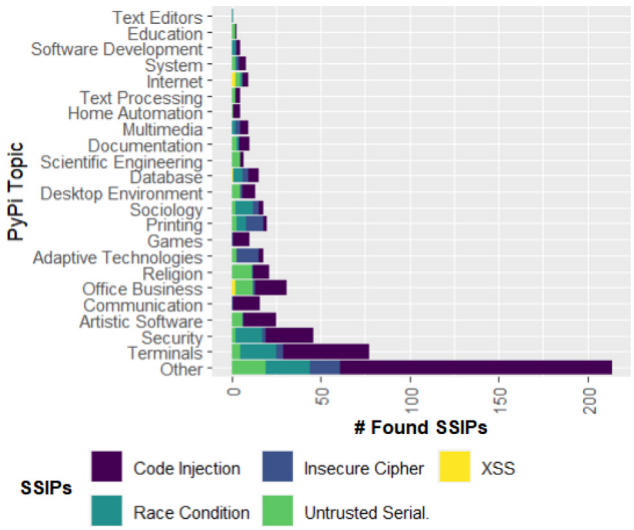

Fig. 6. Compromised PyPi topics and number of SSIPS per Category.

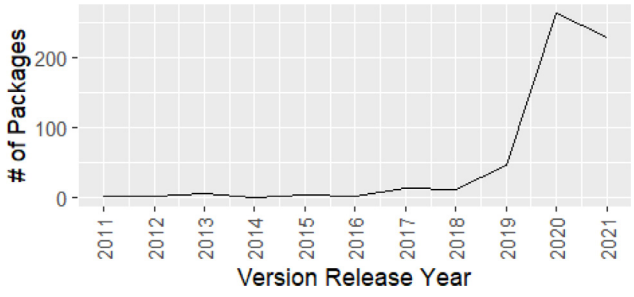

Fig. 7. Number of vulnerable packages per year of the latest version.

affected packages' names (see Section 5), some cases with plenty of SSIPs were prominent, renowned packages.

Regarding releases, we classified the versions according to those using Semantic Versioning format[7] (`major.minor.patch`), or per-year (`year.major.minor`). Table 4 shows version-zero packages account for 454 out of 586 SSIPs; these 454 SSIPs belonged to 56 version-zero packages, out of 291 selected version-zero packages (hence, representing 12.3% of the evaluated packages). These results make them the most vulnerable group—reasonable, given that Semantic Versioning considers that "major version zero is for initial development. The public API should not be considered stable"[7].

Table 4
Vulnerable packages per major version (using SemVer style). 'Date' refers to those using per-year versioning.

Major Version	0	1	2	3	4	8	Date
# of SSIPs	454	51	26	10	28	10	7

Fig. 7 shows the version release year and the vulnerable packages found. We mined the current/latest/active version on the master/main branch to imitate GitHub's search. From our original search, there were 143 packages with a last-release committed in 2020–2021 (out of 470 scouted packages), yet they account for almost 84% of SSIPs. Additionally, 46 packages with the last-releases from 2019 remain vulnerable, 13 packages from 2018, and 14 from 2017. Although it is reasonable that the more packages available the more vulnerabilities we obtain (the case of 2021–2020 releases), the concerning observation comes from the remainder of the packages as there are plenty of packages from (or before) 2019 and up to 2009, which continue to be consistently installed and used, yet they still have unfixed vulnerabilities. This means some packages have not received updates since their last release years ago, and whose vulnerabilities have remained unresolved since then—yet they are still searchable, reinforcing their vulnerability.

This finding is concerning, especially as older unfixed vulnerabilities become easier to find and more commonly known, hence more exploitable (Iannone et al., 2022). Although our preliminary study did not focus on the time-to-fix SSIPs, this supports other prior studies, regarding that plenty SSIPs may take years to fix or even remain exposed for an undetermined period (Iannone et al., 2022).

5. Discussion & implications

GitHub's new "Code Search Technology" brings significant advantages when compared to more traditional code exploration approaches. Currently, the inspection of codebases for potentially-malicious purposes entails (i) pre-selecting specific OSS projects, (ii) analysing their quality reports when available, (iii) downloading the project code, and (iv) run a detection tool to detect possible backdoors. Likewise, current tools using deep learning to search code and natural language comments (such as `DeepCS` (Gu et al., 2018), `CodeBERT` (Feng et al., 2020), `CuBERT` (Kanade et al., 2020)) are large-scale pre-trained models that require to be installed and further trained to detect specific code segments; additionally, they analyse locally available code. Conversely, (as seen in Figs. 2–3), GitHub's "Code Search" presents a simple user interface with a browser/query bar which, given a regex, provides matching lines of code across all OSS available on GitHub.

Due to the new advances in "Code Search Technology", finding SSIPs is a matter of crafting a simple regex, typing it in a search bar, and having the specific lines of code across hundreds of projects provided as search results, without having to download nor analyse any code. Therefore, "Code Search Technology" provides a considerable simplification to a malicious task that, before, required large amounts of craftsmanship. The fact that a simple regex allowed us to find so many potential SSIPs is concerning.

Although tools as `DeepCS`, `CodeBERT` and `CuBERT` can be evolved into AI-assistants for the early detection of vulnerabilities (Nguyen et al., 2022), they do not mitigate the threat of finding insecure practices through the use of simple regexes; in fact, prior works found that can be used to generate exploits (Yang et al., 2023). Similarly, GitHub's CoPilot[3] and similar tools, have been found to produce about 40% of vulnerable code (Pearce et al., 2022), and that relying on its suggestions generates code with more vulnerabilities (Perry et al., 2022). Given the simplicity of accessing and using CoPilot, and the demonstrated ease of detecting vulnerabilities through "Code Search", we argue that GitHub's own search engines could become a perilous endeavour exposing SSIPs vulnerabilities.

This research did not investigate Python's vulnerabilities, but attempted to assess how easy it is to exploit GitHub's new "Code Search Technology" with minimal effort. The answer is clear—it is alarming how simple it is to misuse this technology with some minimal regex knowledge and by leveraging public, open lists of insecure functions (such as the one we used, from Rahman et al., 2019b). Although public lists of vulnerabilities and insecure practices (considered as *potential vulnerabilities*) are helpful to developers, their combination with GitHub's "Code Search" engine can become threatening.

As discussed, such a technology could potentially simplify the identification of insecure code and the 'mining' of security

[7] https://semver.org.

vulnerabilities, reducing the time required to assess high-risk vulnerabilities and possibly allowing attackers to narrow their search. Therefore, in the following subsections, we discuss the Implications of this study and plausible Future Works.

5.1. Implications

The findings yielded in our study have the following implications:

For researchers, our results demonstrate the need to *beta test* new technologies in carefully controlled environments to analyse whether they may introduce "backfire effects" and security vulnerabilities. We hope this short paper acts as a catalyst for further actions assessing the ethical implications and the inherited risks of technologies alike, especially in the era of the internet of things (IoT) (Vidoni and Diaz-Ferreyra, 2022).

For PyPi, the results of this work could inspire them to set up automated checks for each package submission, as other package repository already do (e.g., CRAN, which is R's equivalent to PiPy) (Kumar et al., 2022). Thereby, PyPi could implement a simple regex-oriented approach to automatically check for SSIPs inside packages upfront (i.e., upon upload) and warn their corresponding authors. For the sake of transparency, such security assessment should also be available to "package consumers" (namely those using these packages) to make them aware about potential vulnerabilities in their implementations.

For GitHub and similar platforms, our results translate into practical advice to expedite actions and limit the scope of this new "Code Search" feature. However, given the myriad of actions that git enables, there is no "one-size-fits-all" solution.

For example, restricting "Code Search" to a user's own repositories (namely, regex-based search is available only within the repositories a user owns, either public or private) would likely mitigate most threats as other package repository already do (e.g., CRAN, which is R's equivalent to PiPy). Nevertheless, this could raise issues if a user *forks* an existing public repository with the sole purpose of enabling a regex-based search of SSIPs. A less restrictive approach could allow searching on organisations or collaborations where a user has actively contributed within a period. Still, this approach requires determining who are legitimate contributors of such repositories and who control their access.

Allowing repository owners to set permissions for regex-based search on a repository's setting[8] would give them the ability to control who can search and what. However, it may not be enough to control this problem, since former legitimate user can become malicious for a myriad of causes.

Overall, our advice to GitHub and other platforms proposing new tools is simple—take the time to perform a *beta testing*, not only with users but also with researchers who are likely to study the implications of said new tools from an interdisciplinary perspective/viewpoint. Currently, our world is becoming increasingly AI-driven, and assessing the ethical, privacy, social, and economic implications of the new technology offered openly to the public is paramount.

5.2. Future works

Although this work aimed to assess how easy it is to exploit GitHub's new "Code Search Technology" with minimal knowledge, our findings enable a range of future investigations.

From a security-oriented perspective, we demonstrated that SSIPs are pervasive. Albeit assessing their security impact was out of the scope of this work, it presents an opportunity for future research. At the same time, the prevalent vulnerability of packages categorised as "Terminal" warrants further investigation.

Additionally, given the current technologies for large-scale mining of software repositories (Lazarine et al., 2020; Karampatsis and Sutton, 2020), simply committing SSIPs to an open-source, publicly available-repository can present an exploitable risk. Thus, further studies are required to assess: (a) how long these vulnerabilities exist, (b) who and why introduces/removes them, and (c) how transitive dependencies can further the reach of these SSIPs.

Future works can continue to assess simple exploitable pathways for spotting SSIPs, and provide more comprehensive advice to platforms such as GitHub. The case study presented in this paper can be considerably extended (e.g., through a larger dataset) to assess the persistency of these vulnerabilities, and how transitive dependencies can further the reach of these SSIPs across other programming languages.

From a usability-perspective, we observed that version-zero packages contain 77.3% of all the identified SSIPs, so future studies could focus on assessing the human-factors enabling the decisions causing this trend. Likewise, given that "Code Injection" is the most common type of SSIPs, another empirical study could assess why developers continue to rely on these methods, regardless of their security limitations and drawbacks.

From an ethical perspective, further studies should investigate the need for *beta testing* and quality assurance processes encompassing the ethical and social implications of new technologies of this kind. Examples of heedless technology spread to the public with privacy-threatening outcomes have become all too common in the current times (e.g., LensaAI's case with the generation of nudes,[9] and the use of CoPilot and Codex to cheat on programming assignments Finnie-Ansley et al., 2022), demonstrating the need to extend traditional texting strategies with ethical and privacy-enhancing dimensions.

5.3. Ethical considerations

Enabling the reproducibility and the continuation of research is essential (Erb et al., 2021). However, the datasets used here are intrinsically affected by the dichotomy of 'privacy vs security' (Vidoni and Diaz-Ferreyra, 2022)–we are studying insecure code practices on commonly-installed Python packages.

Therefore, even if we (a) share de-identified datasets, or (b) present detailed statistics of the studied packages, we could be indirectly contributing to identifying the vulnerable packages. Likewise, sharing the list of selected packages will enhance the exposure of the detected SSIPs. Given search engines' current capabilities (Erb et al., 2021; Daskevics and Nikiforova, 2021), which was part of the motivation of this work, sharing the detected lines of code could also allow inferring which packages contain them.

As we are conscious of the risks of sharing the dataset of this study, to avoid further enhancing the vulnerability of the studied packages, we are only sharing the aggregated data through the report in this manuscript.

6. Threats to validity

To a certain extent, the results of our study are subject to limitations related to its experimental design.

Construct. Searching packages through GitHub was problematic; a code-based search would bias the sample towards having more SSIPs while searching by repository could yield packages

[8] GitHub Docs: Managing your repository's settings and features.

[9] What does the Lensa AI app do with my self-portraits and why has it gone viral?, and The inherent misogyny of AI portraits.

not commonly installed by users. By mining packages from PyPi trends, we mitigated both threats while ensuring that the sample was representative of the moment the mining took place, favouring packages currently installed the most.

Internal. We mined PyPi's trending packages only, and the possibility of less-installed packages having a different spread of SSIPs is considered negligibly small. Due to a large number of PyPi packages (345k), it was not feasible to complete a large-scale study, as it would include severely outdated, no longer installed packages. Moreover, completing a cross-language search was deemed not necessary, as the search methodology can be easily replicated.

External. Our findings regarding the search engine's misuse are generalisable to other languages to some extent, as it will require using a language-appropriate list of SSIPs for each language (namely, the list of Python function calls we used Rahman et al., 2019b cannot be used to search for SSIPs in Ruby). Our discussion of trends and Python package characteristics may not apply to other package-based domains; this is less important, as the goal of this study was to use Python packages as a case study to evaluate the searching-by-regexes exploitability.

7. Conclusions

This paper presents a preliminary investigation of the ease of uncovering SSIPs (or Simple Stupid Insecure Practices) through regex-based searches in a way similar to GitHub's new "Code Search Technology". We defined SSIPs as "insecure practices that appear on a single function call, and the corresponding fix is by removing that call or replacing it with a secure alternative".

To study this, we worked on a case study based on popular, commonly installed PyPi packages, but our methodology can be generalised to any programming language. We mined 480 popular Python packages and determined that about 20% have at least one vulnerability. Vulnerable version-zero packages contain 77.3% of all the detected SSIPs. Almost 16% of the detected SSIPs belong to packages with a last-release between 2009–2019 that continue to be consistently installed–this may indicate a trend to not fix SSIPs. Our findings confirm that (as per prior works), "Code Injection" is the most common type of SSIPs. Finally, PyPi's topics of 'Other', 'Terminals' and 'Security' encompass the most vulnerabilities due to a few outlier files.

Nevertheless, our goal was to assess how easy it is to exploit GitHub's new "Code Search Technology" with minimal knowledge—namely, simple regexes based on the SSIP's names, and publicly available lists of insecure function calls. Our findings are clear; it is concerningly easy to exploit this search engine to uncover multiple cases of SSIPs. Given the simplistic approach, although packages/repositories with more than one SSIP-affected file are at higher risk of appearing in multiple searches, many were uncovered regardless of the version year.

CRediT authorship contribution statement

Ken Russel Go: Data Curation, Software, Formal analysis, Investigation. **Sruthi Soundarapandian:** Data Curation, Software, Formal analysis, Investigation. **Aparupa Mitra:** Data Curation, Software, Formal analysis, Investigation. **Melina Vidoni:** Conceptualization, Methodology, Supervision, Writing. **Nicolás E. Díaz Ferreyra:** Conceptualization, Methodology, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

References

Abdalkareem, R., Oda, V., Mujahid, S., Shihab, E., 2020. On the impact of using trivial packages: An empirical case study on npm and PyPI. Empir. Softw. Eng. 25 (2), 1168–1204.

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., Stransky, C., 2017. How internet resources might be helping you develop faster but less securely. IEEE Secur. Priv. 15 (2), 50–60. http://dx.doi.org/10.1109/MSP.2017.24.

Alfadel, M., Costa, D.E., Shihab, E., 2021. Empirical analysis of security vulnerabilities in Python packages. In: IEEE International Conference on Software Analysis, Evolution and Reengineering. pp. 446–457.

Bommarito, E., Bommarito, M.J., 2019. An empirical analysis of the Python package index (PyPI). CoRR, arXiv:1907.11073.

Daskevics, A., Nikiforova, A., 2021. ShoBeVODSDT: Shodan and Binary Edge based vulnerable open data sources detection tool or what Internet of Things Search Engines know about you. In: International Conference on Intelligent Data Science Technologies and Applications. IDSTA, pp. 38–45.

Erb, B., Bosch, C., Herbert, C., Kargl, F., Montag, C., 2021. Emerging privacy issues in times of open science. PsyArXiv. URL http://dx.doi.org/10.31234/osf.io/u236e.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. CoRR arXiv:2002.08155.

Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J., 2022. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In: Australasian Computing Education Conference. ACE '22, Association for Computing Machinery, USA, pp. 10–19. http://dx.doi.org/10.1145/3511861.3511863.

Fischer, F., Stachelscheid, Y., Grossklags, J., 2021. The effect of google search on software security: Unobtrusive security interventions via content re-ranking. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. CCS '21, Association for Computing Machinery, New York, NY, USA, pp. 3070–3084. http://dx.doi.org/10.1145/3460120.3484763.

Garcia, A., Sun, Y., Shen, J., 2014. Dynamic platform competition with malicious users. Dynam. Games Appl. 4 (3), 290–308. http://dx.doi.org/10.1007/s13235-013-0102-y.

Gautam, B., Tripathi, J., Singh, S., 2018. A secure coding approach for prevention of SQL injection attacks. Int. J. Appl. Eng. Res. 13 (11), 9874–9880.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, Association for Computing Machinery, New York, NY, USA, pp. 933–944. http://dx.doi.org/10.1145/3180155.3180167, URL https://doi-org.virtual.anu.edu.au/10.1145/3180155.3180167.

Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., Palomba, F., 2022. The secret life of software vulnerabilities: A large-scale empirical study. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2022.3140868.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code. In: 37th International Conference on Machine Learning. ICML '20, JMLR.org, pp. 1–12.

Kaplan, B., Qian, J., 2021. A survey on common threats in npm and PyPi registries. In: Wang, G., Ciptadi, A., Ahmadzadeh, A. (Eds.), Deployable Machine Learning for Security Defense. Springer International Publishing, Cham, pp. 132–156.

Karampatsis, R.-M., Sutton, C., 2020. How often do single-statement bugs occur? The ManySStuBs4J dataset. In: 7th International Conference on Mining Software Repositories. ACM, USA, pp. 573–577.

Kumar, P., Ie, D., Vidoni, M., 2022. On the developers' attitude towards CRAN checks. In: IEEE/ACM 30th International Conference on Program Comprehension. ICPC, IEEE Computer Society, Pittsburg, USA, pp. 570–574. http://dx.doi.org/10.1145/3524610.3528389.

Lazarine, B., Samtani, S., Patton, M., Zhu, H., Ullman, S., Ampel, B., Chen, H., 2020. Identifying vulnerable GitHub repositories and users in scientific cyberinfrastructure: An unsupervised graph embedding approach. In: International Conference on Intelligence and Security Informatics. ISI, pp. 1–6.

Nguyen, V.-A., Nguyen, D.Q., Nguyen, V., Le, T., Tran, Q.H., Phung, D., 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In: International Conference on Software Engineering: Companion Proceedings. ICSE '22, Association for Computing Machinery, USA, pp. 178–182. http://dx.doi.org/10.1145/3510454.3516865.

Ohm, M., Plate, H., Sykosch, A., Meier, M., 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In: Maurice, C., Bilge, L., Stringhini, G., Neves, N. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, Cham, pp. 23–43.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2022. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In: Symposium on Security and Privacy. IEEE, pp. 754–768. http://dx.doi.org/10.1109/sp46214.2022.9833571.

Perry, N., Srivastava, M., Kumar, D., Boneh, D., 2022. Do users write more insecure code with AI assistants?. arXiv. http://dx.doi.org/10.48550/ARXIV.2211.03622.

Rahman, A., Farhana, E., Imtiaz, N., 2019a. Snakes in paradise?: Insecure python-related coding practices in stack overflow. In: 16th International Conference on Mining Software Repositories. IEEE/ACM, Canada, pp. 200–204.

Rahman, M.R., Rahman, A., Williams, L., 2019b. Share, but be aware: Security smells in python gists. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 536–540. http://dx.doi.org/10.1109/ICSME.2019.00087.

Rahman, M.M., Roy, C.K., 2014. On the use of context in recommending exception handling code examples. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. pp. 285–294. http://dx.doi.org/10.1109/SCAM.2014.15.

Romano, S., Vendome, C., Scanniello, G., Poshyvanyk, D., 2020. A multi-study investigation into dead code. IEEE Trans. Softw. Eng. 46 (1), 71–99.

Simmons, A.J., Barnett, S., Rivera-Villicana, J., Bajaj, A., Vasa, R., 2020. A large-scale comparative analysis of coding standard conformance in open-source data science projects. In: International Symposium on Empirical Software Engineering and Measurement. ESEM '20, Association for Computing Machinery, USA, pp. 1–11. http://dx.doi.org/10.1145/3382494.3410680.

Valiev, M., Vasilescu, B., Herbsleb, J., 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, ACM, USA, pp. 644–655.

Vidoni, M., 2022. A systematic process for mining software repositories: Results from a systematic literature review. Inf. Softw. Technol. 144, 106791.

Vidoni, M., Diaz-Ferreyra, N., 2022. Should I get involved? On the privacy perils of mining software repositories for research participants. In: 1st International Workshop on Recruiting Participants for Empirical Software Engineering. IEEE Computer Society, USA, pp. 1–3.

Vu, D.L., Pashchenko, I., Massacci, F., Plate, H., Sabetta, A., 2020. Towards using source code repositories to identify software supply chain attacks. In: ACM SIGSAC Conference on Computer and Communications Security. CCS '20, ACM, USA, pp. 2093–2095.

Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2023. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 197, 111577. http://dx.doi.org/10.1016/j.jss.2022.111577.

Dr Vidoni is a Lecturer (eq. to Assistant Professor) at the Australian National University, CECC School of Computing. She has ongoing domestic and international collaborations with Canada and Germany. Dr Vidoni's main research interests are mining software repositories, technical debt, software development, and empirical software engineering when applied to data science and scientific software.

She graduated from Universidad Tecnologica Nacional (UTN) as an Information Systems Engineer. Years later, she received her Ph.D. on the same institution, with the maximum qualification. She founded R-Ladies Santa Fe in 2018, and in 2019 moved to Australia to further her research career. Between 2018–2022, she was also Associate Editor for rOpenSci. In 2022, Dr Vidoni joined the Editorial Board of Information and Software Technology.

Dr Ferreyra is a senior researcher and lecturer at the Institute of Software Security of Hamburg University of Technology. His main research focus stands at the intersection of human–computer interaction and privacy engineering, seeking to create technological solutions for supporting the cybersecurity decisions of social network users and software developers.

Before joining the Hamburg University of Technology, Dr Ferreyra worked as a postdoctoral fellow at the University of Duisburg–Essen. From January 2020 to October 2021, he was the coordinator of the RTG "User-Centered Social Media" funded by the German Research Foundation (DFG). Between August 2018 and September 2021, he participated in the H2020 project "PDP4E: Methods and Tools for GDPR Compliance through Privacy and Data Protection Engineering". In the past, he worked as a software engineer in Denmark and as an undergraduate research assistant in Argentina.