System & Software Security: Analyzing Python projects to identify security vulnerabilities

Guus Kleinlein (s3745880) Sven Hepkema (s2454556)
Max Boone (s2081318)

December 13, 2023

Abstract

Developers' dependence on free open-source software (FOSS) packages poses a security risk, as malicious or compromised dependencies can infiltrate downstream software, creating a cascading impact. Package repositories, such as PyPI for Python, play a vital role in preventing and detecting such threats. Recent research indicates that the current fulfillment of this role relies on a socio-technical framework and advocates for this framework over malware scanning tools. In this socio-technical framework, security researchers report malicious packages for personal benefit. We reevaluate these findings with further data and tools, to test if the accuracy and precision of scanning tools is still not sufficient for serious consideration. We confirm that the false positive and false negative rates are too high for tools to be a serious contender, and that three out of the four tested tools report a higher occurrence of potentially malicious behaviour in safe and popular packages than in malicious packages.

1 Introduction

As software products are continued to be developed developers seek to implement their products as effectively and efficiently as possible. One consequence of this is that developers will outsource some complexity and functionality of their software to existing solutions. The main source of these existing solutions come in the form of free open-source software (FOSS) packages. This dependency on FOSS packages has become an attack vector for malicious or post-release compromised dependencies to be compiled into its downstream software. If the downstream software is also a FOSS package it means that all its downstream software may also become vulnerable. Causing a cascading effect, multiplying the impact of the malicious code.

FOSS congregate around package managers, platforms such as PyPI and npm provide ways for developers to search and implement open-source software for use in their own software. Consequently this means that these platforms are an essential pivot in both the spread and mitigation of these malicious packages. It is essential for managers of such platforms to avoid malicious packages from entering their repository and to detect malicious packages for their removal.

A recent paper on this topic with regard to the PyPI platform is "Bad Snakes: Understanding and Improving Python Package Index Malware Scanning" by Vu, Newman and Meyers [18]. In their paper they interview repository managers tasked with this job and analyse available detection tools for malicious code. Such tools could aid in keeping the repositories clean. In this research we will revisit their analysis and expand it with more data extra and updated tooling.

Extending on their work we rewrite the pipeline and scripts use for data collection and analysis to handle a larger and more diverse amount of packages. Furthermore, we ensure that the analysis runs inside a sandbox to keep the host environment clean. Finally, we run the same descriptive statistical analysis on the results, compare them to their outcomes and validate whether the tools would work to classify packages as malicious or benign.

In the following section we'll further explore the findings of the research by Vu, Neyman and Meyers and how our research relates to it. Then, we discuss the methodology of our experiment in

four parts, from collection to analysis. Third, we describe the results from the experiments. Finally, we discuss the results and conclude by comparing the outcomes to the aforementioned work.

2 Background

The "Bad Snakes" paper consists of two main sections. In the first section, the authors examine how the work of those responsible for the housekeeping of PyPI is impacted by malicious packages. They observe that there is currently a socio-technical system in place for the safekeeping of the repository. In this system, individuals such as academics or security experts have incentive to review and find security issues and malicious software to report to registry administrators. Their work is motivated through the reports gaining them publicity.

The interviewees state they are quite positive with how the system currently works, and they also state that the false positive error rate of existing malware scanners is too high to be deployed registry wide. Even though positive with the status quo, it may be useful to further research the possible applications and characteristics of malware scanners for use in registries. In the second section of their research they explore such malware scanners for use on the PyPI repository and assess their usefulness.

As the tools need to be used on the PyPI repository, a set of possible malware scanners is selected based on three criteria. One, the source code must be available. Two, the detection must be anomaly-based. Third, the scanners must use a rule-based detection method with the rules being publicly available. Through this, they select three tools to asses: Bandit4Mal [17] (a fork of Bandit [11] from one of the authors), the then used PyPI YARA Rules [1] and OSSGadget's OSS Backdoor Detection [9], a tool from Microsoft.

Finally, the authors note that the socio-technical system satisfies all the requirements revealed during interviews by the administrators. This includes near zero false-positives, low effort of adoption and little maintenance. They also find that false negatives are acceptable and that scanning for malicious packages may be resource intensive as long as it does not require long processing time. Concluding, the researchers recommend that the socio-technical system should get further investment and aid by academics.

The paper is written with rigor and the logical steps taken by the researchers take are traceable, there are no obvious omissions in the process used to reach the conclusions. However the main take-away of focusing on stimulating the socio-technical system has some minor gripes. We suggest that by nature of this organically produced system it is hard to steer and maintain the system's integrity.

This because in a system where each actor acts upon only their own interest it cannot guarantee trust in the whole system. This point was also made as the reason PyPI administrators are weary of facilitating monetary rewards for actors that find malicious packages. As this would inadvertently also stimulate the creation of malicious packages.

In our research, we evaluate if this is still the best case by assessing the threats to validity to the part of the research that went into tooling [18, p.p. 1632]. The authors state that the used datasets of malicious packages might not be a good representation of real malware on the PyPI repository. They base this on differences in size and veracity of collected malware compared to what interviewees suggest. Furthermore, they state that the tools they used for the analysis were limited and expansion of this set is preferable.

3 Methodology

In this section we reflect on the used methodology for the research which consists of four major parts, visually represented in Figure 1. First, collect relevant benign and malicious Python packages. Second, prepare a safe and consistent runtime environment for analysis. Third, scan the packages in parallel and collect the output. Finally, combining the results and aggregating them to the original paper.

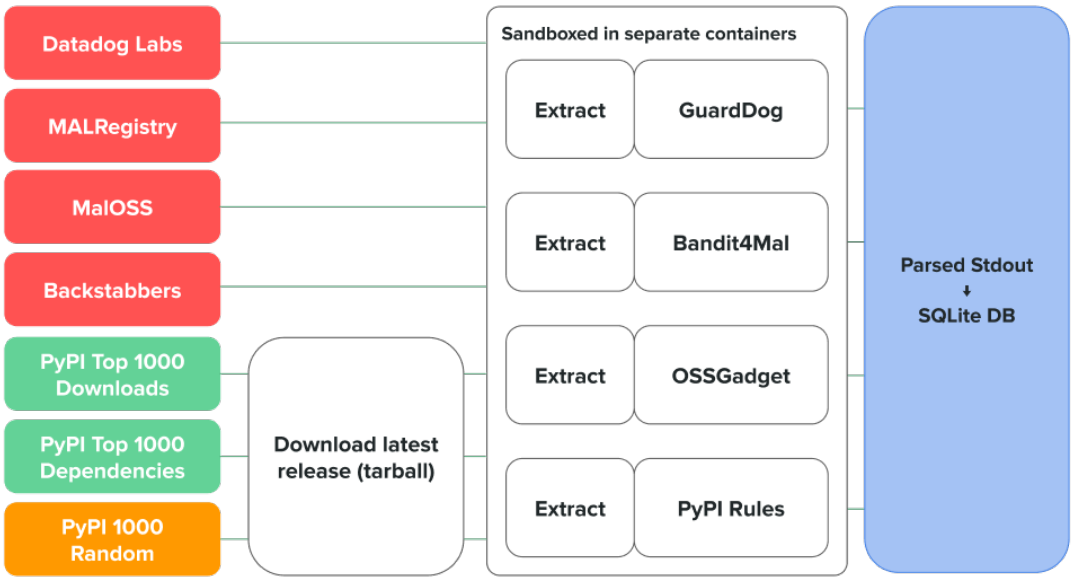

Figure 1: Visual overview of the processing pipeline

3.1 Data Collection

Initially, we intended to use the supplied supplemental information from the authors on GitHub [3]. However, the code of the notebooks referred to various hard-coded file paths which required extensive debugging. Moreover, the notebooks did not use parallelisation for the scanning which was an absolute necessity for the size of our experiment. Furthermore, the Backstabber [10] and MalOSS [2] datasets that the authors used were only available on request. Thus, we rewrote the entire pipeline to reproduce the results from the authors based on their descriptions. As we couldn't ensure timely access and to add more malicious packages to the scan, we included the datasets MALRegistry [4] and DataDog's Malicious Software [7]. The datasets we used for the malicious packages are listed in Table 1.

Dataset	Access	Packages	Active	#Packages
Backstabbers [10]	Restricted	PyPI	✓ (09-2023)	3024
MalOSS [2]	Restricted	PyPI, NPM, Ruby, ...	✓ (08-2023)	608 (579 `*.tar.gz`)
MALRegistry [4]	Open	PyPI	✓ (09-2023)	2951
DataDog [7]	Open	PyPI	✓ (08-2023)	1397

Table 1: Sources for known malicious packages

For the benign packages, we collect the 1000 most used dependencies and the 1000 most downloaded packages. Furthermore, we do a sampling of 1000 random packages from PyPI. A list of the most used dependencies is obtained from libraries.io [15], and a list of the most downloaded packages is obtained from a monthly dump [16]. We select 1000 random packages from PyPI from a list of package names on PyPI, then from those packages we download the latest releases. For the random packages, we filter out the packages without Python code and re-run the sampling if under 1000. The sources are listed in Table 2.

Target	Source	Access	Release	#Packages
Top 1000 Downloaded	Monthly Dump	Open	Listed	1000
Top 1000 Dependencies	Libraries.io	Limited	Latest	1000
1000 Random	PyPI	Open	Latest	1000

Table 2: Sources for benign-considered packages

3.2 Sandboxing the analysis tools

A similar matrix such as in the previous work is used to select tools for scanning, as shown in Table 3, adding GuardDog [6] to the set, checking all requirements.

Tool	OSS	Anomaly-based	Rules
Bandit4Mal [17]	✓	✓	✓
OSSGadget [9]	✓	✓	✓
PyPI Malware Checks [1]	✓	✓	✓
DataDog GuardDog [6]	✓	✓	✓

Table 3: Used tools for malware scanning

Even though the reviewed tools use static analysis, we were not previously familiar with them and don't know how secure they are. Furthermore, for a repeatable environment, we prefer to scan each package from a clean base state. This way, a previous run can impossibly alter successive runs. Another concern is that the malicious packages (even though we do static analysis) might exploit a mechanism of the tools or might cause a crash of the tool due to unexpected contents.

To achieve this, we built a Docker [8] image for each tool to run them inside a gVisor [14] sandbox environment. The tools and the build-up of their docker-image is shown in Table 4, only GuardDog and OSSGadget provided (instructions to build your own) docker image. To each image, we added an entrypoint script that inflates and extracts a `tar.gz` or `zip` file found in the `tmpfs`-mount at `/tmp` to be used by the tool.

Tool	Base	Size	Rootless
Bandit4Mal	`python:3.10`	`1.02 GiB`	×
OSSGadget	`ubuntu:22.04`	`221 MiB`	×
PyPI	`python:3.10`	`1.01 GiB`	×
DataDog GuardDog	`guarddog:latest`	`311 MiB`	✓

Table 4: Descriptives of the generated Docker images

3.3 Running the analysis

We generate batch files for each downloaded archive in the datasets. For the tools Bandit4Mal, OSSGadget and the PyPI rules we do separate scans of the entire package and the setup-file from the package. Malicious software often abuses the setup-file as this is guaranteed to be executed on the installer's host when they install a package from source [12]. GuardDog does not allow scanning single files from a package, so is omitted from the analysis on setup-file basis.

We then run the batch-files using GNU Parallel [13] and write the output from each scan to a file for later processing. Each instance that is spawned is limited to 4 GiB of RAM and 1 CPU core, as we ran into a memory leak in the OSSGadget tool when scanning.

3.4 Analysis of results

In the final step of the experiment we process each job output and add it to a database, including the rule hits ranging from low to high severity and low to high confidence, based on what the tool

reports. Not all the tools report this in the same scale, the mapping is shown in Table 5. GuardDog does not provide any severity or confidence indicators per rule, those are all considered as high severity and confidence.

Tool	Confidence			Severity		
Bandit4Mal	Low	Medium	High	Low	Medium	High
OSSGadget	Low	Medium	High	Moderate	Important	Critical
PyPI	×	×	∗	Indeterminate	×	Threat
GuardDog	×	×	∗	×	×	∗

Table 5: Confidence reporting per tool

For the analysis we examine the hit-rate of the different tools on malicious and benign (random and popular) packages. Based on whether the hits occur more often on malicious packages than on popular packages we assess if the tool is useful.

Finally, we test the performance of the scanners using a a receiver operating characteristic (ROC) curve [5]. This shows if the scanner is useful as a classifier for malicious packages by comparing it to random output. A scanner can be seen as a classifier that classifies a package as malicious if it breaks one or more rules. An optimal classifier would have a true positive rate of 1.0 and a false positive rate of 0, and be located in the top left of the graph of the ROC curve. Then all malicious packages would trigger at least one rule, and all benign packages would trigger none. A completely opposite classifier would have a false positive rate of 1.0 and a true positive rate of 0.0, and be located in the bottom right corner. In this case the classifier perfectly classifies all packages, but with the completely opposite type of packages. Malicious would be marked as non malicious, and non malicious would be marked as malicious. A completely random classifier would be located along the diagonal, where there are exactly as many malicious as non-malicious packages are marked as malicious.

4 Results

In this section we prseent the outputs of the experiment. First, we describe the obtained packages from the datasets, whether data was omitted and the overlap between the datasets. Then, we present the runtimes of the different tools and review the total scans that the tools did. Finally, we discuss the performance metrics of the tools.

4.1 Obtained Packages

In Table 6 the counts of downloaded[1] packages are shown. Note that in the popular packages some packages were discarded due to not providing the source tarball of the latest version. The same was the case for the random dataset, however, there we were able to run the fetch operation again to collect at least 1000 packages. Even though the tools were run over each package in each dataset, in the analysis packages were deduplicated on their name and version. The final count of used packages is shown in Table 6.

4.2 Runtimes

The runtimes were collected and parsed from the GNU Parallel logs and are shown in Table 7. There are some notable omissions for Bandit4Mal and OSSGadget. Bandit4Mal got stuck on one package for 5 hours and OSSGadget at up all the memory within seconds. The Bandit4Mal project is not active anymore, a bug report for OSSGadget was filed[2].

[1] All data from the sources was last retrieved on Tuesday the 12th of December 2023 for a final analysis.

[2] `https://github.com/microsoft/OSSGadget/issues/452`

Dataset	Expected	Retrieved	Deduplicated
Backstabbers	3024	3024	2778
MalOSS	608	579	584
MALRegistry	2951	2951	21
DataDog	1397	1397	798
Top 1000 Downloaded	1000	1000	414
Top 1000 Dependencies	1000	951	951
1000 Random	1000	1390	1384

Table 6: Packages downloaded from dataset

		Bandit4Mal	OSSGadget	PyPI	GuardDog
*.py	Average	4.632s	3.592s	1.893s	10.010s
	Max	92.634s	90.769s	9.831s	9.953s
	Min	0.636s	1.305s	0.671s	5.865s
setup.py	Average	1.935s	2.647s	1.720s	×
	Max	8.742s	9.022s	8.872s	×
	Min	0.566s	0.810s	0.612s	×

Table 7: Runtimes per tool and scope

4.3 Classification Performance

To analyze how well the scanners can discern malicious packages from benign packages, Table 8 shows the percentage of packages that trigger at least one rule or more. For each package, this is split up into a scan of the entire package, and a scan of only the `setup.py` file. A scan of only `setup.py` might be able to give a more accurate result, as the chance of benign packages triggering an alert in the `setup.py` file might be lower in comparison to the `setup.py` files of the malicious packages. The `setup.py` file gives malicious actors the opportunity to run code upon installing the package. There were no scan results for only the `setup.py` file for the scanner GuardDog, as it does not support single file scanning.

Tool	PyPI		OSS Detect Backdoor		Bandit4Mal		GuardDog	
Dataset	setup	*.py	setup	*.py	setup	*.py	setup	*.py
Malicious	0.0%	27.5%	33.1%	91.9%	23.3%	42.8%	-	20.8%
Benign (popular)	0.0%	87.4%	21.4%	98.3%	48.0%	87.8%	-	7.6%
Benign (random)	0.0%	45.7%	9.4%	82.9%	32.9%	66.6%	-	2.1%

Table 8: Packages with at least one alert, by tool (%)

The table shows that for all scanners but GuardDog, the percentage of popular packages breaking one rule or more is higher than for the malicious packages. For PyPI and Bandit4Mal, this is also true for random packages. The GuardDog scanner alerts more often on malicious packages than benign packages, but has a much lower percentage of malicious packages breaking a rule. So it's alerts are better to discern malicious packages from benign, but trigger less often overall.

By looking at the alerts triggered when only scanning the `setup.py` file, OSS Detect Backdoor triggers alerts much more often for malicious packages in comparison to benign packages, but alerts less often overall. For Bandit4Mal, the alert rate is lower overall, but the proportion of malicious packages triggering alerts in comparison to benign packages does not change much. The PyPI scanner did not trigger any alerts for `setup.py` files.

In Table 9 the data is filtered to only contain alerts that are marked as having a high severity by the scanner. The GuardDog scanner does not differ in severity for it's alerts, so for comparison purposes, all rules are considered high severity. The OSS Detect Backdoor scanner supports high severity alerts, but apparently does not contain many of them. The PyPI scanner seems to not

contain any high severity rules that trigger in the `setup.py` file. For the PyPI and Bandit4Mal, filtering on high severity does not improve the rate of packages breaking alerts between malicious and popular packages. The random packages however, are by both scanners now marked less often as potentially malicous, because the number of random packages triggering at least one (high severity) alert is a lot lower then in the previous table.

Tool	PyPI		OSS Detect Backdoor		Bandit4Mal		GuardDog	
Dataset	setup	*.py	setup	*.py	setup	*.py	setup	*.py
Malicious	0.0%	9.9%	0.0%	0.1%	17.0%	37.2%	-	20.8%
Benign (popular)	0.0%	33.8%	0.0%	0.0%	24.5%	73.4%	-	7.6%
Benign (random)	0.0%	10.2%	0.0%	0.0%	10.2%	45.3%	-	2.1%

Table 9: Packages with at least one high severity alert, by tool (%)

4.4 Multiple rule threshold

In this section the classification performance is analyzed by increasing the threshold of triggered alerts for marking a package as potentially malicious. In Figure 2 a visualization is shown where you can see the rate of packages breaking at least one, two, three, four or five rules. This is again also split in scanning the entire package and scanning only the `setup.py` file.

It is remarkable that for all scanners, the malicious packages break the threshold of alerts less often than popular packages, except for GuardDog. For most of the graphs, the malicious packages trigger less alerts than random packages. Scanning only the `setup.py` file does not really improve this difference. For OSSGadget (OSS Detect Backdoor), malicious packages trigger more alerts for only scanning `setup.py` than other packages, but only for a threshold of one alert. This indicates that most malicious packages only trigger one OSS Detect Backdoor rule, and not more.

Increasing the threshold of alerts does not change the proportions of packages that can be marked as potentially malicious. For Bandit4Mal, increasing the threshold does lower the percentage of random packages that get flagged, while the percentage of popular and malicious packages getting flagged remains approximately the same. It is also noticeable that for the three scanners used in the previous research, triggering a single alert usually indicates that many other alerts will get triggered as well. This might indicate that those packages that triggers an alert either contain a pattern or some code that will repeatedly trigger security alerts everywhere.

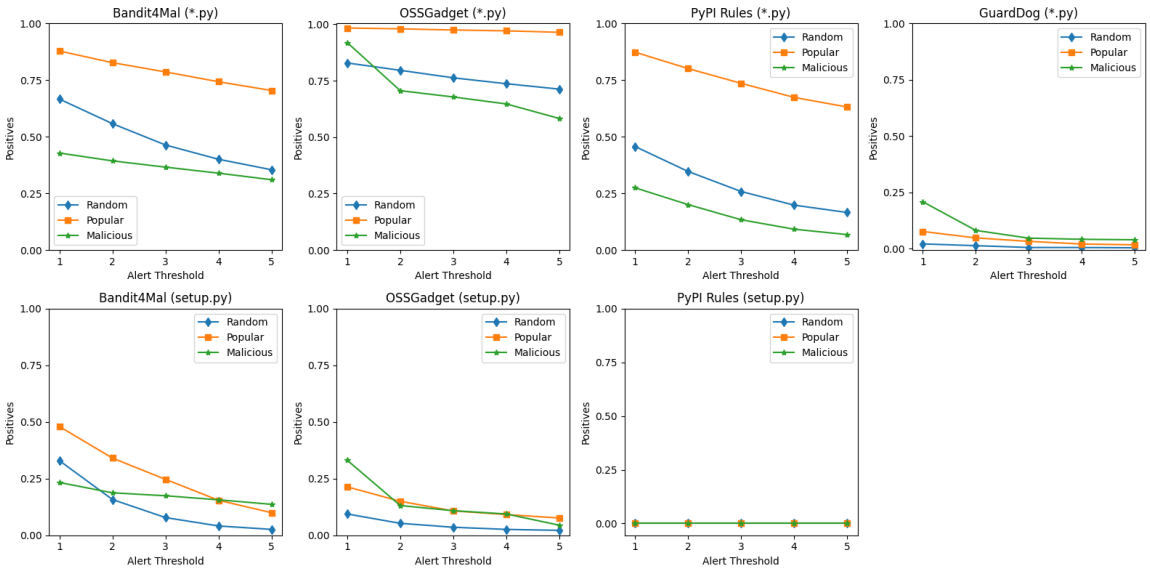

Figure 2: Scanning results with all alerts enabled.

In Figure 3, the same visualization as Figure 2 is shown, filtered on only showing alerts of the highest severity. All of the graphs show roughly the same pattern as in Figure 2. The only exceptions are PyPI, which did not trigger any high severity rules within the `setup.py` file, and OSS Detect Backdoor, which triggered very few rules of the highest severity level.

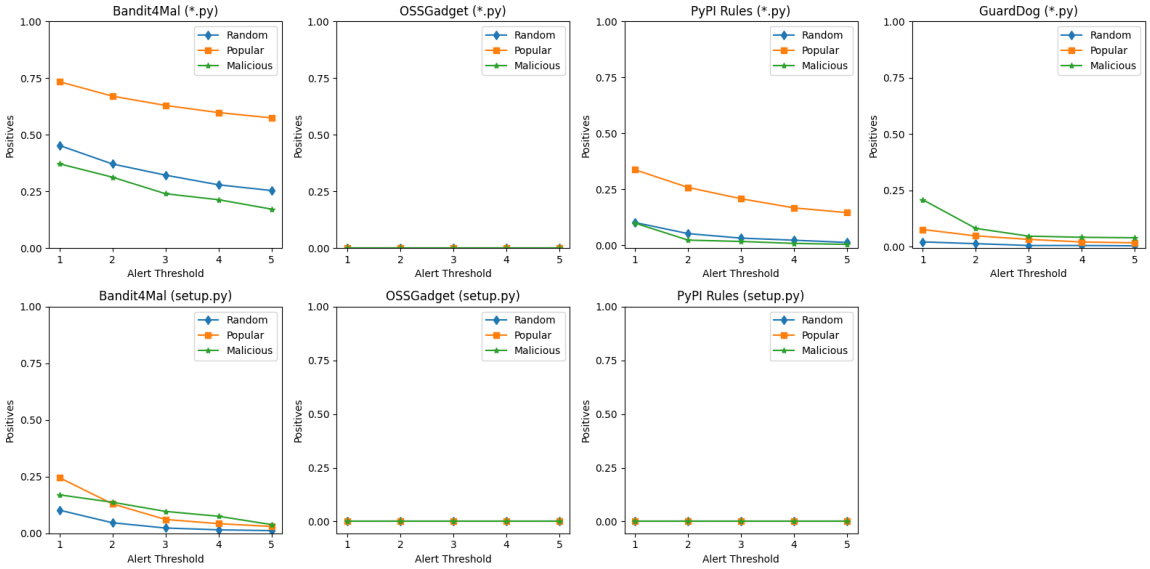

Figure 3: Scanning results with only high-severity alerts enabled.

4.5 ROC Curves

To show how well the scanners would perform as a classifier for malicious packages, we show four ROC curves. In these curves, a package that triggers one alert or more is considered a positive. The true positive rate therefore indicates how many known malicious packages trigger on or more of the scanner's alerts, and the false positive rate indicates how many benign packages trigger one or more alerts.

In Figure 4 the ROC curves for all four scanners are plotted, both scanning the entire package and only the `setup.py`. As expected based on the previous tables, only GuardDog is above the random classifier line for scanning the entire package, as all other scanners will trigger alerts more often for popular packages than for benign packages. The OSS Detect Backdoor scanner performs approximately equal to a random classifier, indicating that it's use as a scanner to detect malicious packages when scanning the entire package is minimal. The PyPI scanner and Bandit4Mal are not equal to a random classifier, however this is mainly due to the false positive rate. This indicates that you should be more suspicious of packages that do not trigger any alerts for these scanners than of those who do.

When only scanning the `setup.py` file, the OSS Detect Backdoor scanner performs comparatively well, while the PyPI scanner is now performing almost exactly as a random classifier. Overall however, the performance of the scanners is not more effective in comparison to scanning the entire package.

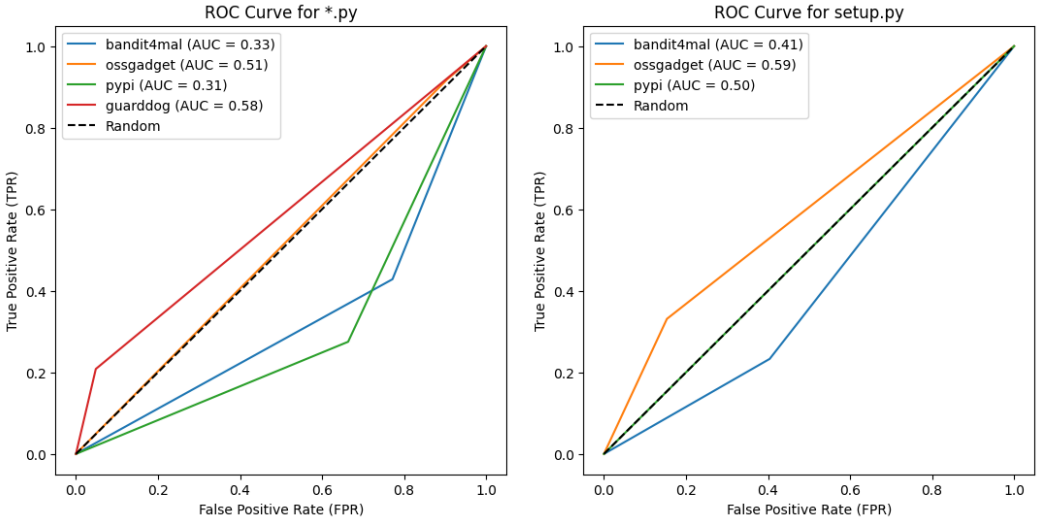

Figure 4: ROC Curves for scanning results.

In Figure 5 the same data is shown as in the previous figure, filtered on only showing alerts or rules of the highest severity. As GuardDog does not differ in severity of it's alerts, it's ROC curve is the same as in the previous figure. For the other scanners, the performance is more similar to a random classifier than in the previous graph, when scanning the entire package.

When only considering high severity alerts in scans of only the `setup.py` file, all of the scanners perform approximately as well as a random classifier. This can be explained by the fact that both OSS Detect Backdoor and the PyPI scanner did not trigger any high severity alerts during the scan of the `setup.py` file, as can be seen in Table 9. The Bandit4Mal scanner does however trigger high severity alerts during the scan of the `setup.py` file. Unfortunately the average of the percentage of packages triggering at least one high severity rule is approximately the same for malicious packages and the aggregated popular and random packages, which are both classified as benign.

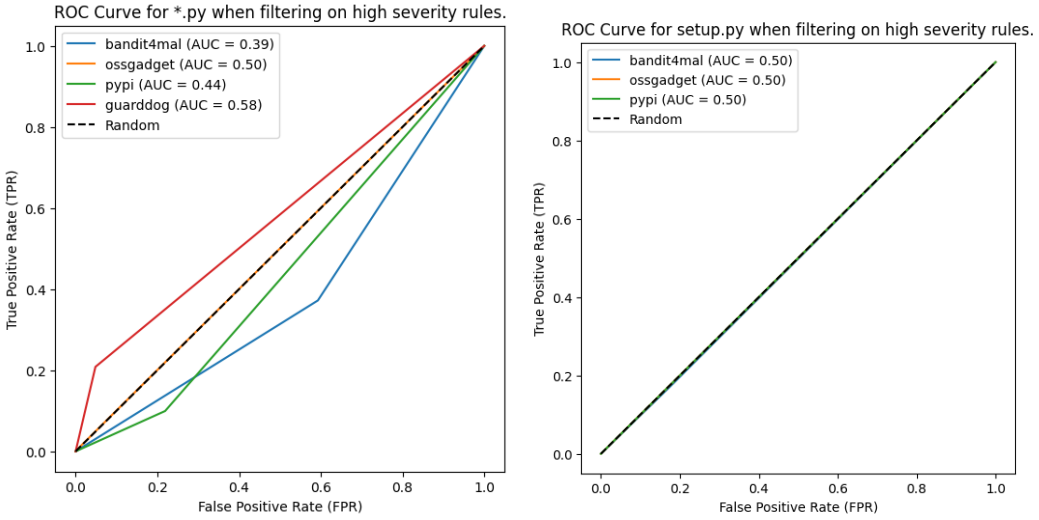

Figure 5: ROC Curves for scanning results, with filtering on high severity alerts.

5 Discussion

The results that were obtained are similar to the previous research. There are some differences, possibly due to using updated scanners and using a larger dataset. The random dataset is different as it is sampled again. However, as the dataset is random, the differences should be similar to changes in the population. The popular packages and their dependencies were also obtained from the PyPI repository again with both the latest versions and the newest top 1000.

5.1 Comparison to previous research

The previous research stated that scanners catch more than half of malicious packages, when only scanning `setup.py`. Our results did not find this to be true, neither in the case of only scanning `setup.py` nor when scanning all Python files within the package. The only scanner that caught more than 50% was OSS Detect Backdoor, mainly due to the very large false positive rate.

False positive rates were very high for the previous research, and were also high in our results. The number of alerts triggered for benign packages was in most cases higher than for malicious packages. This was both true when scanning all Python files as well as scanning only the `setup.py` file or when filtering for high severity alerts. The only scanner where this was not the case is the added scanner, GuardDog. The false positive rate was comparatively low, and lower than the true positive rate. The true positive rate however was not very high, with only triggering an alert for less than a quarter of the malicious packages.

Another finding of the previous research was the fact that a positive package usually had many alerts, which can also be found true for our results. The number of packages that conformed to the threshold rate of alerts did not decrease much when increasing the threshold. GuardDog is also an exception to this rule, as it performed best with a threshold of 1 rule, and after that not many packages are found to be potentially malicious.

In confirmation of the previous research, increasing the threshold did not do much to decrease false positive rates, while the true positive rate also dropped. Proportionally the composition of malicious packages and benign packages did not change when increasing the alert threshold, while the overall hit rate dropped minimally.

In the previous research they authors stated all tools ran within a reasonable amount of time, which we confirmed. However, the Bandit4Mal scanner timed out on a package, where the limit was five hours. The OSS Detect Backdoor scanner had an issue when scanning a particular package of which we notified the authors (who quickly confirmed the bug).

5.2 Additions to previous research

Our additions to the replication study were the inclusion of a bigger malicious dataset, including a new scanner (GuardDog), studying the effect of filtering for high severity alerts, and studying the effectiveness of using the scanners as classifiers for malicious packages.

The previous research found a larger proportion of malicious packages triggering an alert. In our results in Table 8 the hit rates were not as high. This can be due to the fact that the original malicious packages dataset was much smaller. Moreover, the two new datasets of malicious packages might have different rules about considering a package malicious or the creators might have been more thorough in their analysis of potential malicious packages. This might cause more packages to be included that are harder to detect for the scanners.

The GuardDog scanner had the highest true positive rate in comparison to the false positive rate of all scanners when scanning the entire package. This led to it performing the best of all four scanners as a classifier as shown in the ROC curves. The downside of GuardDog is that while the classifying performance is higher, the true positive rate is under 25%. So more than three quarters of all malicious packages will not be detected.

The other three scanners performed almost the same as a random classifier, or had a higher false positive rate than true positive rate. This makes the scanners very unreliable to use as classifiers to detect potential malicious packages. The scanners can mostly be useful to detect glaring mistakes or

possible vulnerabilities in your own code, but do not indicate much about whether a package might be malicious or not.

The popular packages were usually the group of packages that had the highest percentage of triggered rules. This could also be caused by the fact that popular packages might on average contain more code, which could give more opportunity for triggering alerts. A possible improvement is to correct for this when classifying packages, or by excluding popular packages from the metrics, and only comparing classifying performance of random packages to malicious packages, as malicious packages are unlikely to be popular or be similar to popular packages. Except for packages that are malicious due to typosquatting. Excluding popular packages will not however get the classifying performance above the random classifier performance for PyPI, Bandit4Mal and OSS Detect backdoor, as their alerts are also more often triggered by random packages than by malicious packages, as shown in the first two graphs with alert thresholds.

Filtering on only high severity alerts did not result in much performance improvement for any of the scanners. It mostly lowered the overall alert count, but did not change the proportions in which malcious, random and popular packages were triggering alerts. The ROC curves also showed a decrease in classifying performance when filtering for high severity alerts. All scanners performed the same or closer to a random classifier when filtering for high severity alerts. The only scanner for which filtering on high severity rules might be helpful is OSS Detect Backdoor, as it's detection rate when scanning `setup.py` is higher for malicious packages than for random packages. Popular packages however do have a higher hit rate, but when scanning for potential malicious packages you could filter out popular packages by looking up their download count.

5.3 Recommendations

The performance of the scanners in discerning malicious packages from benign packages does not suffice. Most of the classifiers did not even achieve a higher true positive rate than false positive rate, even when limiting the scan to `setup.py` or when filtering on high severity rules. The development of these tools might be improved by benchmarking the tools more often on known malicious datasets and comparing it to datasets of benign packages. Such a data-driven approach could help authors of the tools create more effective rules that focus on discerning malicious from benign packages.

To help authors of scanners, it is necessary to obtain and maintain good quality datasets of malicious packages. It might also be beneficial to either correct for the size of popular packages or to filter them out of the benign dataset. Malicious packages are unlikely to be similar to popular packages in structure, making the comparison less useful, while obfuscating the number of alerts triggered by more comparable random packages, due to their possible larger code size.

6 Conclusion

Our findings are largely in line with the conclusions of the previous research. By using a larger combined dataset of malicious packages, and by including GuardDog as a scanner, the research is improved. The additions might have caused a lower detection rate of malicious packages for the scanners and we deem none of the scanners appropriate to use as a trustworthy classifier for detecting malicious packages. All but one has higher false positive rates than true positive rates. Filtering for only high severity alerts does not improve the performance. Neither does limiting scans to the `setup.py` file of a package. As security researchers find more malicious packages and report these to the authorities, a data-driven approach can help creating better rules, improving the performance of security scanners.

References

[1] Python Packaging Authority. PyPI YARA Rules. https://github.com/pypi/warehouse/blob/04aa8003d4f18a76e05ee61ad5afd15928df1ccc/warehouse/malware/checks/setup_patterns/setup_py_rules.yara, 2020. Removed from main branch in May 2023.

[2] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*, February 2021.

[3] Ly Vu Duc, Zack Newman, and John Speed Meyers. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. https://doi.org/10.5281/zenodo.7578941, January 2023.

[4] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem, 2023.

[5] Zhe Hui Hoo, Jane Candlish, and Dawn Teare. What is an roc curve? *Emergency Medicine Journal*, 34(6):357–359, 2017.

[6] Datadog Security Labs. Guarddog. https://github.com/DataDog/guarddog.

[7] Datadog Security Labs. https://github.com/datadog/malicious-software-packages-dataset, March 2023.

[8] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[9] Microsoft. OSS Gadget. https://github.com/microsoft/OSSGadget, 2023. Downloaded: v0.1.414.

[10] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020.

[11] PyCQA. Bandit: A tool for finding security issues in python code. https://github.com/PyCQA/bandit, 2023.

[12] Python Packaging Authority. Python Packaging User Guide - Binary Distribution Format (Wheel). https://packaging.python.org/en/latest/specifications/binary-distribution-format/, 2023.

[13] Ole Tange. Gnu parallel 20210822 ('kabul'). https://doi.org/10.5281/zenodo.5233953, August 2021.

[14] The gVisor Authors. gVisor Documentation. https://gvisor.dev/docs/, 2023.

[15] Tidelift. Libraries.io Open Source Repository and Dependency Metadata. https://libraries.io/search?order=desc&platforms=PyPI&sort=dependents_count&page=1, December 2023.

[16] Hugo van Kemenade, Martin Thoma, Richard Si, and Zsolt Dollenstein. hugovk/top-pypi-packages: Release 2023.12. https://doi.org/10.5281/zenodo.10245206, December 2023.

[17] D.-L. Vu. A fork of bandit tool with patterns to identifying malicious python code. https://github.com/lyvd/bandit4mal, 2020.

[18] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. Bad snakes: Understanding and improving python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 499–511, 2023.