Response to detailed comments

Note: This document provides a point-by-point response to the reviewers' detailed comments. The text in light blue represents the reviewer's comments, while the ***black text*** *is our response. All the content that requires updates such as the additional experimental results and analysis, the content suggested by the reviewer et.al, will be included in the next version of our paper.*

Reviewer A: Detailed comments for authors

General Comments:

Overall, this is interesting work but I feel it needs a major revision to address concerns, I would like to see an ablation study to see how important or effective different components are of the whole UPiMPP design. There are three faucets of the selections phase. Which one is more effective in finding more code confusion attacks?

Re: We appreciate the reviewer's comment. We have conducted an ablation experiment to evaluate the performance of UPMiPP in detecting code confusion attacks across 149 cases, after removing the name, submodule, and readme respectively. The results are 141 (-8), 149 (-0), and 99 (-50), respectively. It is evident that the most significant performance decline occurs when the readme is removed, followed by the removal of the name. These results suggest that the readme is the most effective and important facet for detecting code confusion attacks, followed by the name, and then the submodule.

UPMiPP uses package name, metadata, and code, all three type of package information to facilitate the analysis.

In-the-wild analysis is a strength for this paper.

Re: We appreciate the reviewer's positive feedback.

Authors have mentioned that they will make the found malicious packages public upon acceptance of the paper which should be appreciated for fostering open science policy.

Re: We appreciate the reviewer's positive feedback. We have made the discovered in-the-wild malicious packages available at https://anonymous.4open.science/r/21872632513/.

Originality: The authors define a new type of attack, code confusion attack, and present a system for detecting this attack.

Re: We appreciate the reviewer's positive feedback.

Importance of contribution: Vulnerability/attack detection is an important issue.

Re: We appreciate the reviewer's positive feedback.

Soundness: The approach is mostly sound, though the assumptions are not justified. The system does a nice job of working with the pre-defined assumptions – but are these assumptions the correct ones?

Re: We appreciate the reviewer's comments. We apologize that the rationale behind our assumptions was not clearly presented in the paper. Our assumptions are indeed valid and correct, as we have real-world cases to support them. Please refer to the detailed responses below regarding the detailed comments on the paper's content.

Evaluation: Performed an evaluation against other methods.

Re: We appreciate the reviewer's positive feedback.

Quality of presentation: I understood most of the parts of the paper, but the overall writing is in 'okay' English and could be improved with more editing.

Re: We appreciate the reviewer's comments. We will carefully refine the presentation throughout the paper to improve its readability and clarity. Additionally, we will invite English professionals to polish the content and presentation details of the paper.

Appropriate comparison to related work: The authors did compare their tool to other tools.

Re: We appreciate the reviewer's positive feedback.

Detailed Comments:

Abstract: The abstract is not clear, which causes it to not be compelling and an invitation for the paper. "Code confusion attack" should be more clearly defined before moving into evasion techniques. How can a code confusion attack be precisely defined – is it when a low % of the code is the malicious code? Something about the package name? How can a code confusion attack be precisely classified with a repeatable process?

Re: We appreciate the reviewer's comments. We feel sorry for not providing a clear definition and explanation of "code confusion attack" in the abstract. In our paper, a code confusion attack is defined as the manipulation of an existing project by injecting malicious code and then uploading the modified package on PyPI, where the code similarity between the malicious one and the original project exceeds 90%. This definition is reproducible and can be used to precisely classify whether a malicious package is a code confusion attack.

I'm unsure about the "candidate projects" versus the "true targets" (lines 14-15). Also in line 15 – what is "discrepancy code"? Perhaps go into less detail in the abstract and stick to the high level concepts. Line 19 – is a "disclosed code confusion attack" a case of malicious code already being identified in the PyPI dataset? Were they already classified as "code confusion attacks" in the dataset?

Re: We appreciate the reviewer's comments. The explanations of these terms are as follows:

"candidate projects" refer to the list of projects generated by UPMiPP during the selection phase.

"true targets" denote the original projects that malicious packages, if they are code confusion attack, has been tampered with.

"discrepancy code" refers to the newly introduced code in a package which is not in the original project.

"disclosed code confusion attack" refers to cases identified by UPMiPP from historical malicious packages that meet the criteria of code confusion attack. Due to the lack of dataset for code confusion malicious packages, we manually verified these cases and confirmed that they align with the definition of code confusion attack. These cases were used as benchmarks for evaluating RQ2 and RQ3.

Line 21-22: searching for code confusion attack in-the-wild is a strength of the paper.

Re: We appreciate the reviewer's positive feedback.

Introduction

Is it worth all the names – code confusion, typosquatting, combosquatting, package confusion, name confusion – if so don't rely on the reader to understand the precise distinction between them. Provide a small background section with definitions so we can understand what you are doing versus the others.

Re: We appreciate the reviewer's comments. We provide the following definitions for these terms:

- Code confusion attack: A malicious package is crafted from an existing project by injecting malicious code and then uploading the modified package on PyPI, where the code similarity between the malicious one and the original project exceeds 90%.
- Typosquatting: An attack where the attacker takes advantage of potential user input errors by creating a package name similar to that of a pupular package, luring developers or users into accidentally installing a malicious package.
- Combosquatting: An attack where the attacker creates misleading package names by adding words or phrases to popular package names, deceiving developers or users into mistakenly installing a malicious package.
- Package confusion: An attack exploits the vulnerabilities or features in the package management system's naming and dependency resolution mechanisms, causing developers or systems to use a malicious package instead of a legitimate or expected one.
- Name confusion: In this paper, this term refers to malicious packages that are created by imitating the names of popular packages.

In the next version of our paper, we will refine the presentation in the introduction section to reduce these terms to improve readability and understanding, and add a separate "small background" section to provide detailed explanations of these names.

Line 52-56: How did you come up with this observation? Is it purely empirical? Or have other previous studies mentioned this fact that 'However, ...'? [line 53] Also I do not fully understand what this sentence means. Since this is a very important line about what the problem is, I believe this part should be crystal clear. First line says most

malicious packages contain only small amount of malicious code, which is easy to detect. The second line starts with 'However' -- so I am expecting something different than the first line -- and says more sophisticated malicious packages contain only minimal amount of malicious code newly introduced. If the newly introduced malicious code is minimal, according to the first line, shouldn't it be easy to detect the malicious code using previous approaches? At this point, I am not totally clear what **code confusion attack** means.

Line 58-60: Similar comments as above. I fail to see the difference between a typical malicious package and packages targeted by code confusion attack. Also no citation provided, so I have no way to understand if this motivation is coming from previous work or unsubstantiated observation of the authors.

Re: Our observations, derived from manually examining the historical malicious packages included in BKC, are as follows: the majority of malicious packages contain code that is entirely used for malicious purposes, which we refer to as simple malicious packages. At the same time, a significant number of malicious packages are created by modifying existing projects to inject malicious code, where the majority of the code is benign, which we refer to as code confusion malicious packages. Through our observations, we found that the portion of the code responsible for malicious behaviors is usually small, which is also investigated by the previous study [9]. The reason why code confusion malicious packages are more difficult to detect is that the injected malicious code is typically mixed with large benign code, making it challenging for existing detectors to effectively distinguish them from the legitimate projects they target. Please refer to Example 1) in Section 2.2 for further details.

We apologize for the confusion caused to the readers in Line 52-56. We will improve the expression in the next version of the paper.

Line 61: Proper citation needed here for this research gap. Also the authors could have added one line for justification how previous methods ignore this type of specific malicious package (maybe they did in the last previous paragraph but it was not clear).

We will add the following citations after the sentence in line 61: [23,33,38,18,20]. What

we intend to convey is that existing methods face challenges in detecting code confusion attacks, as they are not specifically designed to address this type of malicious package. In contrast, our method recognizes this issue and is capable of effectively detecting code confusion attacks.

Line 67: I disagree that package confusion detection methods are ineffective to discover *true target projects*. The way *package confusion* works is that they compare the name of newly published packages or packages with lower download count to check if that package is trying to mimic the name of any highly downloaded packages by using different rules. A quick scan through [15] shows:

- pandar (which was squatting on pandas) Removed from PyPI.
- prompt-tool-kit (which is squatting on prompt-toolkit) Reported to maintainer.
- requestsaa (which is squatting on requests) Reported to maintainer.

Re: Since our goal is to determine whether malicious behavior is present in the discrepancy code, package confusion detection method has two drawbacks: 1) it lacks checks at the code level. In practice, developers may intentionally upload packages with names similar to popular ones to serve as placeholders, so detecting only based on naming is insufficient; 2) In scenarios of inconsistent confusion, the target project identified by the package confusion detection method may not be the actual project(true target project) compromised by the malicious one, which hinders subsequent localization of the discrepancy code.

Is code confusion – typosquatting/package confusion + a small amount of malicious code injected into an existing code base? Both of these must occur at the same time? (The paper has confused me.)

Re: A code confusion attack is a malicious package created by an attacker who borrows an existing project and injects malicious code into it. It does not require the simultaneous fulfillment of the conditions of typosquatting/package confusion + a small amount of malicious code.

Anecdotally, I've heard that in a lot of typosquatting/package confusion attacks, the code is very different from each other – it's not just a small amount of malicious code in the middle of an existing project. I heard that in a presentation by Sonotype (on their State of the Supply Chain report) so I don't have a reference for you, but I challenge you to substantiate the prevalence of code confusion attacks whereby there is a lot of code with a small amount of malicious code added.

Re: As demonstrated in RQ1, among over 3,000 historical malicious packages, 149 were created by injecting malicious code into existing projects. In all these 149 malicious packages, over 90% of the code originates from the existing projects, with the malicious code accounting for only a small portion, and the longest containing only 62 lines. Additionally, our method discovered 27 real-world code confusion attacks, all of which were confirmed and removed by PyPI.

Line 101-103: What is the full form of UPMiPP? The first time you mention a short form, you should mention the full form. Disregard this comment if this is not an abbreviation/acronym.

Re: UPMiPP is derived from the title of the paper: Unveiling Potential Malicious Behaviors with Identifying Code Confusion of PyPI Packages.

Line 102 – you need to provide reference for this "widely used dataset"

Re: With the inclusion of the reference, the sentence will be revised to:"We evaluated UPMiPP on a widely used dataset [25] for malicious packages."

Line 111-115: First two contributions could be merged into one since they basically echo each other. Also the phrase "inconsistent confusion" is not compelling – which is revealing inconsistent confusion a contribution? Would consistent confusion be better? (You define consistent and inconsistent confusion in Section 2, but it's too late. I also think the names (consistent and inconsistent) are not intention revealing – they are confusing)

Re: We appreciate the reviewer's comments. After careful consideration, we believe

that the first two contributions are suitable for merging. Regarding "inconsistent confusion," we believe that revealing it can help in the design of our method. Relying solely on name-based checks cannot identify the true targets of inconsistent code confusion attacks. As you can see, our method's selection phase considers not only the name but also integrates the submodules and Readme to select candidate projects. We do acknowledge that we should provide a more clear explanation of these names(consistent and inconsistent) to better help understand our motivations.

Section 2: Background and Motivation

Line 128: (Section 2.1) I like the effort to discuss different type of attack timing for malicious code and their occurrence frequency. Is the information in Table 1 from [9]? You should make that clear. How was the occurrence frequency and detection difficulty determined scientifically?

Re: We appreciate the reviewer's comments. The content in Table 1 is empirically derived from experiments conducted on samples from BKC. As shown in Fig. 5, the distribution of frequency follows the order: Install-time > Import-time > Invoke-time. In terms of the number of affected source code files, the order is Install-time < Import-time < Invoke-time. So the scope of detected code files also increases, leading to a corresponding increase in detection difficulty.

Line 168: 'jeIlyfish' should be written as 'jeilyfish' similar to Fig.1 if that is the correct name.

Re: 'jeilyfish' in Fig.1 is wrong. It should be 'jeIlyfish' consistent with the text.

Line 176: It would be nice to have the naming convention in one place. *Code confusion*, *name confusion*, *inconsistent confusion* and *consistent confusion* used by the paper (and also *package confusion* which is referred to as *name confusion* in this paper).

Re: We appreciate the reviewer's comments. In the next version of the paper, we will dedicate a separate section to explain the definitions of these names.

1. The UPMiPP Framework

Line 231-233: There is no justification given why such inclusion criteria have been used or no previous work has mentioned that influenced criteria (1). With these two criteria, the EP will have some packages that have only one version released more than 2 years ago. Comparing to the previous package confusion (or name confusion) attacks where packages are only compared against popular counterparts, code confusion will have much higher number of packages to compare against (essentialy, O(N^2)).

Line 235-237: This line threw me off. What was the purpose of criteria (2)? Is it to remove malicious packages from EP? Also, is this the criteria for your paper or is this the criteria that should always be used by UPMiPP in the future?

Re: We apologize for the confusion caused by this expression. Since there has been no prior work proposing selection criteria specifically for the code confusion attack scenario, we construct these criteria ourselves. Many existing methods only compare popular projects, which is quite limited. In the context of code confusion attacks, attackers may also borrow less popular projects to create malicious packages. Therefore, we expand the selection scope to include projects with releases, resulting in Criterion(1). Although we included more projects for comparison, we achieved good execution efficiency through carefully designed methods, as demonstrated in Section 4.6. Regarding Criterion(2), it aims to exclude malicious packages that may already exist, as some malicious packages might remain on PyPI for a period of time. This criterion can continue to be used by UPMiPP, which can update the EP (PyPI known projects) based on the time point when the tool is used in future applications.

266-268 and 269-270– Can you substantiate this "observation from real world samples"? This is a basic assumption/design point of your system which may be false.

Re: As demonstrated in RQ1, we identified 149 real-world code confusion attack cases. These malicious packages were created by attackers who injected a small amount of malicious code into existing projects, making them appear identical to the original projects. This supports our point: *we found that attackers typically hide malicious code*

within known projects and render their meticulously crafted one seemingly unexceptional(Line 266-268). Additionally, through manual observation, we found that the majority of these 149 cases retained the original project's module structure and files, which supports our point: *attackers typically make sub-module names identical to target project names and retain README file from the target project to exhibit an air of legitimacy(Line 269-270).*

Line 278-280: Authors have claimed that they introduced three additional detection techniques, name combination, additional_prefix/suffix, and removed_prefix/suffix. I disagree. The additional_prefix/suffix and removed_prefix/suffix simply fall into **Prefix Suffix Augmentation** from Typomind.

Re: We acknowledge that these three naming rules overlap with those in Typomind. We will remove them in the next version of the paper.

Line 281: Quotation marks are not correctly used.

Re: We will change ’-’, ’_’, and ’.’ to '-', '_', and '.' .

Line 284: How did the authors come up with the common used affixes and strings? It was not mentioned in this passage.

Line 296: What was the process/methodology of finding the **common** submodule names? How are you claiming these are common?

Re: We manually inspected many sub-module names of benign projects and found that a lot of these names are frequently used, making it difficult for such commonly used sub-module names to serve as targets for code confusion. Therefore, we summarized the commonly occurring sub-module names to form an exclusion list.

Line 292-301: One interesting idea would be to check if the sub-module names have *name confusion* attacks or not using the previous methodology (step one of selection phase). My hypothesis is only package names are not the target of *name confusion* attack, rather the package name and also all submodule names as well.

Re: We greatly appreciate the reviewer's valuable idea. Submodule names could indeed become targets for name confusion attacks. In the future, we will extend the name confusion detection technique to include submodule names in the selection phase of our methodology.

Line 327-328: Why did you set the threshold of k to 10 and similarity value 0.8? I do understand this is to remove the packages with higher distance in the vector space (to reduce the complexity from O(M x N^2)). But citing previous papers who followed the similar score might be a good way to justify those empirical values.

Re: These two thresholds were determined through experimentation. Specifically, for each of the 149 code confusion attack cases, we calculated the Readme similarity and top k in the EP for the corresponding target project. We found that the similarity values were all above 0.8 and all ranked within the top 10. For detailed experimental data, please refer to: https://anonymous.4open.science/r/21872632513.

Line 352: Understanding the priority order and its justification is good but we need to know exactly what values were used for Wd, Ws, and Wn. Only the priority order does not provide the full picture. What if the priority is set to WnWn = 1, WsWs = 10000, and WnWn = 10^6?

Re: During the rebuttal period, we conducted an additional experiment. While maintaining the priority order $W_n < W_s < W_d$, we tested the rankings of the true target projects for the 149 code confusion attack cases at different values of W_n, W_s, and W_d. The final results showed no changes in the rankings. For detailed experimental results, please refer to: https://anonymous.4open.science/r/21872632513.

Line 368: (Section 3.2.1) I like the idea of interval release sampling to reduce the number of packages to compare with. But this step inherently assumes that developers always follow the SEMVER rules during version releases (e.g., PATCH release only contain bug fixes). In reality, developers do not always follow SEMVER [2, 3, 4]. This can definitely go to the threats to validity section.

Re: We appreciate the reviewer's valuable suggestion. We will discuss this scenario in the "Threats to Validity" section. In lines 393-395 of the paper, we explain the sampling strategy for cases where projects do not adhere to SEMVER: *If a project's version numbering does not conform to the semantic versioning norm, we sample versions at a step of 5% of the total number of its releases.*

Line 398: I did not understand the process followed to determine the 'code similarity measurement'. Did you check line by line? Did you check file by file? How does this process fare with code copied in a file from multiple sources? Or how does fare with a slightly modified copy of a file (e.g., changing the order of function definition or declaration)? How does this process different from simple diff tool?

Re: We apologize for not providing a detailed explanation of the 'code similarity measurement' process. The process of calculating the code similarity between two packages can be summarized as the following algorithm:

```
Input: Package P, Candidate package CP
Output: Code similarity value CSV
function Measure_code_similarity(P, CP):
    p_source_files ← get_all_code_files(P)
    cp_source_files ← get_all_code_files(CP)
    p_code_line_set ← ∅
    cp_code_line_set ← ∅
    for file in p_source_files:
        code_lines = read_code_lines(file) // read code lines into a set from files
        p_code_line_set = p_code_line_set ∪ code_lines
    for file in cp_source_files:
        code_lines = read_code_lines(file) // read code lines into a set from files
        cp_code_line_set = cp_code_line_set ∪ code_lines
    discrepancy_code ← p_code_line_set – cp_code_line_set
    N_p ← p_code_line_set.size()
    N_cp ← cp_code_line_set.size()
    N_dc ← discrepancy_code.size()
    code_similarity ← avg(N_dc / N_p,  N_dc / N_cp)
return code_similarity
```

We process each line of code in every code file within the package and store all the code lines from a package in a set for deduplication. We then extract the discrepancy

code by performing set difference operations. Even if code within a file is copied from multiple places, our algorithm handles it effectively. If only the order of code lines within a file is adjusted without changing the code itself, these lines will not be identified as discrepancy code. Compared to simple diff tool, our algorithm is more robust for handling modifications to file names or directory structures, as discussed in lines 425-431.

Line 419: (Section 3.2.2) I am also not clear how this 'discrepancy code extraction' works. Just saying '[...] that extracts all the code line from package p and directly judges their existence within tp' (in line 436) does not tell us how this step actually works. Also why did you use line number for comparison since adding one empty line or a comment at the beginning of a file will change the subsequent line numbers of that file?

Re: In our previous response, we add an algorithm to clearly demonstrate how 'discrepancy code extraction' works. We apologize for the confusion caused to the reviewers by line 439. In fact, we only retained the line numbers to identify the locations where the discrepancy code appears. The line numbers do not participate in the code similarity comparison.

Line 446: "through a survey of research works..." There is no methodology presented for doing this survey. Please provide the references.

Line 457: Similar comment as above. What was the process of finding those tools? How many did you start with at the beginning?

Re: We apologize for not presenting more details about the "survey" in the paper. We conducted a literature review, investigating studies related to malicious Python packages and referencing the tools evaluated in those studies [9][37].

Line 460: 'Technical details are clearly available' is a subjective comment. What are the criteria for determining if it is clear or not? Who is judging if this is clear or not?

Re: We appreciate the reviewer’s comments. The technical details were determined based on whether the tools are open-source, as we can exam source code to acquire

technical details.

Line 468: Citation missed for 'Bandit' tool.

Re: We will add the citation for the ‘Bandit’ tool in the next version of the paper. Bandit is an open-source tool designed to find common security issues in Python code. See https://github.com/PyCQA/bandit.

1. Evaluation

Line 491: (Section 4.1) BKC dataset was published in 2020 and we can safely assume that the dataset was collected even before that. There are a couple of new malicious package dataset which tries to combine malicious packages from multiple sources, e.g., MalwareBench [1]. It would be interesting to see how UPMiPP works in other datasets.

Re: Although the BKC dataset was initially released in 2020, it has been continuously updated with new malicious packages. As of the time of writing, the BKC dataset contains 3,014 malicious packages, which is several times larger than the 2020 version. Therefore, we believe that the BKC dataset is still a recent dataset.

We also actively took the reviewers’ suggestions into account and attempted to apply our method to MalwareBench for evaluation. However, accessing MalwareBench requires contacting the authors. We reached out to the authors but have not yet received a response. Nevertheless, we found that the samples in MalwareBench were collected from Backstabber’s Knife Collection (BKC) [1], Guo et al. [2], Datadog [3], and MALOSS [4]. Therefore, we evaluated our method on Guo et al., Datadog, and MALOSS. The experimental results show that there are 37, 35, and 14 cases of code confusion attacks on Guo et al., Datadog, and MALOSS, respectively. These results will be included in the next version of the paper.

[1] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber’s knife collection: A review of open source software supply chain attacks. (2020), 23–43.

[2] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. arXiv preprint

arXiv:2309.11021 (2023).
[3] Datadog Security Labs. 2023. Open-Source Dataset ofMalicious Software Packages. Retrieved December 2, 2023 from https://github.com/datadog/malicious-software-packages-dataset
[4] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. arXiv preprint arXiv:2002.01139 (2020).

Using the data in Section 4.2, you could substantiate some of your assumptions and provide more empirical evidence. You sort of did that in lines 565-587, but the presentation could be more thorough and systematic.

Re: We greatly appreciate the reviewer's valuable suggestion. The data in Section 4.2 strongly supports our assumptions empirically. We will improve the presentation in the next version of the paper to make it more thorough and systematic.

Can you compare the quantity of name confusion attack (name confusion without code similarity) with completely different code with the name confusion + code confusion/a smaller amount of malicious code?

Re: We appreciate the reviewer for providing a valuable direction. This requires additional experiments for evaluation, which will take some time. In the revision stage, we will include this comparison in the paper.

Line 703: When you identify packages as malicious, shouldn't you inform the PyPI registry about your findings and facilitate taking down those malicious packages? I do not know if keeping the information to yourselves is a good thing here. What is the point of in-the-wild analysis then? You aid the community in taking down malicious packages *and* the feedback from the registry can be considered validation of your work. You say (lie 704) that all the packages were removed from PyPI – how did that happen?

Re: We apologize for the confusion caused to the reviewer by this expression. Actually, the malicious packages we discovered were reported to PyPI, and they were

subsequently removed.

Line 878: (Section 8) Although authors have said that they are providing a replication package in the Zenodo record, **the Zenodo record does not contain the replication package rather the output of their replication package only.** Taking a Quick Look at it, it seems like the packages found as malicious from the BKC dataset were also provided with the source code.

Re: We make our code publicly available at https://anonymous.4open.science/r/21872632513 .

Related works should not just be a laundry list of papers rather it should mention how your work is different than the previous works. I understand that was mentioned in other sections of the paper. Still I think writing a one-line summary of differences of previous works is worth it. Also I do not need to know about other supply chain attacks -- but if you can mention what are the closely related ones and how they differ in related work section, that would be great.

Re: We appreciate the reviewer's suggestion. We commit to providing a detailed discussion of the relationship and differences between UPMiPP and other related works in the next version of the paper.

[1] Zahan, Nusrat, et al. "MalwareBench: Malware samples are not enough." 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR). IEEE, 2024.

[2] Raemaekers, Steven, Arie van Deursen, and Joost Visser. "Semantic versioning and impact of breaking changes in the Maven repository." Journal of Systems and Software 129 (2017): 140-158.

[3] Dietrich, Jens, et al. "Dependency versioning in the wild." 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 2019.

[4] Zhang, Lyuye, et al. "Has my release disobeyed semantic versioning? static detection based on semantic differencing." Proceedings of the 37th IEEE/ACM

International Conference on Automated Software Engineering. 2022.

Reviewer B: Detailed comments for authors

• While the first three sections of the paper are clearly written and well-organized, the clarity diminishes significantly from Section 3.3 onward, particularly affecting Section 4. This decrease in readability makes it challenging for readers to follow the methodology and evaluation, potentially detracting from the paper's overall impact. Improvements in clarity and structure in these sections would strengthen the presentation of the study's methodology and results.

Re: We appreciate the reviewer's valuable feedback. We will carefully improve Section 3.3 and the subsequent content to improve the clarity and help readers better understand our work.

• The paper lacks clarity in presenting its research objectives, as it introduces two research questions in the introduction but expands to five in the evaluation section. This inconsistency creates confusion about the study's focus and raises concerns about coherence in addressing the primary research goals. Ensuring alignment between sections would strengthen the paper's structure and improve readability.

Re: In the introduction, we introduced two questions, which summarize the challenges in detecting code confusion attacks. The two phases of our method were specifically designed to address the two challenges. In the evaluation section, we set up five research questions (RQs) to comprehensively assess the superiority of UPMiPP from multiple perspectives. This "inconsistency" actually pertains to different research angles.

• The paper's description of UPMiPP's integration of sourced tools lacks clarity, particularly in explaining the roles of referenced works like MPHunter, GuardDog, and others within the framework. While the authors list these tools for detecting malicious packages, they do not clearly outline how each specific tool's techniques contribute to

the UPMiPP framework. A more precise explanation of each method's purpose and relevance within UPMiPP would strengthen readers' understanding of the methodology and the study's contribution. This refinement would also facilitate replication and comparison in future studies.

Re: We apologize for not providing clear presentation. UPMiPP locates the discrepancy code through two phases. Whether the discrepancy code is malicious requires further determination. So these tools are used to detect whether the discrepancy code extracted by UPMiPP is malicious. This is primarily reflected in: (1) evaluating the performance of these tools in detecting code confusion attacks, and how UPMiPP can help improve their performance metrics, especially in terms of recall and precision; (2) detecting newly uploaded packages combined with UPMiPP.

• In Section 4.2, the authors introduce terms like "Malicious code line," "Repeated attacks," and "(In)consistency confusion" without prior definitions, which disrupts the flow and creates confusion for readers.

Re: We apologize for the unclear presentation that caused reading difficulties for the reviewer. Below are explanations for the terms:

Malicious code line: The line of code that executes the malicious functionality.

Repeated attacks: The same malicious code is used in multiple malicious packages.

(In)consistency confusion: Consistency confusion refers to a situation where the naming confusion target and the code confusion target do not align. Please refer to Example 1 in Section 2.2. Inconsistent confusion is the opposite case, as explained in Example 2 in Section 2.2.

Candidate project rank: This should be referred to as the "target project rank," indicating the rank of the true target project borrowed for the code confusion attack within the code similarity comparison sequence.

Confusion facet hit: This refers to which of the three confusion facets are hitted by a code confusion attack case.

• In Section 4.2, the statement about "meticulous examination" lacks detailed

explanation. The authors should clarify the process they followed to ensure that no malicious behavior was present in the samples with a code similarity of 1.0. Was this examination based on static analysis, dynamic analysis, or some other method? Detailing the methodology would enhance the scientific rigor of the claim.

Re: We apologize for not providing more details due to space limitations. We determined the presence of malicious behavior by manually inspecting each code file of these packages, and confirmed that they do not contain any malicious code.

Minor comments

• The statement, "For instance, the same 62 lines of code was identified in 28 instances," is presented without sufficient context.

Re: This statement means that the same 62-line malicious code appeared in 28 malicious packages, which further supports our finding: ***F2: Repeated attacks.*** *The same malicious code is repeated in multiple code confusion attacks.* (Line 570-571).

• In section 4.2, the authors speculation can undermine the objectivity and reliability of the findings. The authors' claim that the samples with 1.0 code similarity "might have been inadvertently included" needs to be substantiated with evidence or removed.

Re: We apologize for not providing more details due to space limitation. In fact, after identifying the samples with 1.0 code similarity, we reported them to the BKC repository, and some of them have already been deleted. Therefore, our claim is substantiated.

• The paper claims that 149 packages with code similarity in the range [0.9, 1.0) "conform to the nature of code confusion attack" but does not explain the methodology used to confirm this. Clear criteria or analysis steps are needed to validate this claim. Did you manually check them?

Re: We apologize for not providing more details. We manually inspected these 149 packages and found that they all borrowed from existing projects, retaining the majority

of the original code while injecting malicious code additionally. Therefore, we believe these packages "conform to the nature of a code confusion attack."

Reviewer C: Detailed comments for authors

Originality

Originality is limited as it primarily uses well-known malicious package datasets for answering three our of four research questions. Scalability is achieved through heuristics, which may have severe limitations.

Re: The BKC dataset we used includes over 3,000 real-world malicious PyPI packages collected from multiple sources. Due to the lack of a dedicated dataset for code confusion attacks, we identified 149 cases of code confusion attacks from this dataset to evaluate our method. Experimental results show that our approach can effectively identify the true target projects of code confusion attacks. Although our method was inspired by observations of malicious packages, the experimental results demonstrate its strong generalizability, as evidenced by the discovery of 27 in-the-wild code confusion attacks among newly uploaded packages.

Moreover, a large number of prior works have conducted similar analyses to identify malicious Android apps that clone popular apps and insert malicious code /payload.

Importance of contribution

Identification of malicious PyPI packages is essential. However, this paper fails to contribute much to improving knowledge in this direction.

Re: Our method addresses a novel problem scenario—detection of code confusion malicious packages on PyPI—with the following novelties:

(1) Innovative Problem Scenario: Clone attacks in Android apps[2][3][4] rely on code behavior similarities, while PyPI code confusion attacks focus on names, dependencies, and lightweight code tampering, requiring different detection strategies. Thus, our work does not simply adapt existing methods but is specifically designed and optimized for PyPI's unique threat model.

(2) Novel method design: Our method combines code similarity detection and package metadata analysis to form a comprehensive detection framework. It can effectively locate the true targets of code confusion and more precisely identify discrepancy code with a line-level code similarity analysis, going beyond the global code similarity checks common in Android clone apps.

(3) Experimental Uniqueness: Compared to Android app detection methods, our evaluations use different data and scenarios, demonstrating its superior performance.

(4) Contribution to PyPI: Our method has been validated in real-world scenarios, offering a new tool for defending against supply chain attacks. It successfully discovered 27 in-the-world attacks.

Soundness

• First, the new terminology, "code confusion attack," does not seem appropriate to me. Existing terminology such as "Trojanization," , "Cloning attack", "Dependency injection," or "Typo squatting" covers what the authors describe. To me, "Trojanization" would be a more appropriate one. In Android, this attack is known as "Cloning".

-The UPMiPP framework achieves scalability based on heuristics rules derived from the analyses of existing malicious packages' naming conventions, module organization, and README files. Hence, it would fail for sophisticated malicious ones that can evade detection by using strategies to bypass those heuristics.

Re: We proposed the term "code confusion attack" as a counterpart to the concept of "package confusion". While package confusion involves mimicking existing package names, code confusion refers to maintaining a high degree of similarity in code with existing projects. Therefore, we introduced the term "code confusion attack," which can better reflect the characteristics of software supply chain attacks and what we aim to study.

Our approach was inspired by observations of real-world samples and integrates various types of package metadata and code similarity analysis to identify the true target projects of code confusion attacks. Additionally, our method has been applied to detecting newly uploaded malicious packages, leading to the discovery of 27 in-the-

wild code confusion attacks. This further demonstrates the practical value of our approach.

• The weight values for the three facets should be empirically determined through regression analysis instead of assigning values that the authors think is appropriate.

Re: We thank the reviewer for your suggestions. During the rebuttal period, we conducted a regression analysis to test the ranking changes of the true target projects when assigning different weights to the three confusion dimensions. The experimental results show that as long as $W_d > W_s > W_n$ is maintained, the final ranking remains unchanged. We have included the results of this experiment at the anonymized site: https://anonymous.4open.science/r/21872632513/.

Evaluation

-Three out of the four research questions are based on existing malicious dataset.

• In RQ4 the authors mention that "Fortunately, they have been removed from the PyPI." -- which indicates that PyPI already has detection mechanisms for this type of attack.

Re: We sincerely apologize for the inconvenience this expression has caused the reviewers. In fact, we reported the malicious packages we discovered to PyPI, and these malicious packages were removed.

• Static code analysis tools, especially taint analysis tools, are well-known for identifying malicious code without using heuristics. For example

1. https://python.docs.openrefactory.com/
2. https://pyre-check.org/docs/pysa-basics/

Evaluation should also include "Taint analysis" based on security analysis tools.

Re: During the rebuttal period, we attempted to conduct evaluations using these two tools. We apologize for the fact that, despite our efforts, we failed to obtain performance results on these packages for the two recommended static taint analysis tools. (1)iCR for Python is a commercial tool and not open-source, making it difficult to obtain. (2)As

for Pyre, we tried it many times but failed to run it properly. We configured the OS environment, Python version, watchman, and other dependencies according to the documentation, but every time we ran "pyre analyze" command, we encountered the error: "ƛ Pyre exited with non-zero return code: 1.". Therefore, we conclude that PyRE cannot be applied to the detection of malicious packages.

Related works

Missing related works on other ecosystems to identify malicious packages. For example, a large number of works have been done on Android apps to identify malicious apps that clone popular ones and insert malicious codes.

Re: We sincerely thank the reviewers for your valuable suggestions. There has been some works dedicated to detecting malware in the NPM and Android ecosystems[1][2][3][4][5][6][7]. We will include an introduction to these related works in the next version of the paper.

[1] Yu Z, Wen M, Guo X, et al. Maltracker: A fine-grained npm malware tracker copiloted by llm-enhanced dataset[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024: 1759-1771.

[2] Zhang J, Huang K, Chen B, et al. Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence[J]. arXiv preprint arXiv:2309.02637, 2023.

[3] Zhang J, Huang K, Chen B, et al. Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence[J]. arXiv preprint arXiv:2309.02637, 2023.

[4] Sejfia A, Schäfer M. Practical automated detection of malicious npm packages[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1681-1692.

[5] Chen J, Alalfi M H, Dean T R, et al. Detecting android malware using clone detection[J]. Journal of Computer Science and Technology, 2015, 30: 942-956.

[6] Foroughipour A, Stakhanova N, Abazari F, et al. AndroClonium: Bytecode-level

code clone detection for obfuscated Android apps[C]//IFIP International Conference on ICT Systems Security and Privacy Protection. Cham: Springer International Publishing, 2022: 379-397.

[7] Xie J, Yan X, Lin Y, et al. An accurate and efficient two-phase scheme for detecting Android cloned applications[J]. Concurrency and Computation: Practice and Experience, 2021, 33(5): e6009.