PROFMAL: Detecting Mali the Synergy between Stati

Yiheng Huang*, Wen Zheng*, Su
You Lu*, Zhuotong Zhou*, Yiheng
*College of Computer Science and Artificial Intel
†Correspondin

***Abstract*—Open source software (OSS) has become the foundation of modern applications, but its transitive dependencies make it especially vulnerable to supply chain attacks. One common tactic is to inject malicious code into third-party packages. NPM, in particular, due to its widespread use and large volume of packages, has become the popular target of malicious code injection. While various detectors have been proposed, they suffer three limitations, i.e., inadequate behavior modeling of obfuscated code, ignoring object-centric features of JavaScript, and lack of synergy between static and dynamic analysis. These limitations lead to imprecise modeling of program behavior and hinder detection effectiveness.**

To address these limitations, we propose PROFMAL to identify malicious NPM packages, which leverages the synergy between static and dynamic analysis to construct behavior graphs for each package. Specifically, our static analysis constructs the behavior graphs through object-sensitive analysis, while identifying sensitive API calls and locating statically unresolved calls. Our dynamic analysis augments the behavior graphs by resolving those statically unresolved calls. Based on these comprehensive behavior graphs, we train a graph-based classifier to identify maliciousness. Our evaluation has indicated that PROFMAL achieves the highest F1-score of 92.4%, outperforming the state-of-the-arts by 6.2% to 48.8%. During a three-month real-world detection, PROFMAL has detected 496 previously unknown malicious NPM packages, and all of them have been confirmed and removed from NPM.

I. INTRODUCTION

Open-source software (OSS) constitutes up to 90% of a modern application's codebase, with all ecosystem downloads exceeding 6.6 trillion in 2024 [43]. Despite this ubiquity, limited visibility into transitive dependencies and insufficient investment in software supply chain (SSC) have left OSS vulnerable to certain threats, such as incompatibilities [16], vulnerabilities [2], and malicious packages [30], increasing SSC risks [18].

Problem. Malicious packages, which may be delivered via updates to legitimate libraries [10] or by publishing new packages that exploit common typos and namespace collisions [48], have emerged as a critical SSC threat. Since November 2023, over 512,847 malicious packages have been identified (a 156% year-over-year increase) [43], highlighting an escalating wave of SSC compromise. The NPM registry [27], in particular, has seen a marked proliferation of malicious packages, eroding the security of downstream applications. Given NPM's massive, rapidly expanding registry and ecosystem, manually inspecting newly published packages is infeasible. Therefore, automated malicious package detectors are essential.

Limitations of Existing Techniques. Various detectors have been proposed to detect malicious packages in the NPM ecosys-

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00042

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

P ROF M A L：通过静态意NPM包

Yiheng Huang*，Wen Zheng*，Su
You Lu*，Zhuotong Zhou*，Yiheng Ca
*复旦大学计算机科学技术学院，中国上海
†通讯作者

摘要—开源软件（OSS）已成为现代应用的基础，但其传递依赖使其尤其容易受到供应链攻击。一种常见策略是将恶意代码注入第三方包。NPM由于其广泛使用和庞大的包数量，尤其成为恶意代码注入的流行目标。尽管已有多种检测器被提出，但它们存在三个局限性，即对混淆代码的行为建模不充分、忽略了JavaScript的以对象为中心的特征，以及缺乏静态与动态分析之间的协同。这些局限性导致程序行为建模不精确，并阻碍了检测效果。

为解决这些局限性，我们提出PROFMAL来识别恶意NPM包，该方法利用静态与动态分析之间的协同作用为每个包构建行为图。具体而言，我们的静态分析通过对象敏感分析构建行为图，同时识别敏感API调用并定位静态无法解析的调用。我们的动态分析通过解析那些静态无法解析的调用来增强行为图。基于这些全面的行为图，我们训练了一个基于图的分类器来识别恶意性。我们的评估表明，PROFMAL达到了92.4%的最高F1分数，比现有最优方法高出6.2%至48.8%。在为期三个月的真实世界检测中，PROFMAL检测到496个此前未知的恶意NPM包，且所有这些包均已被确认并从NPM中移除。

一、引言

开源软件（OSS）占现代应用程序代码库的比例高达90%， 2024 [43]年所有生态系统的下载量超过6.6万亿次。尽管OSS无处不在，但对传递依赖的可见性有限以及对软件供应链（SSC）的投入不足，使OSS容易受到某些威胁的影响，例如不兼容性 [16], 、漏洞 [2], 和恶意软件包 [30], ，从而增加了SSC风险 [18]。

问题。恶意包——可能通过更新合法库 [10] 或发布利用常见拼写错误和命名空间冲突的新包 [48],来传播——已成为SSC的关键威胁。自2023年11月以来，已识别出超过512,847个恶意包（同比增长156%） [43], ，凸显了SSC安全威胁的不断升级。尤其是NPM仓库 [27], ，恶意包数量显著激增，侵蚀了下游应用的安全性。鉴于NPM庞大且快速扩张的仓库和生态系统，手动检查新发布的包是不可行的。因此，自动化的恶意包检测器至关重要。

现有技术的局限性。已有多种检测器被提出用于检测NPM生态系统中的恶意软件包，

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00042

licious NPM Packages by ic and Dynamic Analysis

Susheng Wu*, Bihuan Chen*†,
ng Cao*, Xiaoyu Li*, Xin Peng*
ntelligence, Fudan University, Shanghai, China
ding Author

与动态分析的协同检测恶

Susheng Wu*，Bihuan Chen*†，
Cao*，Xiaoyu Li*，Xin Peng*

tem, including rule-based [3, 6, 15, 22, 36, 57], learning-based [12, 14, 19, 25, 38, 50, 54], and LLM-based techniques [52]. Rule-based detectors often use static and/or dynamic analysis to model malicious behavior as patterns, and match them against predefined rules. Learning-based detectors typically adopt static and/or dynamic analysis to extract features, and train classifiers for detection. In both cases, the two analyses are usually employed in parallel (e.g., static and dynamic features used independently) or with dynamic analysis serving to confirm static results. LLM-based detectors leverage LLMs to directly understand the malicious behavior through prompt engineering, but are constrained in handling complex program behavior.

However, existing detectors suffer from inadequate modeling of program behavior, which hinders their effectiveness. This inadequacy arises from three main limitations.

L1: Inadequate Behavior Modeling of Obfuscated Code. Several detectors [19, 38] treat obfuscation as a feature in the learning, resulting in false positives, as such patterns are also present in benign packages. Huang et al. [15] design a dedicated classifier to directly detect obfuscation, and treat it as a sign of maliciousness, which leads to false positives when benign packages are obfuscated. Instead, after detecting obfuscation, Huang et al. [14] adopt dynamic analysis to model behavior sequences, but noise in such sequences can still cause false positives. Zahan et al. [52] leverage LLM to identify the presence of obfuscation, and incorporate it as one of the factors in their final decision-making process. While this allows for certain flexibility, it still heavily relies on the capability of the LLM to discern malicious patterns. Some detectors [50, 54] completely ignore obfuscated code, potentially leading to false negatives.

L2: Ignoring Object-Centric Features of JavaScript. Detectors [14, 15, 38, 50, 54] that rely on lexical or syntactic patterns often struggle with JavaScript's object-centric nature, and thus fail to model the suspicious behaviors embedded within object-based constructs, which leads to false negatives. While program dependency graph (PDG)-based analyses [15, 50] are effective at modeling program behavior by capturing control and data dependencies between statements, they exhibit weaknesses when analyzing object-oriented operations (e.g., property access and object binding) as they are object-insensitive. This limitation results in missed detection of sensitive API calls. It creates opportunities for malicious packages to conceal harmful behaviors beyond the reach of existing detectors.

L3: Lack of Synergy between Static and Dynamic Analy-

包括基于规则的[3, 6, 15, 22, 36, 57], 基于学习的[12, 14, 19, 25, 38, 50, 54], 以及基于大语言模型的技术 [52]。基于规则的检测器通常使用静态和/或动态分析将恶意行为建模为模式，并与预定义规则进行匹配。基于学习的检测器通常采用静态和/或动态分析来提取特征，并训练分类器进行检测。在这两种情况下，两种分析通常并行使用（例如，静态特征和动态特征独立使用），或以动态分析用于确认静态结果。基于大语言模型的检测器利用大语言模型通过提示工程直接理解恶意行为，但在处理复杂程序行为方面受到限制。

然而，现有检测器对对程序行为建模不足，这阻碍了其有效性。这种不足源于三个主要局限。

L1：对混淆代码的行为建模不足。若干检测器[19, 38] 将混淆视为学习中的一个特征，导致误报，因为此类模式在良性包中也存在。Huang等 [15] 设计了一个专用分类器来直接检测混淆，并将其视为恶意性的标志，当良性包被混淆时会导致误报。相反，在检测到混淆后，Huang等 [14] 采用动态分析来对行为序列建模，但此类序列中的噪声仍可能导致误报。Zahan等 [52] 利用大语言模型来识别混淆的存在，并将其纳入最终决策过程的因素之一。虽然这提供了一定的灵活性，但仍严重依赖大语言模型识别恶意模式的能力。一些检测器 [50, 54] 完全忽略混淆代码，可能导致假阴性。

L2：忽视JavaScript中以对象为中心的特征。依赖词法或语法模式的检测器 [14, 15, 38, 50, 54] 往往难以应对JavaScript以对象为中心的特性，因而无法对嵌入在基于对象的构造中的可疑行为进行建模，导致假阴性。基于程序依赖图（PDG）的分析 [15, 50] 虽然能够通过捕获语句间的控制依赖和数据依赖有效地对程序行为进行建模，但在分析面向对象操作（例如属性访问和对象绑定）时存在弱点，因为它们对对象不敏感。这一局限性导致漏检敏感API调用，为恶意软件包提供了隐藏有害行为、逃避现有检测器检测的机会。

L3：静态与动态分析之间缺乏协同

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

sis. Existing detectors struggle to model behaviors that involve dynamic features, e.g., on-the-fly module loading and computed property access, which often leads to false negatives. Dynamic analysis is crucial for capturing JavaScript's runtime behavior. However, existing detectors either overlook dynamic analysis [50, 52, 54], or separately use it as confirmation after static analysis [14, 15] (e.g., detecting obfuscation or sensitive API calls that need further confirmation). Detectors that run static and dynamic analyses in parallel produce disjoint results without merging their evidences [6]. Purely dynamic detectors, on the other hand, generate API call sequences, which are often overwhelmed by noise and cause false positives [57]. Consequently, no detector produces a unified representation that integrates dynamic evidences into static ones, preventing static and dynamic analysis from fully leveraging the strengths of the other.

Our Approach. To address these limitations, we propose PROFMAL, a unified graph-based detector that leverages the synergy between static and dynamic analysis for malicious NPM package detection. PROFMAL focuses on malicious behaviors embedded during installation and import, as the majority of malicious packages inject code at these stages [30]. PROFMAL constructs behavior graphs for a package, modeling control flows and data dependencies between statements, even in the presence of obfuscation, thereby addressing **L1**. Our object-sensitive static analysis is performed to resolve object references and aliasing, enabling accurate identification of sensitive API calls, thereby addressing **L2**. Our object-sensitive static analysis further allows PROFMAL to identify statically unresolved calls, which are then resolved via dynamic analysis. Our dynamic analysis results are merged back into the behavior graphs, yielding a unified and comprehensive representation, thereby addressing **L3**. Finally, suspicious behavior subgraphs are extracted and classified by a trained detector.

Evaluation. We compared PROFMAL to five state-of-the-arts, i.e., GUARDDOG [3], SPIDERSCAN [15], MALTRACKER [50], CEREBRO [54] and SOCKETAI [52]. PROFMAL achieved the highest F1-score of 92.4%, outperforming state-of-the-arts by 6.2% to 48.8%. We ran all the detectors in real-world and monitored newly published NPM packages for three months. PROFMAL detected 496 previously unknown malicious packages, achieving the lowest false positive rate of 28.3%. All these malicious packages have been confirmed and removed from NPM.

Contribution. This work makes the following contributions.

- We proposed PROFMAL to detect malicious NPM packages, which leverages the synergy between static and dynamic analysis to model behaviors with unified behavior graphs.
- We conducted experiments to demonstrate the effectiveness and practical usefulness of PROFMAL, and detected 496 new malicious packages in real-world detection.
- We released the code of PROFMAL at our website [46].

II. MOTIVATING EXAMPLES

Fig. 1 shows the code snippet of a malicious package *bit-soex_react-design-system_14.1.4*, illustrating how an attacker employs code obfuscation to conceal sensitive API calls and exfiltrate personally identifiable information (PII) via network

。现有检测器难以对涉及动态特征的行为进行建模，例如即时模块加载和计算属性访问，这往往导致假阴性。动态分析对于捕获JavaScript的运行时行为至关重要。然而，现有检测器要么忽视动态分析[50, 52, 54], ，要么在静态分析之后将其单独用作确认手段 [14, 15] （例如检测需要进一步确认的混淆或敏感API调用）。并行运行静态和动态分析的检测器会产生互不相交的结果，而不会合并各自的证据 [6]。另一方面，纯动态检测器会生成API调用序列，这些序列往往被噪声淹没并导致误报 [57]。因此，没有任何检测器能够产生一种将动态证据整合到静态证据中的统一表示，这使得静态与动态分析无法充分发挥彼此的优势。

我们的方法。为解决这些局限性，我们提出PROFMAL，一种统一的基于图的检测器，利用静态分析与动态分析之间的协同作用进行恶意NPM包检测。PROFMAL聚焦于安装和导入阶段嵌入的恶意行为，因为大多数恶意包在这些阶段注入代码 [30]。PROFMAL为包构建行为图，建模语句间的控制流和数据依赖，即使在存在混淆的情况下也能做到，从而解决**L1**。我们执行对象敏感静态分析以解析对象引用和别名，从而准确识别敏感API调用，进而解决**L2**。我们的对象敏感静态分析还使PROFMAL能够识别静态无法解析的调用，这些调用随后通过动态分析加以解析。我们将动态分析结果合并回行为图，形成统一且全面的表示，从而解决**L3**。最后，提取可疑行为子图并由训练好的检测器进行分类。

评估。我们将PROFMAL与五种最先进方法进行了比较，即GUARDDOG [3], 、SPIDERSCAN [15], 、MALTRACKER [50],、CEREBRO [54] 和SOCKETAI [52]。PROFMAL取得了最高的F1分数92.4%，比最先进方法高出6.2%至48.8%。我们在真实环境中运行了所有检测器，并对新发布的NPM包进行了为期三个月的监控。PROFMAL检测到496个先前未知的恶意包，实现了28.3%的最低误报率。所有这些恶意包已被确认并从NPM中移除。

贡献。本工作做出了以下贡献。

- 我们提出了PROFMAL来检测恶意NPM包，它利用静态分析和动态分析之间的协同作用，通过统一行为图来建模行为。
- 我们进行了实验，证明了PROFMAL的有效性和实用性，并在真实世界检测中发现了496个新的恶意包。
- 我们在网站上发布了PROFMAL的代码 [46]。

II. 动机示例

图 1 展示了恶意包 *bit-soex react-design-system 14.1.4* 的代码片段，说明了攻击者如何利用代码混淆来隐藏敏感API调用，并通过网络窃取个人可识别信息（PII）

420
uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于 2026 年 11 月 02 日 05:12:30 UTC 从 IEEE Xplor

```
1  var _0x26cfab = _0x4fc7;
2  function _0x4fc7(_0x1a8550, _0x5d5dfc) {
3    ... }
4  var os = require("os");
5  admin_text = os[_0x26cfab(0x1b8)]()[_0x26cfab(0x1a1)];
6  try {
7   const { execSync } = require(_0x26cfab(0x1b9));
8   let stdout = execSync(_0x26cfab(0x1ba))
9        [_0x26cfab(0x1bc)]()
10       [_0x26cfab(0x1ab)]("\x0a", "");
11       admin_text += "\x20" + stdout;
12  } catch {
13    ... }
14 const https = require(_0x26cfab(0x1b0)),
15 options = {
16    hostname: _0x26cfab(0x1a3),
17    port: 0x1bb,
18    path: "/?Username=" +
19          encodeURI(username +
20                    "\x20(" +
21                    admin_text + ")") +
22          ...,
23    method: _0x26cfab(0x1a5),
24 },
25 req = https["request"](options);
```

Fig. 1: Code Snippet of *bitsoex_react-design-system_14.1.4*

requests. Specifically, the attacker adopts syntactic symbol obfuscation and dynamic property generation, which are common tactics used to evade detection [30]. This code snippet defines a calculation function `_0x4fc7` at Line 2, which maps numeric codes to string literals at runtime, and binds it to an obfuscated variable `_0x26cfab` at Line 1. It loads the module `os` at Line 4, and calls `os.userInfo()` at Line 5, where the property name `userInfo` is dynamically generated by `_0x26cfab`. At Line 7, it uses `_0x26cfab` to require `child_process` and obtain `execSync`, then executes an obfuscated system command at Line 8 and appends the result to `admin_text` at Line 11. Finally, it calls `https.request()` with its module name dynamically generated at Line 14, and sends an HTTPS request that embeds the stolen PII in the URL. While static analysis struggles to detect these obfuscated sensitive API calls due to their dynamic nature (illustrating **L1**), it remains capable of constructing control flows and data dependencies among these calls. To resolve these obfuscated API calls, dynamic analysis is required to capture the underlying API invocations [14, 57]. This example emphasizes the importance of complementing static analysis with dynamic analysis, illustrating **L3**.

Fig. 2 shows the code snippet of a malicious package *automation.samples_0.1.15*, illustrating how an attacker leverages JavaScript's object-centric features, specifically function object reference and aliasing, to hide sensitive API calls and exfiltrate PII. Specifically, the attacker builds a `data` dictionary at Line 9, containing PII by passing the function object of sensitive APIs `os.hostname` at Line 11 and `os.homedir` at Line 12 to the function `tryGet` defined at Line 2. The function `tryGet` then invokes these APIs via `toCall()` at Line 4, which acts as an alias for the actual API calls `os.hostname()` and `os.homedir()`. This indirection through function object references and aliasing not only hides the direct API calls but also reflects JavaScript's object-centric features, where in this example, functions are treated as objects and passed around. This tactic weakens detectors [15, 38, 50, 54] that rely on direct identifier resolution, illustrating **L2**. This example emphasizes the critical need for object-sensitive analysis to detect sensitive

```
1  var _0x26cfab = _0x4fc7;
2  functisn _0x4fc7"_0x1a0000[ _0x0h0hfc0 4
3 ... }
4  var 操作系统模块 = riuuiri"8sx80;
0  aenmp1xf(x = 操作系统模块"16(00dhac]6(8c0_0]_1_0x26cfab"0x1a102;
6  尝试 4
7    csnxt 4 ixicb)nc 5 = riuuiri"_0x26cfab"0x1b00;
0    pit 标准输出 = f(fda)pd]16(00dhac]6(8ca__
0         1_0x26cfab(0x1bc)2()
10        1_0x26cfab(0x1ab)2("\x0a", "");
11        aenmp1xf(x 0= 86x208 0 标准输出;
12   5 捕获 4
13 ... }
14  csnxt ntttx = riuuiri"_0x26cfab"0x1b000[
10  选项 = 4
16 主机名: _0x26cfab(0x1a3),
17 端口: 0x1bb,
18 路径: "/?用户名="0
19 encodeURI(用户名 0
20 "\x20("0
21                    aenmp1xf(x 0 8080 0
22 ...,
23 方法: _0x26cfab(0x1a5),
24 },
20  riu = ixxru"2tfsyfux20（选项）;
```

图 1: *bitsoex react-design-system 14.1.4* 的代码片段

请求。具体而言，攻击者采用了语法符号混淆和动态属性生成，这些是用于规避检测的常见策略 [30]。该代码片段在第2行定义了一个计算函数 _0x4fc7，它在运行时将数字代码映射到字符串字面量，并在第1行将其绑定到混淆变量0x26cfab。它在第_4行加载操作系统模块，并在第5行调用os.userInfo()，其中属性名userInfo由 _0x26cfab动态生成。在第7行，它使用 _0x26cfab来require child_process并获取execSync，然后在第8行执行一个混淆的系统命令，并在第11行将结果追加到admin_text。最后，它在第14行动态生成模块名并调用https.request()，发送一个将stolenPII嵌入URL的HTTPS请求。尽管静态分析由于其动态特性而难以检测这些混淆的敏感API调用（如**L1**所示），但它仍能够构建这些调用之间的控制流和数据依赖。要解析这些混淆的API调用，需要动态分析来捕获底层的API调用 [14, 57]。这个例子强调了用动态分析补充静态分析的重要性，如**L3**所示。

图2展示了恶意包*au-tomation.samples 0.1.15*的代码片段，说明了攻击者如何利用JavaScript以对象为中心的特征，特别是函数对象引用和别名，来隐藏敏感API调用并窃取PII。具体而言，攻击者通过在第11行将敏感API os.hostname的函数对象和第12行将os.homedir的函数对象传递给第2行定义的tryGet函数，在第9行构建了包含PII的数据字典。然后，tryGet函数在第4行通过toCall()调用这些API，toCall()充当实际API调用os.hostname()和os.homedir()的别名。通过函数对象引用和别名进行的这种间接调用不仅隐藏了直接API调用，还反映了JavaScript以对象为中心的特征——在此示例中，函数被视为对象进行传递。这一策略削弱了依赖直接标识符解析的检测器 [15, 38, 50, 54]，说明了**L2**。此示例强调了进行对象敏感分析以检测敏感

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

420

lore 下载。适用限制。

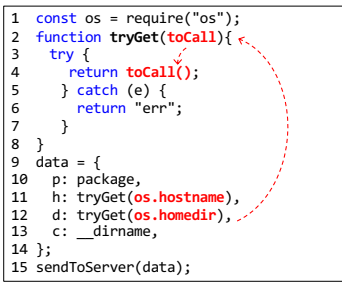


Fig. 2: Code Snippet of *automation.samples_0.1.15*

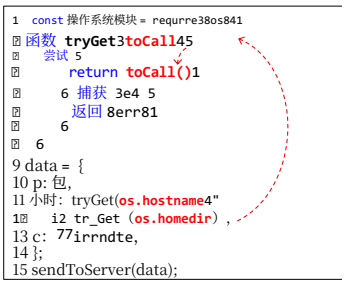


图 2：*automation.samples 0.1.15*的代码片段

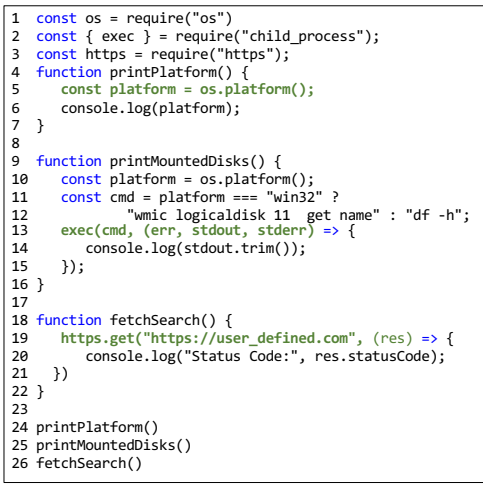


Fig. 3: Code Snippet of *echo-tool-1.0.1*



图 3：*echo-tool-1.0.1*的代码片段

API calls and thus detect malicious behavior.

Fig. 3 shows the code snippet of a benign package *echo-tool-1.0.1*, which performs a simple startup check, i.e., printing platform information, listing mounted disks, and testing network connectivity. Specifically, it detects the operating system using `os.platform()` at Line 5, and logs the result at Line 6. It then uses `child_process.exec()` at Line 13 to execute a platform-specific shell command, determined by the value of `platform`. Finally, it tests network connectivity via `https.get()` at Line 19. From a static analysis perspective, there is no direct data dependency among `os.platform()` (Line 5), `child_process.exec()`, and `https.get()`. Each API is invoked independently, and no sensitive data is transferred or leaked. Hence, a static detector can identify this behavior as benign. However, from a dynamic analysis perspective that relies on behavior sequence, the series of operations, i.e., system inspection via `os.platform()`, command execution via `child_process.exec()`, and network communication via `https.get()` is similar to the behavioral pattern as shown in Fig. 2, which indicates PII leakage. Without the code structure and context specifying the relations between these API calls, this similarity leads to false positives in purely dynamic or behavior-sequence-based detectors [14, 57]. This example also emphasizes the critical need for complementing dynamic analysis with static analysis, illustrating **L3**.

API调用并进而检测恶意行为的关键必要性。

图3展示了一个良性包*echo-tool-1.0.1*的代码片段，它执行简单的启动检查，即打印平台信息、列出已挂载磁盘以及测试网络连通性。具体而言，它在第5行使用os.platform()检测操作系统，并在第6行记录结果。随后在第13行使用child_process.exec()执行由platform值确定的平台特定Shell命令。最后，它在第19行通过https.get()测试网络连通性。从静态分析的角度来看，os.platform()（第5行）、child_process.exec()和https.get()之间不存在直接的数据依赖。每个API独立调用，没有敏感数据被传输或泄露。因此，静态检测器可以将此行为识别为良性。然而，从依赖行为序列的动态分析角度来看，这一系列操作——即通过os.platform()进行系统检查、通过child_process.exec()执行命令，以及通过https.get()进行网络通信——与图2所示的行为模式相似，该模式指示PII泄露。在没有代码结构和上下文来明确这些API调用之间关系的情况下，这种相似性会导致纯动态或基于行为序列的检测器产生误报 [14, 57]。此示例还强调了将动态分析与静态分析相结合的关键必要性，如**L3**所示。

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

421

权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

III. Approach

To address the limitations of existing detectors, we propose ProfMal, a unified graph-based approach that leverages the synergy between static and dynamic analysis to detect malicious NPM packages. As shown in Fig. 4, it has five key modules.

- **Script Analyzer** (Sec. III-A). This module scans a package's installation scripts to detect malicious shell commands. It also identifies all entry files used during installation and import time, along with third-party libraries, laying the foundation for subsequent behavior graph construction.
- **Behavior Graph Static Generator** (Sec. III-B). Using script analyzer's outputs, this module constructs behavior graphs by modeling control flows and data dependencies between statements, and identifies sensitive API calls by our object-sensitive analysis, thereby addressing limitations **L1** and **L2**.
- **Behavior Graph Dynamic Augmentor** (Sec. III-C). The behavior graphs may include behaviors that cannot be statically resolved as illustrated in Sec. II. This module augments the behavior graphs, addressing limitations **L1** and **L3**.
- **Sensitivity Evaluator** (Sec. III-D). This module assigns sensitivity scores to specific sensitive API calls based on a predefined list, taking into account concrete arguments and return values. Higher scores indicate greater potential risk.
- **Maliciousness Detector** (Sec. III-E). This module first extracts a suspicious behavior subgraph from each behavior graph, retaining only sensitive API calls and their associated control flow and data dependency relations. Then it classifies the subgraph using a trained graph neural network (GNN). A package is flagged as malicious if the GNN's confidence score exceeds a predefined threshold.

A. Script Analyzer

Our script analyzer processes the *package.json* file of a package to 1) detect malicious shell commands in installation scripts, 2) identify entry files used during installation and import time, and 3) resolve third-party libraries depended on by the package.

Malicious Shell Command Detection. During package installation via `npm install`, the package manager automatically executes shell commands specified in the installation script fields of *package.json*, e.g., `preinstall`, `install`, and `postinstall`. These scripts are executed without user interaction, and can contain arbitrary shell commands, thereby introducing a significant attack surface, as attackers can leverage them to run malicious code during the installation time. To address this threat, we first detect potential malicious shell commands in these fields. If a malicious shell command is detected, ProfMal reports it while continuing to analyze the package code for any further malicious behavior.

We detect malicious shell commands based on their behavior, similar to the malicious shell command detection approach used in SpiderScan [15] and SocketAI [52]. ProfMal prompts the LLM to analyze the behavior of the command and judge its maliciousness. Following prior work, our prompt design adopts the structure of SpiderScan, which consists of four components: role specification, task description, few-shot examples, and a structured output format. We extend

III. 方法

为解决现有检测器的局限性，我们提出PROFMAL，一种基于图的统一方法，利用静态与动态分析的协同作用来检测恶意NPM包。如图4所示，它包含五个关键模块。

- **脚本分析器**（第III-A节）。该模块扫描包的安装脚本以检测恶意shell命令。它还识别安装时和导入时使用的所有入口文件以及第三方库，为后续行为图构建奠定基础。
- **行为图静态生成器**（第III-B节）。利用脚本分析器的输出，该模块通过建模语句之间的控制流和数据依赖来构建行为图，并通过我们的对象敏感分析识别敏感API调用，从而解决局限**L1** 和**L2**。
- **行为图动态增强器**（第III-C节）。如第II节所示，行为图可能包含无法静态解析的行为。该模块增强行为图，解决局限**L1** 和**L3**。
- **敏感性评估器**（第 III-D 节）。该模块基于预定义列表为特定敏感 API 调用分配敏感度分数，并考虑具体参数和返回值。分数越高，表示潜在风险越大。
- **恶意性检测器**（第 III-E 节）。该模块首先从每个行为图中提取可疑行为子图，仅保留敏感 API 调用及其关联的控制流和数据依赖关系。然后使用训练好的图神经网络（GNN）对子图进行分类。如果 GNN 的置信度分数超过预定义阈值，则该包被标记为恶意。

A. 脚本分析器

我们的脚本分析器处理包的*package.json* 文件，以 1) 检测安装脚本中的恶意Shell命令，2) 识别安装和导入时使用的入口文件，以及 3) 解析包所依赖的第三方库。

恶意Shell命令检测。在通过 npm install 安装包的过程中，包管理器会自动执行 *package.json* 安装脚本字段中指定的 Shell 命令，例如 preinstall、install 和 postinstall。这些脚本在无需用户交互的情况下执行，并且可以包含任意 Shell 命令，从而引入了显著的攻击面，因为攻击者可以利用它们在安装时运行恶意代码。为应对这一威胁，我们首先检测这些字段中潜在的恶意Shell命令。如果检测到恶意Shell命令，PROFMAL 会报告该命令，同时继续分析包代码以发现任何进一步的恶意行为。

我们基于行为来检测恶意shell命令，类似于SPIDERSCAN [15]和SOCKETAI [52]中使用的恶意shell命令检测方法。PROFMAL提示大语言模型分析命令的行为并判断其恶意性。遵循先前工作，我们的提示设计采用SPIDERSCAN的结构，该结构由四个部分组成：角色设定、任务描述、少样本示例和结构化输出格式。我们扩展

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
下载。适用相关限制。

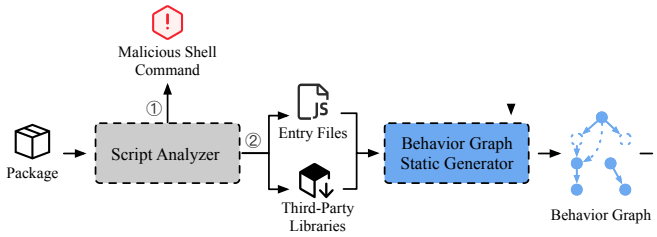


Fig. 4: Approach Overv

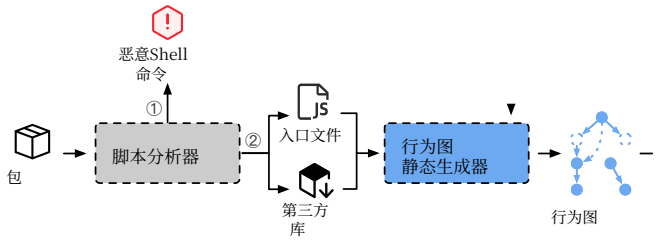


图 4：PROFMAL 方法概览

Prompt Snippet

Task: You are a security expert with extensive experience in Linux shell programming and security risk assessment. You will receive one input: `<Shell Command>`. Your task is to analyze the command's behavior and determine if it is malicious.

Guidelines: Malicious behaviors are defined as follows (with examples):
1. Data Exfiltration: Commands that retrieve local data (e.g., passwords, system logs, /etc/passwd) and send it externally (e.g., via remote server, URL, or DNS lookup).
2. Script or Binary Execution: Commands that execute shell scripts (.sh), batch files (.bat), or executable binaries (.exe).
3. Download and Execution: Commands that both download and execute files in a single step.
...
If a command does not clearly match any of these categories, use expert judgment to assess whether it exhibits malicious behavior.

Output: Only `Malicious` or `Benign` with no additional explanation.

Fig. 5: Prompt Snippet for Malicious Shell Command Detection

this structure by adding few-shot examples aligned with our expanded taxonomy of 13 malicious behaviors. These behaviors are distilled from prior studies [14, 15, 30, 47, 52, 58] and include: (1) Data Exfiltration, (2) Script or Binary Execution, (3) Dropper, (4) Download and Execution, (5) Critical File Tampering, (6) Process Injection, (7) Data Obfuscation and Encoding, (8) Reverse Shell, (9) System Shutdown, (10) Critical File Deletion, (11) Unusual URL Interaction, (12) Non-Typical Node Execution, and (13) Resource Exhaustion. Each behavior is abstracted into a natural language description and integrated into the prompt. The prompt snippet used for malicious shell command detection is presented in Fig. 5, while the complete prompt is available at our replication website [46].

Entry Files Extraction. To detect malicious behavior during import time, where harmful code is often embedded, we extract entry files, including JavaScript files referenced in the `main`, `exports`, and `bin` fields of *package.json*, which are typically triggered on `require` statements. Besides, we also use regular expressions to extract JavaScript files executed in the installation scripts. The output of this step is a set of entry files.

Third-Party Libraries Resolution. To gather all dependent third-party libraries, we parse the `dependencies` field of *package.json*, and then download the libraries. While not the only way to incorporate third-party libraries, this field remains the predominant and conventionally recommended mechanism.

B. Behavior Graph Static Generator

Our static generator takes entry files and third-party libraries as inputs, and constructs a behavior graph (BG) for each entry

提示片段

任务：你是一位在Linux Shell 编程和安全风险评估方面拥有丰富经验的安全专家。你将收到一个输入：<Shell 命令>。你的任务是分析该命令的行为并判断其是否具有恶意。

指南：恶意行为的定义如下（附示例）：1. 数据外泄：获取本地数据（例如密码、系统日志、/etc/passwd）并将其发送至外部（例如通过远程服务器、URL 或 DNS 查询）的命令。2. 脚本或二进制执行：执行 shell 脚本（.sh）、批处理文件（.bat）或可执行二进制文件（.exe）的命令。3. 下载并执行：在单一步骤中同时下载并执行文件的命令。……如果某条命令未明确匹配上述任何类别，请运用专家判断来评估其是否表现出恶意行为。**输出：**仅输出“恶意”或“良性”，不作额外解释。

图 5：恶意shell命令检测的提示片段

这一结构，通过添加与扩充后的13种恶意行为分类体系对齐的少样本示例来实现。这些行为提炼自先前研究 [14, 15, 30, 47, 52, 58]，包括：(1) 数据外泄，(2) 脚本或二进制执行，(3) 投放器，(4) 下载并执行，(5) 关键文件篡改，(6) 进程注入，(7) 数据混淆与编码，(8) 反向 Shell，(9) 系统关机，(10) 关键文件删除，(11) 异常 URL 交互，(12) 非典型节点执行，以及 (13) 资源耗尽。每种行为都被抽象为自然语言描述并集成到提示中。用于恶意shell命令检测的提示片段见 Fig. 5，完整提示可在我们的复现网站 [46] 获取。

入口文件提取。为了检测导入时（此时往往嵌入有害代码）的恶意行为，我们提取入口文件，包括 *package.json* 中 main、exports 和 bin 字段所引用的 JavaScript 文件，这些文件通常在 require 语句时被触发。此外，我们还使用正则表达式提取在安装脚本中执行的 JavaScript 文件。此步骤的输出是一组入口文件。

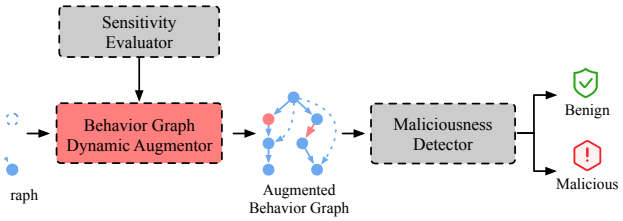
第三方库解析。为了收集所有依赖的第三方库，我们解析*package.json*的 dependencies 字段，然后下载这些库。虽然这不是引入第三方库的唯一方式，但该字段仍然是主要且传统上推荐的机制。

B. 行为图静态生成器

我们的静态生成器以入口文件和第三方库作为输入，并基于代码的语法结构为每个入口文件构建行为图（BG）

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载



erview of PROFMAL

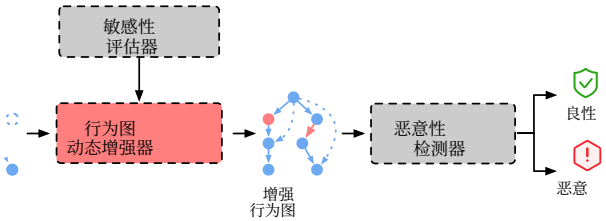
file based on the syntactic structure of the code. The generated BGs capture the overall behavior of the package, including its interactions with third-party libraries.

1) Overall Procedure: To generate the behavior graph (BG), we first construct a code property graph (CPG) for the package and its third-party libraries using Joern [17]. The CPG comprises the control flow graph (CFG) and the data dependency graph (DDG). The DDG models data dependencies within each single function, where nodes represent statements and edges capture the data dependencies between them. The CFG captures the control flows among these statements. To unify control flows and data dependencies, we integrate the CFG into the DDG by adding control flow edges among statements, resulting in a graph denoted as DDG^{+}. In parallel, we generate a call graph (CG) using Jelly [24], where call edges link call sites to their corresponding callees, enabling inter-procedural analysis across functions.

Next, we model each entry file as an implicit *main* function, encompassing all code in the file's global scope. Starting from this implicit *main* function of each entry file, we traverse the nodes in its DDG^{+} following the control flow to mimic the execution, and use the CG to step into other function at a function call node. The traversal runs object-sensitive and flow-sensitive analysis by analyzing each expression within a statement. For example, when encountering the statement `tryGet(os.hostname)`, the analysis first resolves the subexpression `os.hostname` and then proceeds to the outer call expression `tryGet` following AST semantics.

Sensitive API calls are important to model malicious behavior. One major challenge in identifying sensitive API calls lies in the inaccuracy caused by indirect calls as shown in Section II, where the actual target of an API call is hidden by an intermediate variable. To address this, the goal of our analysis is to identify all API calls and then match them against a predefined list of sensitive APIs to pinpoint the sensitive ones. Specifically, we track object references (e.g., resolving the property access `os.hostname` to its corresponding function object, as shown in Fig. 2), and maintain alias relations (e.g., treating the identifier `toCall` in `tryGet()` as an alias for the object referenced by `os.hostname`, as shown in Fig. 2) throughout the code.

Additionally, our analysis detects call sites that cannot be resolved statically, such as unidentified sensitive API calls (e.g., `os[0x26cfab(0x1b8)]()` in Fig. 1, where the property value is dynamically generated), or unresolved calls to third-party library functions (e.g., `axios.get()`, where the callee is dynamically bound via `fn.apply()`). These unresolved



览

生成的行为图捕获了包的整体行为，包括其与第三方库的交互。

1) 整体流程：为了生成行为图（BG），我们首先使用 Joern [17] 为包及其第三方库构建代码属性图（CPG）。CPG 包含控制流图（CFG）和数据依赖图（DDG）。DDG 对每个单一函数内的数据依赖进行建模，其中节点表示语句，边捕获它们之间的数据依赖关系。CFG 捕获这些语句之间的控制流。为了统一控制流和数据依赖，我们通过在各语句之间添加控制流边，将 CFG 集成到 DDG 中，得到一个记为 DDG^{+} 的图。与此同时，我们使用 Jelly [24]，生成调用图（CG），其中调用边将调用点链接到其对应的被调用者，从而支持跨函数的跨过程分析。

接下来，我们将每个入口文件建模为一个隐式*main*函数，涵盖该文件全局作用域中的所有代码。从每个入口文件的这个隐式*main* 函数开始，我们遍历其 DDG^{+} 中的节点，沿控制流模拟执行，并在函数调用节点处使用 CG 进入其他函数。该遍历通过在语句内分析每个表达式来执行对象敏感与流敏感分析。例如，当遇到语句 tryGet(os.hostname) 时，分析首先解析子表达式 os.hostname，然后按照 AST 语义继续处理外层调用表达式 tryGet。

敏感API调用对于建模恶意行为至关重要。识别敏感API调用的一个主要挑战在于间接调用所导致的不准确性，如第II节所示，其中API调用的实际目标被中间变量所隐藏。为解决这一问题，我们分析的目标是识别所有API调用，然后将其与预定义的敏感 API 列表进行匹配，以精确定位敏感调用。具体而言，我们在整个代码中跟踪对象引用（例如，将属性访问 os.hostname 解析为其对应的函数对象，如图2所示），并维护别名关系（例如，将 tryGet() 中的标识符 toCall 视为 os.hostname 所引用对象的别名，如图2所示）。

此外，我们的分析能够检测到无法静态解析的调用点，例如未识别的敏感API调用（如图1中的`os[0x26cfab(0x1b8)]()`，其中属性值是动态生成的），或对第三方库函数的未解析调用（例如`axios.get()`，其中被调用者通过`fn.apply()`动态绑定）。这些未解析的

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

422

下载。适用限制。

calls are subsequently handled by dynamic analysis (Sec. III-C).

After analyzing each node in DDG$^+$, we add it to the BG and connect it to related nodes based on their relations in DDG$^+$. For function call nodes, we attach analysis-derived attributes. Formally, the resulting BG is denoted as a directed graph $G = (V, E)$, where V is the set of nodes, and each node represents a statement; and E is the set of edges connecting nodes, including control-flow and data-dependency edges. Each function call node $v \in V$ is annotated with $\langle T_v, C_v, U_v, E_v \rangle$ and s_v. Here, $T_v = 1$ denotes a *sensitive* node, indicating that the node represents a sensitive API call. $C_v = 1$ denotes a *conditional* node, indicating that the node is potentially sensitive but requires further assessment. For example, `fs.readSync` is considered sensitive due to its potential to leak data, but the actual risk depends on the usage context, such as its arguments and the data being accessed. The *sensitivity score* s_v is defined only when $T_v = 1$ or $C_v = 1$, and quantifies the risk level of each sensitive API. $U_v = 1$ denotes an *unknown* node, meaning the call cannot be resolved statically; and $E_v = 1$ indicates an *eval* node, which executes dynamically generated code via `eval`.

To track object references and maintain alias relations, we mainly analyze four types of statements, i.e., **Require**, **Property Access**, **Assignment**, **Function Call**, during the traversal. These four types of statements are central to how objects are created, propagated, and aliased throughout the program.

The traversal is single-pass. As we traverse the DDG$^+$, we maintain two mappings at each program point.

- **Identifier Map** ($idMap$): Its keys are identifiers id, and its values are the corresponding objects obj, allowing the retrieval of the object associated with a given identifier by $obj = idMap[id]$. It maintains alias relations by recording the object each identifier refers to during the mimicked execution, enabling us to track how different identifiers point to the same object across the program. Since identifier visibility varies across different scopes, $idMap$ also maintains information about the scope in which each identifier is accessible. To capture these visibility relations, it employs a stack-based structure, updating the available identifiers as the traversal enters or exits code scopes (e.g., function entries and exits). Specifically, $idMap$ is updated upon assignment statements where the left-hand side is a single identifier (Sec. III-B4).
- **Property Map** ($propMap$): Its keys are object-property pairs $\langle obj, prop \rangle$ (denoting the property $prop$ of an object obj), and its values are the corresponding target objects, allowing the retrieval of the object referenced by a given object-property pair by $obj_{ref} = propMap[\langle obj, prop \rangle]$. It tracks the object referenced by each property of an object, thereby facilitating the analysis of property-based references. Specifically, $propMap$ is updated upon assignment to the property of an object (Sec. III-B4).

For each object, we maintain two attributes.

- **Qualified Object Path** ($qPath$): It records the hierarchical property access sequence leading to the first definition of the object itself. The $qPath$ is denoted as:

$$qPath = \langle root, props \rangle, \text{ with } props = [p_1, ..., p_n]$$

调用随后由动态分析处理（第III-C节）。

在分析 DDG$^+$ 中的每个节点后，我们将其添加到 BG 中，并根据其在 DDG$^+$ 中的关系将其与相关节点连接。对于函数调用节点，我们附加分析得出的属性。形式上，所得 BG 表示为有向图 $G = (V, E)$，其中 V 是节点集合，每个节点表示一条语句；E 是连接节点的边集合，包括控制流边和数据依赖边。每个函数调用节点 $v \in V$ 标注有 $\langle T_v, C_v, U_v, E_v \rangle$ 和 s_v。这里，$T_v = 1$ 表示一个 敏感节点，表明该节点表示敏感 API 调用。$C_v = 1$ 表示一个条件节点，表明该节点可能敏感但需要进一步评估。例如，fs.readSync 因其可能泄露数据而被视为敏感，但实际风险取决于使用上下文，例如其参数和正在访问的数据。敏感度分数 s_v 仅在 $T_v = 1$ 或 $C_v = 1$ 时定义，并量化每个敏感 API 的风险级别。$U_v = 1$ 表示一个 未知节点，意味着该调用无法静态解析；$E_v = 1$ 表示一个 *eval*节点，该节点通过 eval 执行动态生成的代码。

为了跟踪对象引用并维护别名关系，我们在遍历过程中主要分析四类语句，即**Require、属性访问、赋值、函数调用**。这四类语句是对象在整个程序中如何被创建、传播和产生别名的核心。

遍历是单遍的。当我们在DDG$^+$中遍历时，我们在每个程序点维护两个映射。

- **标识符映射**（$idMap$）：其键是标识符 id，其值是对应的对象 obj，从而允许通过$obj = idMap[id]$检索与给定标识符关联的对象。它通过在模拟执行期间记录每个标识符所引用的对象来维护别名关系，使我们能够跟踪不同标识符在整个程序中如何指向同一对象。由于标识符的可见性在不同作用域中有所不同，$idMap$ 还维护每个标识符可访问的作用域信息。为了捕获这些可见性关系，它采用基于栈的结构，在遍历进入或退出代码作用域（例如函数入口和出口）时更新可用标识符。具体而言，$idMap$ 在左侧为单个标识符的赋值语句处更新（Sec. III-B4）。

- **属性映射**（$propMap$）：其键为对象-属性对$\langle obj,\ prop \rangle$（表示对象 obj的属性 $prop$ ），值为对应的目标对象，从而可通过$obj_{ref} = propMap[\langle obj, prop \rangle]$检索给定对象-属性对所引用的对象。它跟踪对象的每个属性所引用的对象，从而便于分析基于属性的引用。具体而言，$propMap$ 在对对象的属性进行赋值时更新（第III-B4节）。

对于每个对象，我们维护两个属性。

- **限定对象路径**（$qPath$）：它记录通向对象自身首次定义的分层属性访问序列。$qPath$ 表示为：

$$qPath = \langle root, props \rangle, \text{ with } props = [p_1, ..., p_n]$$

423
uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

where $root$ denotes the original root object, which is the initial object in the chain of property accesses, and $props$ is the list of properties accessed in order; e.g., the function object of `os.userInfo` has a $qPath$ of $\langle obj_{os}, [userInfo]\rangle$, where obj_{os} is the object referred to by the identifier os.

- **Qualified Object Name** ($qName$): It is a string that represents the object's full qualified name, and is used for constructing the full qualified name of an API call; e.g., the builtin object `fs` has a $qName$ of "fs", while the function object `os.userInfo` has a $qName$ of "os.userInfo".

Before our traversal, we pre-build an object for every builtin module [26] by initializing its $qName$ to the module's name and its $qPath$ to $\langle obj_{builtin}, \phi\rangle$ where $obj_{builtin}$ is the builtin module object. These objects are stored in a set $\mathcal{B}$ for subsequent analysis. This ensures that each builtin module corresponds to a unique object during our traversal.

2) Require: The `require` statement is used to import builtin, local or external modules, and can be defined as:

$$\text{require}(module)$$

where $module$ is an expression representing the name or path of the module. This expression can be a string literal, variable, or computed value. We handle it by the following two cases.

1) **Builtin module.** If $module$ is a string literal and matches against a Node.js builtin module, we retrieve the corresponding pre-built object $obj_{builtin}$ from the set $\mathcal{B}$.
2) **Non-builtin or dynamically computed module.** If $module$ is either a string literal not in the builtin list, or a dynamically computed value (i.e., the module name is computed at runtime), we create a new module object obj_{new}, set its $qName$ to ϕ, and initialize its $qPath$ as $\langle obj_{new}, \phi\rangle$.

By assigning qualified names exclusively to builtin modules, we ensure accurate resolution of builtin API calls. For other modules, the qualified name is left empty (ϕ), allowing the identification of unresolved call sites in subsequent dynamic analysis. In all cases, the value returned by `require` is treated as an object and used in subsequent procedures.

3) Property Access: Property access plays a fundamental role in interacting with objects, and occurs primarily in two forms, e.g., `a.b.c` and `a[b][c]`. In both cases, `a` is the leftmost identifier, while `b` and `c` represent literal property names. When a property, such as `b` or `c` in `a[b][c]`, is not a literal, we use a wildcard symbol $\perp$ to denote the non-literal property.

For every property access expression e, we represent it as $e = \langle id_0, [p_1, \ldots, p_n]\rangle$, where id_0 is the leftmost identifier and $[p_1, \ldots, p_n]$ is the left-to-right sequence of property names. For example, the expression `a.b.c` is represented as $\langle a, [b, c]\rangle$, and `d[e]` as $\langle d, [e]\rangle$. To handle a nested expression, which reflects multi-level property access, we decompose e into a sequence of atomic steps:

$$\langle id_0, [p_1, \ldots, p_n]\rangle \Rightarrow \langle id_1, [p_2, ..., p_n]\rangle \Rightarrow \cdots \Rightarrow \langle id_{n-1}, [p_n]\rangle,$$
$$\text{with } id_j = \langle id_{j-1}, p_j\rangle$$

where id_j is an identifier that refers to the object obtained by accessing property p_j of id_{j-1}, and is used as the base identifier in the next atomic step.

其中 $root$ 表示原始根对象，即属性访问链中的初始对象，$props$ 是按顺序访问的属性列表；例如，os.userInfo的函数对象具有 $qPath$ 为 $\langle obj_{os}, [userInfo]\rangle$，其中$obj_{os}$ 是标识符 os所引用的对象。

- **限定对象名称**（$qName$）：它是一个表示对象完整限定名的字符串，用于构造API调用的完整限定名；例如，内置对象fs的 $qName$ 为“fs”，而函数对象os.userInfo 的 $qName$ 为“os.userInfo”。

在遍历之前，我们为每个内置模块预构建一个对象 [26]，将其 $qName$ 初始化为模块名称，将其 $qPath$ 初始化为 $\langle obj_{builtin}, \phi\rangle$，其中 $obj_{builtin}$ 是内置模块对象。这些对象存储在一个集合 $\mathcal{B}$ 中，以供后续分析。这确保了在我们的遍历过程中，每个内置模块都对应一个唯一对象。

2) require：require 语句用于导入内置模块、本地模块或外部模块，可以定义为：

$$\text{require}(module)$$

其中 $module$是表示模块名称或路径的表达式。该表达式可以是字符串字面量、变量或计算值。我们通过以下两种情况进行处理。

1) **内置模块**。如果 $module$是字符串字面量且与某个 Node.js 内置模块匹配，我们从集合 $\mathcal{B}$中检索对应的预构建对象 $obj_{builtin}$。

2) **非内置模块或动态计算的模块**。如果 $module$是不在内置列表中的字符串字面量，或是动态计算的值（即模块名称在运行时计算），我们创建一个新的模块对象obj_{new}，将其 $qName$ 设置为 ϕ，并将其 $qPath$ 初始化为 $\langle obj_{new}, \phi\rangle$。

通过仅将限定名分配给内置模块，我们确保准确解析内置 API 调用。对于其他模块，限定名留空（ϕ），以便在后续动态分析中识别未解析的调用点。在所有情况下，require 返回的值都被视为对象，并用于后续流程。

3) 属性访问：属性访问在与对象交互中起着基础性作用，主要以两种形式出现，例如 a.b.c 和 a[b][c]。在两种情况下，a 是最左侧标识符，而 b 和 c 表示字面量属性名。当某个属性（如 a[b][c],中的 b 或 c）不是字面量时，我们使用通配符 $\perp$ 来表示该非字面量属性。

对于每个属性访问表达式 e，我们将其表示为$e = \langle id_0, [p_1, \ldots, p_n]\rangle$，其中 id_0 是最左侧标识符，$[p_1, \ldots, p_n]$ 是从左到右的属性名序列。例如，表达式 a.b.c 表示为 $\langle a, [b, c]\rangle$，而 d[e] 表示为 $\langle d, [e]\rangle$。为了处理反映多级属性访问的嵌套表达式，我们将 e 分解为一系列原子步骤：

$$\langle id_0, [p_1, \ldots, p_n]\rangle \Rightarrow \langle id_1, [p_2, ..., p_n]\rangle \Rightarrow \cdots \Rightarrow \langle id_{n-1}, [p_n]\rangle,$$
$$\text{with } id_j = \langle id_{j-1}, p_j\rangle$$

其中 id_j 是一个标识符，它指代通过访问 id_{j-1}的属性 p_j 所获得的对象，并在下一个原子步骤中用作基础标识符。

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
下载。适用限制条款。

For each atomic step $id_j = \langle id_{j-1}, p_j \rangle$ in property access, we operate as follows.

1) **Identifier to object mapping.** We retrieve the referenced object obj_{j-1} of the identifier id_{j-1} from $idMap$.
2) **Property lookup.** We obtain the object referenced by the object-property pair $\langle obj_{j-1}, p_j \rangle$ from $propMap$. If such a mapping exists, obj_{pl} is the target object of the lookup.
3) **Qualified path initialization.** If the property lookup fails to find an object of the given object-property pair, possibly because the object-property pair has not been explicitly assigned before, such as originating from a builtin module (e.g., `os.hostname`, whose function object is not pre-assigned, i.e., no explicit object is created for the property before), or the property is $\perp$, we create a new object obj_{new}, and add it to the map by $propMap[\langle obj_{j-1}, p_j \rangle] = obj_{new}$, which would not disrupt the remainder of the analysis based on that object. For obj_{new}, we set its qualified path through concatenation, i.e., $obj_{new}.qPath = \langle obj_{j-1}.qPath.root, obj_{j-1}.qPath.props \parallel p_j \rangle$, where $\parallel$ denotes appending property p_j to the existing sequence $obj_{j-1}.qPath.props$.
4) **Object aliasing.** Let obj_{res} denote the resulting object for this property access. Specifically, $obj_{res} = obj_{pl}$ if the property lookup succeeds, or $obj_{res} = obj_{new}$ if the property lookup fails. Upon the assignment $id_j = obj_{res}$, we update the map by setting $idMap[id_j] = obj_{res}$ (Sec. III-B4).

In each atomic step, we obtain the corresponding referenced object, and set one of its alias to an identifier, which then serves as the leftmost identifier in the un-nested expression. Once all atomic steps are complete, we obtain a resulting object as the result of the property access.

4) Assignment: Assignment is a fundamental operation that establishes or updates the binding between variables, object properties, and values. We represent the assignment as:

$$expr_{lhs} = expr_{rhs}$$

The analysis of an assignment proceeds in two phases: 1) resolving the right-hand side (RHS) expression to an object, and 2) analyzing this object with the left-hand side (LHS) target, which may be an identifier or a property access expression.

1) **Right-hand side resolution.** The goal is to resolve $expr_{rhs}$ to a specific object obj_r based on the type of $expr_{rhs}$ as follows: (1) if $expr_{rhs}$ is a property access expression, we analyze it to obtain the target object (Sec. III-B3); (2) if it is an identifier, we retrieve the referenced object via $idMap$; (3) if it is a string literal, we create a new object, and set its qualified name to the literal value; (4) if it is a `require`, we analyze it to obtain the corresponding object (Sec. III-B2); and (5) if it is a function call, we analyze it to obtain the target object (Sec. III-B5).
2) **Left-hand side binding.** Once obj_r is determined, we analyze $expr_{lhs}$ as follows: (1) if $expr_{lhs}$ is a standalone identifier id, we set $idMap[id] = obj_r$, making id an alias to obj_r; and (2) if $expr_{lhs}$ is a property access, we follow the property access procedure (Sec. III-B3), leaving the last property unresolved to the form $\langle obj_{last}, p_{last} \rangle$, and then update the property map as $propMap[\langle obj_{last}, p_{last} \rangle] = obj_r$.

对于属性访问中的每个原子步骤 $id_j = \langle id_{j-1},\ p_j \rangle$，我们按如下方式操作。

1) **标识符到对象的映射**。我们从 $idMap$ 中检索标识符 id_{j-1} 所引用的对象 obj_{j-1}。

2) **属性查找**。我们从 $propMap$ 中获取对象-属性对 $\langle obj_{j-1}, p_j \rangle$ 所引用的对象。如果存在这样的映射，obj_{pl} 即为该查找的目标对象。

3) **限定路径初始化**。如果属性查找未能找到给定对象-属性对对应的对象，可能是因为该对象-属性对之前未被显式赋值，例如来源于内置模块（例如 os.hostname，其函数对象未被预先赋值，即在属性之前没有为其创建显式对象），或者该属性为 $\perp$，则我们创建一个新对象 obj_{new}，并通过 $propMap[\langle obj_{j-1}, p_j \rangle] = obj_{new}$将其添加到映射中，这不会影响基于该对象的后续分析。对于 obj_{new}，我们通过拼接设置其限定路径，即 $obj_{new}.qPath = \langle obj_{j-1}.qPath.root, obj_{j-1}.qPath.props \parallel p_j \rangle$，其中表示将属性 p_j追加到现有序列 $obj_{j-1}.qPath.props$。

4) **对象别名**。令 obj_{res} 表示此属性访问所得到的对象。具体而言，$obj_{res} = obj_{pl}$ 如果属性查找成功，或 $obj_{res} = obj_{new}$ 如果属性查找失败。在赋值 $id_j = obj_{res}$ 时，我们通过设置 $idMap[id_j] = obj_{res}$ 来更新映射（第 III-B4 节）。

在每个原子步骤中，我们获得相应的被引用对象，并将其一个别名设置为一个标识符，该标识符随后作为未嵌套表达式中的最左侧标识符。一旦所有原子步骤完成，我们便获得一个结果对象，作为属性访问的结果。

4) 赋值：赋值是一项基本操作，用于建立或更新变量、对象属性与值之间的绑定。我们将赋值表示为：

$$expr_{lhs} = expr_{rhs}$$

对赋值的分析分为两个阶段：1) 将右侧（RHS）表达式解析为一个对象，以及 2) 结合左侧（LHS）目标来分析该对象，该目标可以是标识符或属性访问表达式。

1) **右侧解析**。目标是根据 $expr_{rhs}$ 的类型，将 $expr_{rhs}$ 解析为特定对象 obj_r，具体如下：(1) 如果 $expr_{rhs}$ 是属性访问表达式，则对其进行分析以获取目标对象（第 III-B3 节）；(2) 如果它是标识符，则通过 $idMap$ 检索所引用的对象；(3) 如果它是字符串字面量，则创建一个新对象，并将其限定名设置为该字面量值；(4) 如果它是 require，则对其进行分析以获取相应的对象（第 III-B2 节）；(5) 如果它是函数调用，则对其进行分析以获取目标对象（第 III-B5 节）。

2) **左值绑定**。一旦确定 obj_r，我们按如下方式分析 $expr_{lhs}$：(1) 如果 $expr_{lhs}$ 是独立标识符 id，我们设置$idMap[id] = obj_r$，使 id 成为 obj_r的别名；(2) 如果 $expr_{lhs}$ 是属性访问，我们遵循属性访问过程（Sec.III-B3），将最后一个属性保留为未解析形式 $\langle obj_{last}, p_{last} \rangle$，然后更新属性映射为 $propMap[\langle obj_{last}, p_{last} \rangle] = obj_r$。

424
uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

TABLE I: Part of the Sensitive API List

Qualified Full Name	Behavioral Type	ARD
os.hostname	System Information Retrieval	✗
zlib.brotliCompress	Data Compression	✗
child_process.execSync	Command Execution	✓
net.Socket	Network Creation	✗
https.get	GET Request	✗
fs.readSync	File Reading	✓
dns.resolve	DNS resolution	✗
crypto.createCipheriv	Cipher Creation	✗

For example, in the assignment `a.b.c = r`, we resolve up to `a.b` and retain the last property `c`, and update the property map as $propMap[\langle obj_{a.b}, \mathrm{c}\rangle] = obj_r$.

5) Function Call: Calls to user-defined functions (including third-party and custom functions) or builtin APIs are important program behavior. A call expression is represented as:

$$expr_{callee}(expr_{a_1}, ..., expr_{a_n})$$

where $expr_{callee}$ is the callee expression, which can be an identifier, a property access expression, or `require`, etc. $expr_{a_i}$ is an argument expression. Each call node is analyzed as follows.

1) **User-defined function call.** If the CG resolves the callee for $expr_{callee}$, the node is a user-defined function call. An inter-procedural control flow edge is added from the caller to the callee in the BG, and we traverse into the callee. If $expr_{callee}$ is not `require`, formal parameters are bound to actual arguments to track how argument objects propagate. For each formal parameter id_{fp_i}, we record an alias to the object resolved from the corresponding argument expression $expr_{a_i}$, and update $idMap$ accordingly. During the traversal, return statements are handled in a flow-sensitive way. For each return expression $expr_r$, the object resolved from it is used as the function's return value. If there is no return value, a new object is created with its qualified name set to ϕ.
2) **API call.** If the CG cannot resolve the callee, we compute the full name $\mathcal{F}$ of the call for API identification. Specifically, we resolve $expr_{callee}$ to obtain the corresponding object obj_{callee} based on its type, e.g., by retrieving the referenced object from $idMap$ if it is an identifier, or by resolving the property chain if it is a property access. The full name is computed based on $obj_{callee}.qPath$. In detail, given $obj_{callee}.qPath = \langle o_{root}, [p_1, ..., p_n]\rangle$, $\mathcal{F} = o_{root}.qName{+}(.).concat([p_1, ..., p_n])$. Then, we determine whether $\mathcal{F}$ appears in a predefined sensitive API list that is constructed by following prior work [14, 15, 50, 54, 57]. A partial list is shown in Table I, and the full list is available at our website [46]. Each entry in the list includes the API's full name, behavioral category, and an ARD (Arguments & Return Values Dependent) flag indicating whether its sensitivity depends on usage context, i.e., arguments and return values. If $\mathcal{F}$ matches a sensitive API, the call node v is marked as *sensitive* ($T_v = 1$) with a default sensitivity score s_v of 0.5. If the ARD flag is true, the node is marked as *conditional* ($C_v = 1$). Since this is an API call, we do not bind actual arguments to formal parameters. Instead, we record the argument values as part of the call's metadata for

表 I：敏感API列表的一部分

限定全名	行为类型	ARD
os.hostname	系统信息检索	✗
zlib.brotliCompress	数据压缩	✗
child process.execSync	命令执行	✓
net.Socket	网络创建	✗
https.get	GET 请求	✗
fs.readSync	文件读取	✓
dns.resolve	DNS 解析	✗
crypto.createCipheriv	密码创建	✗

例如，在赋值a.b.c = r中，我们解析到a.b并保留最后一个属性c，然后将属性映射更新为$propMap[\langle obj_{a.b},\mathrm{c}\rangle] = obj_r$。

5) 函数调用：对用户定义函数（包括第三方和自定义函数）或内置API的调用是重要的程序行为。调用表达式表示为：

$$expr_{callee}(expr_{a_1}, ..., expr_{a_n})$$

其中 $expr_{callee}$ 是被调用者表达式，可以是标识符、属性访问表达式或require等。 $expr_{a_i}$ 是参数表达式。每个调用节点按如下方式分析。

1) **用户定义函数调用**。如果CG为 $expr_{callee}$解析出被调用者，则该节点为用户定义函数调用。在BG中从调用者到被调用者添加一条过程间控制流边，并遍历进入被调用者。如果$expr_{callee}$ 不是require，则将形式参数绑定到实际参数，以跟踪参数对象的传播方式。对于每个形式参数 id_{fp_i}，我们记录一个指向从对应参数表达式$expr_{a_i}$解析出的对象的别名，并相应地更新$idMap$ 。在遍历过程中，返回语句以流敏感的方式处理。对于每个返回表达式 $expr_r$，从中解析出的对象被用作函数的返回值。如果没有返回值，则创建一个新对象，其限定名设置为 ϕ。

2) API 调用。如果调用图（CG）无法解析被调用者，我们计算该调用的全名 $\mathcal{F}$ 以进行 API 识别。具体而言，我们解析 $expr_{callee}$ ，根据其类型获取相应的对象 obj_{callee} ，例如，如果它是标识符，则从 $idMap$ 中检索所引用的对象；如果它是属性访问，则解析属性链。全名基于 $obj_{callee}.qPath$ 计算得出。详细来说，给定 $obj_{callee}.qPath = \langle o_{root}, [p_1, ..., p_n]\rangle$, ; $o_{root}.qName{+}$ . .$concat$ $[p_1, ..., p_n]$ ()()。

然后，我们判断 $\mathcal{F}$ 是否出现在一个预定义的敏感 API 列表中，该列表是参照先前工作 [14, 15, 50, 54, 57] 构建的。部分列表见表 I，完整列表可在我们的网站 [46] 获取。列表中的每个条目包括该 API 的全名、行为类别，以及一个 ARD（依赖参数和返回值）标志，该标志指示其敏感性是否取决于使用上下文，即参数和返回值。如果 $\mathcal{F}$ 匹配到敏感 API，则调用节点 v 被标记为敏感（$T_v = 1$），默认敏感度分数 s_v 为 0.5。如果 ARD 标志为真，则该节点被标记为条件（$C_v = 1$）。由于这是一次 API 调用，我们不会将实际参数绑定到形式参数。相反，我们将参数值记录为调用元数据的一部分，以供

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
下载。适用限制。

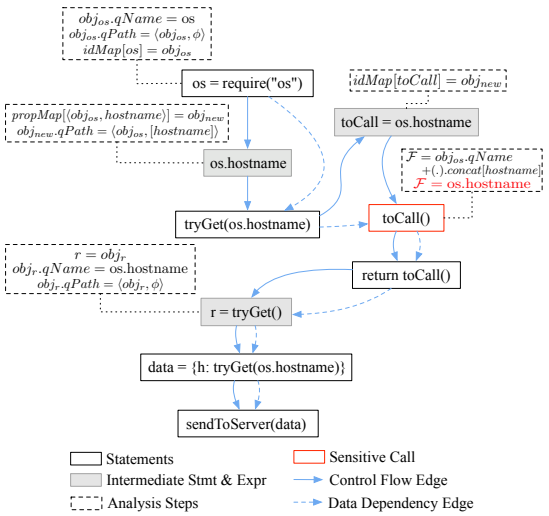


Fig. 6: Behavior Graph of *automation.samples_0.1.15*

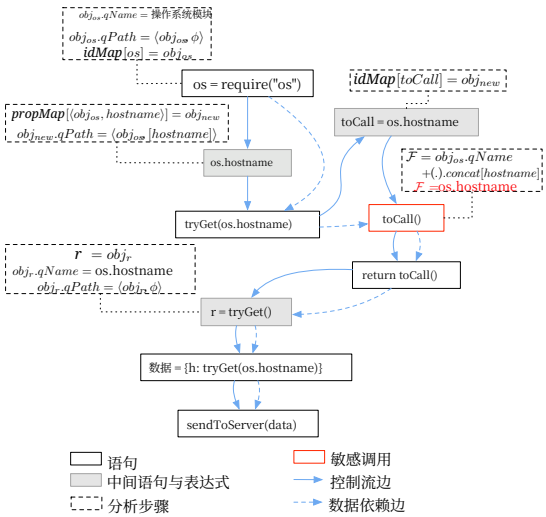


图 6：行为图*automation.samples 0.1.15*

subsequent sensitivity score assessment by the sensitivity evaluator (Section III-D). Finally, a new object is created to represent the return value of the API call, and $\mathcal{F}$ is assigned as its qualified name.

3) **Unknown call.** If the full name $\mathcal{F}$ cannot be statically computed, either because $o_{root}.qName = \phi$ or $qPath$ contains a property that is $\perp$ (e.g., `_ox26cfab(0x1b8)` in the `os[_ox26cfab(0x1b8)]`), the node v is marked as *unknown* ($U_v = 1$). Such unknown calls potentially indicate sensitive API calls or unresolved user-defined function calls. In such cases, we create a new object with its qualified name set to ϕ to represent the unresolved return value of the call, allowing subsequent dynamic analysis to properly resolve it.

4) **Eval.** If $expr_{callee}$ equals to `eval` and its arguments are dynamically generated, we mark the node v as *eval* ($E_v = 1$). We also create a new object for this call, with its qualified name set to ϕ, representing the return value of `eval`.

After analyzing each function call node, the node along with its attributes is added to the BG.

6) An Example of BG Construction: Fig. 6 shows the partial BG generated for the package in Fig. 2. Specifically, we first analyze the assignment `os = require("os")`. Following the process for **Assignment** (Sec. III-B4), we analyze the RHS expression `require("os")` following the process for **Require** (Sec. III-B2), which yields a pre-built object obj_{os} representing the `os` module, with $qName =$ "os" and $qPath = \langle obj_{os}, \phi \rangle$. Then, we map the identifier to the object, $idMap[os] = obj_{os}$.

At the function call `tryGet(os.hostname)`, we follow the process for **Function Call** (Sec. III-B5), resolve the callee via CG, and prepare to analyze it. Before entering the callee, we analyze the argument `os.hostname` following the process for **Property Access** (Sec. III-B3). We retrieve obj_{os} from $idMap$ for the leftmost identifier `os`, and look up the pair $\langle obj_{os}, hostname \rangle$ in $propMap$. Since the function object of `os.hostname` is not pre-assigned, we create a new object obj_{new} with its $qPath = \langle obj_{os}, [hostname] \rangle$.

敏感性评估器随后进行敏感度分数评估（第三节-D）。最后，创建一个新对象来表示该 API 调用的返回值，并将 $\mathcal{F}$ 指定为其限定名。

3) **未知调用**。如果全名 $\mathcal{F}$ 无法被静态计算，这或者是因为 $o_{root}.qName = \phi$ 或 $qPath$ 包含一个为 $\perp$ 的属性（例如，os[_ox26cfab(0x1b8)]中的 _ox26cfab(0x1b8)），那么节点 v 被标记为 未知 ($U_v = 1$)。此类未知调用可能表示敏感 API 调用或未解析的用户定义函数调用。在这种情况下，我们创建一个新对象，其限定名设置为 ϕ，以表示该调用的未解析返回值，从而使后续的动态分析能够正确解析它。4) **Eval**。如果 $expr_{callee}$ 等于 eval 且其参数是动态生成的，我们将节点 v 标记为 *eval* (E_v =1)。我们还会为此调用创建一个新对象，其限定名设置为 ϕ，表示 eval 的返回值。

分析完每个函数调用节点后，该节点连同其属性一起被添加到行为图中。

6) 行为图构建示例：图6展示了为图2中的包生成的部分行为图。具体而言，我们首先分析赋值 os = require("os")。按照**赋值**的处理流程（第III-B4节），我们按照**Require**的处理流程（第III-B2节）分析右侧表达式 require("os")，得到一个预构建对象 obj_{os}，表示os模块，其中 $qName =$ "os" 和 $qPath = \langle obj_{os}, \phi \rangle$。然后，我们将该标识符映射到该对象，$idMap[os] = obj_{os}$。

在函数调用 tryGet(os.hostname) 处，我们遵循**函数调用**的处理流程（第 III-B5 节），通过CG解析被调用者，并准备对其进行分析。在进入被调用者之前，我们按照**属性访问**的处理流程（第 III-B3 节）分析参数 os.hostname。我们从$idMap$ 中检索最左侧标识符 os 对应的 obj_{os}，并在$propMap$中查找键值对 $\langle obj_{os}, hostname \rangle$。由于 os.hostname 的函数对象尚未预先赋值，我们创建一个新对象obj_{new}，其 $qPath = \langle obj_{os}, [hostname] \rangle$。

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

425

权许可使用限制：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore

Then, we set the formal parameter `toCall` as an alias of obj_{new}, and update $idMap$ accordingly (i.e., $idMap[toCall] = obj_{new}$). At the function call `toCall()`, we derive the full name from $obj_{new}.qPath$, which is $\mathcal{F} = obj_{os}.qName + (.).concat[hostname]$ = "os.hostname". As it matches an entry in the sensitive API list, we mark the node as *sensitive*.

If the resulting BG includes *unknown*, *conditional* or *eval* nodes, it is forwarded to our dynamic augmentor (Sec. III-C).

C. Behavior Graph Dynamic Augmentor

Our dynamic augmentor augments the BG by handling nodes marked as *conditional*, *unknown*, or *eval*, and produces an augmented BG by dynamically executing the package. In line with prior work [14], we focus on the installation and import phases, which are known hotspots for malicious code injection [14, 30], and monitor at the Node.js API level during execution, excluding low-level system calls to reduce noise.

During execution, we generate a dynamic call graph (DCG), a sensitive call log (SCL), and an Eval log (EL). The DCG contains call entries $\langle caller, callee\rangle$, where $caller$ is the call statement and $callee$ is the callee's definition location. The SCL contains entries $\langle caller, \langle f, a, r\rangle\rangle$, where f is the full name of the called sensitive API, a is the arguments, and r is the return value. The EL contains entries $\langle caller, code\rangle$, where $code$ is the actual dynamically generated code passed to `eval`. In these three sets, $caller$ includes precise source location information, enabling direct mapping to nodes in the BG and DDG$^+$.

Our dynamic augmentor begins by executing the package from its designated entry file. We instrument the JavaScript runtime to record each function call including the function `eval`, and thereby construct the DCG and EL. Besides, we modify the Node.js framework's builtin module layer by instrumenting each sensitive function to log the $caller$, full name (f), arguments (a), and return value (r); and for asynchronous functions (e.g., `fs.readFile`), we instrument the callback handlers to capture returned data. In this way, we construct the SCL.

Finally, our dynamic analysis augments the BG. For each *conditional* node v ($C_v = 1$), we find a match in the SCL based on the $caller$. If found, we use our sensitivity evaluator (Sec. III-D) to update s_v; otherwise, we keep the default value of 0.5.

For each *unknown* node v ($U_v = 1$), we look for its mappings to DCG's $caller$. If none exists, it indicates that no *unknown* node corresponds to an unresolved user-defined function. Then, we search each *unknown* node in the BG for a match in the SCL, seeking unresolved sensitive API calls. Each matched node v is set to *sensitive* ($T_v = 1$) or *conditional* ($C_v = 1$), and then we update its sensitivity score accordingly; and those unmatched nodes are left unchanged.

If mappings to the DCG's $caller$ exist, it indicates the presence of unresolved user-defined function calls. In this case, each *unknown* node in the BG that does not match an entry in the SCL is mapped to its corresponding $caller$ in the DCG. If multiple *unknown* nodes are mapped, we identify the control-dominant node and regenerate the BG from that point, ensuring flow-sensitive and accurate analysis, especially when new behaviors appear in previously unvisited functions.

然后，我们将形参 toCall 设置为obj_{new}的别名，并相应地更新$idMap$ （即$idMap[toCall]$ =obj_{new}）。在函数调用 toCall() 处，我们从$obj_{new}.qPath$推导出全名，即 $\mathcal{F} = obj_{os}.qName + (.)$ $.concat[hostname]$ = "os.hostname"。由于它与敏感API列表中的条目匹配，我们将该节点标记为敏感。

如果生成的行为图中包含*unknown*、*conditional* 或*eval*节点，则将其转发给我们的动态增强器（第III-C节）。

C. 行为图动态增强器

我们的动态增强器通过处理标记为*conditional*、*unknown*或*eval*的节点来增强行为图，并通过动态执行该包生成增强行为图。与先前工作一致 [14], ，我们关注安装和导入阶段，这些阶段是恶意代码注入的已知热点[14, 30], ，并在执行期间在Node.js API层面进行监控，排除底层系统调用以减少噪声。

在执行过程中，我们生成动态调用图（DCG）、敏感调用日志（SCL）和Eval日志（EL）。DCG包含调用条目 $\langle caller, callee\rangle$，其中 $caller$ 是调用语句， $callee$ 是被调用者的定义位置。SCL包含条目 $\langle caller, \langle f, a, r\rangle\rangle$，其中 f是被调用敏感API的全名，a 是参数， r 是返回值。EL包含条目 $\langle caller, code\rangle$，其中 $code$ 是传递给eval的实际动态生成代码。在这三个集合中， $caller$ 包含精确的源位置信息，从而能够直接映射到BG和DDG$^+$中的节点。

我们的动态增强器从指定的入口文件开始执行包。我们对JavaScript运行时进行插桩，以记录包括eval函数在内的每次函数调用，从而构建DCG和EL。此外，我们修改Node.js框架的内置模块层，对每个敏感函数进行插桩，以记录 $caller$、全名（f）、参数（a）和返回值（r）；对于异步函数（例如fs.readFile），我们对回调处理器进行插桩以捕获返回的数据。通过这种方式，我们构建SCL。

最后，我们的动态分析会增强BG。对于每个条件节点 v （$C_v = 1$），我们基于 $caller$在SCL中查找匹配项。如果找到，我们使用敏感性评估器（第III-D节）来更新 s_v；否则，我们保留默认值0.5。

对于每个未知节点 v （$U_v = 1$），我们查找其到DCG的 $caller$的映射。如果不存在映射，则表明没有未知节点对应于未解析的用户自定义函数。然后，我们在BG中的每个未知节点中搜索SCL中的匹配项，寻找未解析的敏感API调用。每个匹配的节点 v 被设置为敏感（$T_v = 1$）或条件（$C_v = 1$），然后我们相应地更新其敏感度分数；而未匹配的节点则保持不变。

如果存在到 DCG 的 $caller$ 映射，则表明存在未解析的用户定义函数调用。在这种情况下，BG 中每个与 SCL 中条目不匹配的 *unknown* 节点都会映射到其在 DCG 中对应的 $caller$ 。如果映射了多个 *unknown* 节点，我们会识别控制支配节点并从该点重新生成 BG，从而确保流敏感且准确的分析，尤其是当新行为出现在此前未访问过的函数中时。

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

lore下载。适用限制。

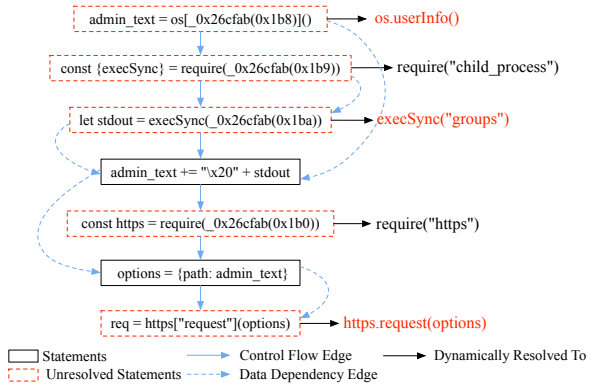


Fig. 7: Augmented BG of *bitsoex_react-design-system_14.1.4*

A similar process applies to dynamically generated code via `eval`. If *eval* nodes in the BG can be mapped to EL entries, we inject the corresponding code for each mapped node. Due to the non-isolated scope of `eval`, the injected code may reference or redefine identifiers from the surrounding context. Therefore, we cannot treat it as an independent unit. When both DCG mappings and EL mappings are available, we identify the control-dominant node that precedes the `eval` statement in the control flow graph. Depending on which context provides better coverage, we regenerate the BG either from this dominant node or from the entry point of the injected code.

Before regeneration, we merge the DCG with the CG to form the merged call graph (MCG), combining dynamic precision with static completeness for more accurate analysis.

During regeneration, we enhance the process in two ways. First, we mark nodes as *sensitive* or *conditional*, and update sensitivity scores accordingly if they appear in the SCL. This includes newly discovered API calls that could not be resolved statically but are found in the traversal of previously unvisited functions in the DCG, ensuring no sensitive calls are missed. Second, we replace the CG with the MCG to add inter-procedural edges, improving accuracy.

Fig. 7 illustrates the augmented BG generated for the package in Fig. 1. Statements outlined with dashed red boxes indicate statically unresolved calls. Our dynamic augmentor resolves these nodes to their corresponding calls, as indicated by the black arrows with sensitive API calls highlighted in red.

D. Sensitivity Evaluator

Our sensitivity evaluator targets sensitive APIs whose ARD flag is set to true (see Table I). For each such API, it uses an LLM to analyze the arguments and return value, guided by a carefully crafted, API-specific prompt. The prompt strategy follows the approach in SocketAI [52]. We reuse their 0–1 security risk scoring scheme along with its descriptive field, recontextualized for our API sensitivity evaluation. For each API type, we design scenario-specific few-shot examples representing high-risk cases to guide the model's reasoning. Each prompt then returns a sensitivity score between 0 and 1, where a higher score indicates a greater risk. For example, for file-reading APIs, we prompt the LLM to analyze both the file

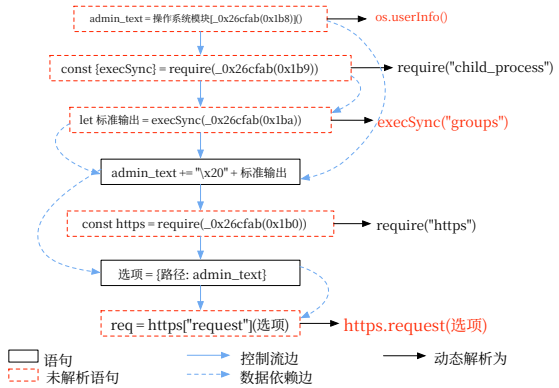


图 7：增强行为图，对象为 *bitsoex react-design-system 14.1.4*

类似的过程也适用于通过 eval 动态生成的代码。如果 BG 中的*eval* 节点可以映射到 EL 条目，我们会为每个映射的节点注入相应的代码。由于 eval 的作用域是非隔离的，注入的代码可能会引用或重定义周围上下文中的标识符。因此，我们不能将其视为独立单元。当 DCG 映射和 EL 映射均可用时，我们会识别控制流图中位于 eval 语句之前的控制支配节点。根据哪个上下文提供更好的覆盖率，我们从该支配节点或注入代码的入口点重新生成 BG。

在重新生成之前，我们将DCG与CG合并，形成合并调用图（MCG），结合动态精确率与静态完整性，以实现更准确的分析。

在重新生成期间，我们通过两种方式增强该过程。首先，我们将节点标记为敏感或条件，如果它们出现在SCL中，则相应地更新敏感度分数。这包括新发现的API调用，这些调用在静态分析中无法解析，但在DCG中遍历先前未访问的函数时被发现，从而确保不会遗漏任何敏感调用。其次，我们将CG替换为MCG以添加过程间边，从而提高准确性。

图7展示了为图1中的包生成的增强行为图（BG）。用红色虚线框标出的语句表示静态未解析的调用。我们的动态增强器将这些节点解析为对应的调用，如黑色箭头所示，其中敏感API调用以红色突出显示。

D. 敏感性评估器

我们的敏感性评估器针对 ARD 标志设为 true 的敏感 API（见表 I）。对于每个此类 API，它使用大语言模型分析参数和返回值，并由精心构建的 API 特定提示引导。提示策略遵循 SOCKETAI [52] 中的方法。我们复用其 0–1 安全风险评分方案及其描述字段，并针对我们的 API 敏感度评估重新进行语境化。对于每种 API 类型，我们设计场景特定的少样本示例来表示高风险案例，以引导模型的推理。每个提示随后返回 0 到 1 之间的敏感度分数，分数越高表示风险越大。例如，对于文件读取 API，我们提示大语言模型分析文件

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

426

权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

path and the file content being read. The LLM is guided by sensitive file paths and instructions to assess potential user- or system-level secrets in the file content. The prompt snippet used for file-reading API sensitivity evaluation is shown in Fig. 8. The remaining prompts used in our sensitivity evaluation are available at our replication website [46].

Prompt Snippet

Task: You are a security expert. You are provided with two inputs:
1. `<File Path>`: A string representing the path of the file being read.
2. `<File Content>`: The actual content read from the file.
Your task is to analyze both the file path and the file content to determine the sensitivity of the file-reading behavior. Based on your expert analysis, assign a sensitivity score as a floating-point number in the range [0, 1].

Sensitivity Scoring:
0: Not sensitive at all.
0 to 0.5: Low sensitivity (serves as a warning).
0.5 to 1: High sensitivity (requires close scrutiny).
1: Absolutely sensitive.

Guidelines: Cases for sensitivity analysis include:
Path-Based Sensitivity:
1. Shell Configuration and History Files: e.g., `.zshrc`, `.bashrc`, `.bash_history`, which may reveal user-specific configurations or command histories.
2. Authentication Files: e.g., `/etc/passwd`, `/etc/shadow`, or files with keywords such as "password" or "auth", which contain sensitive authentication data.
...
Content-Based Sensitivity:
Evaluate whether the file content contains the following information:
1. Credentials: usernames, passwords, tokens, API keys.
2. Cryptographic Material: private keys, certificates, encoded cryptographic data.
...
If the file path does not clearly indicate sensitivity, adjust the score based on the file content. Ensure that your judgment reflects expert security knowledge.

Output: Only return the final sensitivity score (a floating-point number between 0 and 1), with no additional explanation.

Fig. 8: Prompt Snippet for File Reading Sensitivity

E. Maliciousness Detector

Our maliciousness detector is based on GNN, which is effective in various software analysis tasks [1, 21, 55]. We use the Heterogeneous Graph Transformer (HGT) to capture program behavior heterogeneity, incorporating node- and edge-type dependent parameters to model attention over each edge.

SBG Construction. We first construct a suspicious behavior graph (SBG) as an input to the HGT. The SBG is a subgraph of the augmented BG that focuses on sensitive nodes, where $T_v =$ 1, and edges denoting their relations. An edge e exists between two sensitive nodes if they are transitively connected in the augmented BG through either *data dependency* or *control flow*. This means that there exists a path in the augmented BG connecting the nodes, either via a series of data dependencies or control flows, indicating sensitive data or control flows between them.

Meta-Relation Construction. Each edge e in the SBG is represented as a meta-relation tuple $\langle \tau(v_1), \varphi(e), \tau(v_2) \rangle$. Here, $\tau(v)$ denotes the behavioral type of a node v, determined by matching its full name against our sensitive API list, with each API associated with a specific behavioral category. $\varphi(e)$

路径和正在读取的文件内容。大语言模型在敏感文件路径和指令的引导下，评估文件内容中潜在的用户级或系统级机密。用于文件读取 API 敏感度评估的提示片段如图 8 所示。我们敏感度评估中使用的其余提示可在我们的复现网站 [46] 获取。

提示片段

任务：你是一名安全专家。你被提供了两个输入：1. `<文件路径>`：一个表示正在读取的文件路径的字符串。2. `<文件内容>`：从文件中读取的实际内容。你的任务是分析文件路径和文件内容，以确定文件读取行为的敏感性。根据你的专家分析，分配一个在 [0, 1]范围内的浮点数作为敏感度分数。

敏感度评分：0
：N[操作系统模块敏感] iti已到达ll
。0 到 0.5：低敏感度（作为警告）。0.5 到
1：高敏感度（需要仔细审查）。1：绝对敏感。

指南：敏感度分析的适用情形包括：
基于路径的敏感度：
1. Shell 配置和历史文件：例如 .zshrc、.bashrc、.bash_history，这些文件可能泄露用户特定的配置或命令历史记录。
2. 身份验证文件：例如 /etc/passwd、/etc/shadow，或包含"password"或"auth"等关键词的文件，这些文件包含敏感的身份验证数据。
...
基于内容的敏感性：
评估文件内容是否包含以下信息：
1. 凭据：用户名、密码、令牌、API 密钥。
2. 密码学材料：私钥、证书、编码的密码学数据。
...
如果文件路径未明确表明敏感性，则根据以下因素调整评分基于文件内容。确保你的判断体现了专家安全知识。

输出：仅返回最终的敏感度分数（一个浮点数介于0和1之间），不附加额外解释。

图 8：文件读取敏感性提示片段

E. 恶意性检测器

我们的恶意性检测器基于GNN，GNN在各种软件分析任务中有效 [1, 21, 55]。我们使用异构图Transformer（HGT）来捕获程序行为的异质性，融合节点类型和边类型相关的参数，以对每条边建模注意力。

可疑行为图构建。我们首先构建可疑行为图（SBG）作为HGT的输入。SBG是增强行为图（BG）中聚焦于敏感节点的子图，其中 T_v =1，边表示它们之间的关系。若两个敏感节点在增强BG中通过数据依赖或控制流传递相连，则它们之间存在一条边。这意味着增强BG中存在一条连接这些节点的路径，该路径或经由一系列数据依赖，或经由控制流，表明它们之间存在敏感数据或控制流。

元关系构建。SBG 中的每条边 e 都表示为一个元关系元组 $\langle \tau(v_1)\ \varphi(e)\ \tau(v_2) \rangle$。其中，$\tau(v)$ 表示节点 v 的行为类型，通过将其全名与我们的敏感API列表进行匹配来确定，每个 API 都关联到一个特定的行为类别。 $\varphi(e)$

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
426
下载。适用限制。

indicates whether the edge represents a data dependency or control flow. This meta-relation representation enables the HGT to reason over the heterogeneous nature of the SBG.

Before feeding the SBG into the HGT, each node v in the SBG is initialized with a feature vector X_v, which is created by concatenating a type-specific one-hot encoding of the node's behavioral type $\tau(v)$ and its sensitivity score s_v:

$$X_v = [\text{OneHot}(\tau(v)) \| s_v]$$

Graph-Level Classification. After embedding, HGT computes a vector h_v for each of the N nodes in the SBG. To obtain a graph-level representation, we apply weighted average pooling, scaling each node's contribution by its sensitivity score s_v:

$$GlobalRep = \frac{\sum_{v=1}^{N}(s_v \cdot h_v)}{\sum_{v=1}^{N} s_v + 1e^{-9}}$$

The resulting vector $GlobalRep$ is passed through an MLP to output a scalar value $y \in [0, 1]$, denoting the probability of being malicious. If $y > 0.5$, it is considered as malicious.

In the offline training phase, the HGT is trained using binary cross-entropy loss. In the online prediction phase, our detector, based on the trained HGT, outputs a maliciousness prediction given the extracted SBG for an NPM package.

IV. Evaluation

We have implemented ProfMal in 8K lines of Python code. The CPG is generated by Joern [17], and the CG is produced by Jelly [24]. For dynamic analysis, we adopt NodeProf [44], and containerize dynamic execution environment via Docker [5]. LLM-related tasks are handled by DeepSeek-V3 [4], selected for its optimal balance between performance and cost. The GNN classifier is implemented based on HGT [13].

We design four research questions to evaluate ProfMal.

- **RQ1 Effectiveness Evaluation:** How is the effectiveness of ProfMal, compared to state-of-the-art detectors?
- **RQ2 Ablation Study:** How is the contribution of each module to the achieved effectiveness of ProfMal?
- **RQ3 Usefulness Evaluation:** How useful is ProfMal in the real-world detection scenario?
- **RQ4 Efficiency Evaluation:** What is ProfMal's time overhead, and what overhead is brought by dynamic analysis?

A. Evaluation Setup

NPM Dataset. The NPM dataset consists of both malicious and benign packages. The malicious portion is derived from two sources, publicly available datasets and grey literature. Public datasets include Malware Bench [51], Backstabber's Knife Collection [30], MalOSS [6], and OSCAR [57]. Grey literature comprises vulnerability databases (e.g., Snyk [40], GitHub Advisory [8], and OSV [34]) and reports and blogs from sources like Sonatype [42], Phylum [35], etc. The malicious packages from grey literature started from July 2024 to December 2024.

We initially collected a total of 9,247 malicious packages, and deduplicated them first by package name. To further eliminate redundancy caused by code reuse across packages, we performed file-level deduplication via perfect code matching, as

表示该边代表的是数据依赖还是控制流。这种元关系表示使 HGT 能够对 SBG 的异构特性进行推理。

在将SBG输入HGT之前，SBG中的每个节点 v 都用一个特征向量 X_v进行初始化，该特征向量通过拼接节点行为类型 τ（v）的类型特定独热编码及其敏感度分数 s_v而生成：

$$X_v = [\text{OneHot}(\tau(v)) \| s_v]$$

图级分类。在嵌入之后，HGT为SBG中的 N 个节点中的每一个计算一个向量 h_v。为了获得图级表示，我们应用加权平均池化，通过每个节点的敏感度分数 s_v来缩放其贡献：

$$GlobalRep = \frac{\sum_{v=1}^{N}(s_v \cdot h_v)}{\sum_{v=1}^{N} s_v + 1e^{-9}}$$

生成的向量 $GlobalRep$通过MLP输出一个标量值 $y \in [0, 1]$，表示其为恶意的概率。如果 $y > 0.5$，则被认为是恶意的。

在离线训练阶段，HGT使用二元交叉熵损失进行训练。在在线预测阶段，我们的检测器基于训练好的HGT，在给定为NPM包提取的SBG的情况下输出恶意性预测。

IV. 评估

我们使用 8000 行 Python 代码实现了 PROFMAL。CPG 由 Joern [17]，生成，CG 由 Jelly [24] 生成。对于动态分析，我们采用 NodeProf [44]，并通过 Docker [5] 将动态执行环境容器化。LLM 相关任务由 DeepSeek-V3 [4]，处理，选择它是因为其在性能与成本之间达到了最佳平衡。GNN 分类器基于 HGT [13] 实现。

我们设计了四个研究问题来评估 PROFMAL。

- **RQ1 有效性评估：**与最先进检测器相比，PROFMAL 的有效性如何?
- **RQ2 消融研究：**每个模块对 PROFMAL 所达到有效性的贡献如何?
- **RQ3 有用性评估：**PROFMAL 在真实世界检测场景中有多有用?
- **RQ4 效率评估：**PROFMAL 的时间开销是多少，动态分析又带来了多少开销?

A. 评估设置

NPM数据集。NPM数据集由恶意包和良性包组成。恶意部分来源于两个渠道：公开可用的数据集和灰色文献。公开数据集包括 Malware Bench [51]，、Backstabber's Knife Collection [30]，、MALOSS [6],和 OSCAR [57]。灰色文献包括漏洞数据库（如 Snyk [40]，、GitHub Advisory [8]，和 OSV [34]）以及来自 Sonatype [42]，、Phylum [35]，等来源的报告和博客。来自灰色文献的恶意包的时间范围为2024年7月至2024年12月。

我们最初共收集了9,247个恶意软件包，首先按包名进行了去重。为了进一步消除跨包代码复用造成的冗余，我们通过完全代码匹配进行了文件级去重，因为

427
uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

many malicious packages are identical except for their version numbers, resulting in 1,016 packages. We then manually filtered out packages based on the following criteria: (1) packages that contained no actual malicious code (e.g., those created purely as security placeholders or proof-of-concepts); (2) obfuscated packages that could not be dynamically executed; (3) packages whose malicious behavior could not be triggered because it were not embedded in installation or import phases. Regarding (2), we excluded them because non-executable code cannot exhibit malicious behavior or cause any actual harm. After this curation process, we retained 745 malicious packages.

The benign dataset is composed of three sources: (1) the top 5,000 NPM packages by download count, following previous work [15, 54]; (2) the benign packages in Malware Bench [51]; (3) the benign packages in OSCAR [57]. We first ran PROFMAL on all benign packages to construct suspicious behavior graphs involving sensitive APIs. Packages yielding empty graphs, which indicate no observable suspicious behavior, were removed, as they are inherently classified as benign by design and thus provide no value for training. After this filtering, we retained 3,000 benign packages.

Baseline Selection. To compare the effectiveness of PROFMAL, we selected two rule-based detectors GUARDDOG [3] and SPIDERSCAN [15], two learning-based detectors CEREBRO [54] and MALTRACKER [50], and one LLM-based detector SOCKETAI [52]. They were representative in each category. MALTRACKER was implemented with XGBoost, as it achieved the highest F1-score in their experiments with package-level detection. For SOCKETAI, a score above 0.5 in its report indicated a malicious result. In GUARDDOG, detection was flagged as malicious if any alarm was raised. The baselines originally using OpenAI GPT-3.5 were updated to GPT-4o-mini [32].

B. Effectiveness Evaluation (RQ1)

RQ1 Setup. For learning-based detectors, we conducted 10-fold cross-validation. Rule-based and LLM-based detectors were evaluated on the entire dataset, and we reported their average results across the ten folders for a fair comparison. Effectiveness was measured by precision, recall, and F1-score.

Overall Effectiveness Results. Table II shows the effectiveness results. PROFMAL achieves the highest F1-score of 92.4%, outperforms the state-of-the-art detectors by 6.2% to 48.8%, while SOCKETAI is the best state-of-the-art. We also summarize reasons for false positives and false negatives. First, some malicious packages require specific OS platforms and environmental conditions, which causes our BGs to miss sensitive nodes and results in false negatives. Second, when our dynamic analysis cannot satisfy environmental conditions, benign and malicious packages may yield similar BGs without sensitivity scores to differentiate similarities, leading to false positives. Third, both benign and malicious packages include operations like remote downloads and executions, but the underlying system-level behavior analysis is out of our scope, causing false positives.

Summary. PROFMAL outperforms all state-of-the-art detectors on the dataset in terms of precision, recall, and F1-score, with F1-score achieving 6.2% to 48.8% improvement.

许多恶意软件包除版本号外完全相同，最终得到1,016个包。随后我们根据以下标准手动筛选包：（1）不包含实际恶意代码的包（例如仅为安全占位符或概念验证而创建的包）；（2）无法动态执行的混淆包；（3）恶意行为因未嵌入安装或导入阶段而无法触发的包。关于第（2）条，我们排除它们是因为不可执行的代码无法表现出恶意行为或造成任何实际危害。经过此筛选过程，我们保留了745个恶意软件包。

良性数据集由三个来源组成：（1）按下载量排名前5,000的NPM包，遵循先前工作 [15, 54]；（2）Malware Bench [51]中的良性包；（3）OSCAR [57]中的良性包。我们首先对所有良性包运行 PROFMAL，以构建涉及敏感 API 的可疑行为图。生成空图的包（表明无可观察的可疑行为）被移除，因为它们在设计上本质上被归类为良性，因而对训练没有价值。经过此过滤后，我们保留了3,000个良性包。

基线选择。为了比较PROF-MAL的有效性，我们选择了两个基于规则的检测器GUARDDOG [3]和SPIDERSCAN [15]、两个基于学习的检测器CEREBRO[54] 和MALTRACKER [50]，以及一个基于大语言模型的检测器SOCKETAI [52]。它们在各类别中具有代表性。MALTRACKER采用XGBoost实现，因为其在包级检测实验中取得了最高的F1分数。对于SOCKETAI，其报告中分数高于0.5即表示恶意结果。在GUARDDOG中，若触发任何告警则检测被标记为恶意。原本使用OpenAI GPT-3.5的基线已更新为GPT-4o-mini [32]。

B. 有效性评估（*RQ1*）

RQ1设置。对于基于学习的检测器，我们进行了10折交叉验证。基于规则和基于大语言模型的检测器在整个数据集上进行评估，并且我们报告了它们在十个文件夹上的平均结果以进行公平比较。有效性通过精确率、召回率和F1分数来衡量。

整体有效性结果。表II展示了有效性结果。PROFMAL取得了最高的F1分数，达到92.4%，比最先进检测器高出6.2%至48.8%，而SOCKETAI是最佳的最先进检测器。我们还总结了误报和假阴性的原因。首先，一些恶意软件包需要特定的操作系统平台和环境条件，这导致我们的BGs遗漏敏感节点，从而产生假阴性。其次，当我们的动态分析无法满足环境条件时，良性和恶意软件包可能产生相似的BGs，且没有敏感度分数来区分相似性，从而导致误报。第三，良性和恶意软件包都包含远程下载和执行等操作，但底层的系统级行为分析超出了我们的研究范围，从而导致误报。

总结。PROFMAL在精确率、召回率和F1分数方面均优于数据集中所有最先进的检测器，其中F1分数提升了6.2%至48.8%。

427
wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
下载。适用限制。

TABLE II: Results of Effectiveness Evaluation

Tool	Precision	Recall	F1-Score
GUARDDOG	50.8%	79.9%	62.1%
SPIDERSCAN	86.7%	65.8%	74.8%
CEREBRO	82.0%	76.9%	79.1%
MALTRACKER	67.5%	68.4%	67.9%
SOCKETAI	84.2%	89.8%	87.0%
PROFMAL	**92.9%**	**91.8%**	**92.4%**

TABLE III: Results of Ablation Study

Ablated Version	Precision	Recall	F1-Score
PROFMAL	92.9%	91.8%	92.4%
PROFMAL w/o MSC-Detection	91.1%	78.7%	84.3% (↓ 8.8%)
PROFMAL w/o Sensitivity Score	88.5%	82.4%	85.3% (↓ 7.7%)
PROFMAL w/o Dynamic	86.7%	75.6%	80.8% (↓ 12.6%)

C. Ablation Study (RQ2)

RQ2 Setup. We conducted three ablation versions of PROFMAL, i.e., 1) eliminating the malicious shell command detection (MSC-Detection) from our script analyzer; 2) eliminating the sensitivity score of nodes, and treating all nodes as contributing equally; and 3) eliminating our dynamic augmentor.

Overall Ablation Results. Table III presents the results of our ablation study. Eliminating any individual module leads to a drop in overall effectiveness, underscoring the importance of each module. Specifically, eliminating the malicious shell command detection module from our script analyzer results in an 8.8% decrease in F1-score, primarily due to reduced recall. This is because several malicious packages in the dataset inject malicious shell commands during the installation phase.

Eliminating the sensitivity score causes a 7.7% drop in F1-score. The sensitivity score serves as a measure of how security-relevant each node is. Without this score, the HGT struggles to distinguish between benign and malicious behaviors, as the HGT learns from the sensitivity score to weigh the influence of neighboring nodes. Furthermore, in the pooling stage, the sensitivity score determines how much each node contributes to the final graph-level representation. Nodes with higher scores have a stronger impact on the overall embedding, guiding the HGT to focus on potentially malicious operations.

Eliminating our dynamic augmentor causes the largest degradation of 12.6% in F1-score. Dynamic analysis reveals sensitive API calls that static analysis alone may miss, particularly those hidden in the obfuscated code. Dynamic analysis also identifies unresolved third-party library function calls, making our BG more comprehensive. Without dynamic analysis, these crucial behaviors remain hidden, greatly reducing the effectiveness.

Summary. Each module contributes to PROFMAL's overall effectiveness, with dynamic analysis contributing the most by uncovering runtime behaviors that static analysis misses.

D. Usefulness Evaluation (RQ3)

RQ3 Setup. We deployed all the detectors except for SOCKETAI in the real-world to detect newly published NPM packages. SOCKETAI incurred a cost of $161 in our effectiveness evaluation, making it impractical for real-world detection due to the high expense, while PROFMAL incurred a cost of only $18.1. We monitored all newly published NPM packages every five

表 II：有效性评估结果

Tool	精确率	召回率	F1分数
GUARDDOG	50.8%	79.9%	62.1%
SPIDERSCAN	86.7%	65.8%	74.8%
CEREBRO	82.0%	76.9%	79.1%
MALTRACKER	67.5%	68.4%	67.9%
SOCKETAI	84.2%	89.8%	87.0%
PROFMAL	**92.9%**	**91.8%**	**92.4%**

表 III：消融研究结果

消融版本	精确率	召回率	F1分数
PROFMAL	92.9%	91.8%	92.4%
PROFMAL w/o MSC-Detection	91.1%	78.7%	84.3% (↓ 8.8%)
PROFMAL w/o 敏感度分数	88.5%	82.4%	85.3%(↓ 7.7%)
PROFMAL w/o 动态分析	86.7%	75.6%	80.8% (↓ 12.6%)

C. 消融研究（*RQ2*）

RQ2设置。我们进行了三个PROFMAL的消融版本，即：1）从我们的脚本分析器中移除恶意shell命令检测（MSC-Detection）；2）移除节点的敏感度分数，并将所有节点视为贡献相同；3）移除我们的动态增强器。

整体消融结果。表III展示了我们消融研究的结果。消除任何单个模块都会导致整体有效性下降，这凸显了每个模块的重要性。具体而言，从我们的脚本分析器中消除恶意shell命令检测模块会导致F1分数下降8.8%，这主要是由于召回率降低所致。这是因为数据集中的若干恶意软件包在安装阶段注入了恶意shell命令。

消除敏感度分数会导致F1分数下降7.7%。敏感度分数用作衡量每个节点与安全相关程度的指标。没有这个分数，HGT难以区分良性行为和恶意行为，因为HGT从敏感度分数中学习以权衡相邻节点的影响。此外，在池化阶段，敏感度分数决定了每个节点对最终图级表示的贡献程度。分数较高的节点对整体嵌入的影响更强，引导HGT关注潜在恶意操作。

消除我们的动态增强器会导致F1分数最大幅度的下降，达12.6%。动态分析揭示了仅靠静态分析可能遗漏的敏感API调用，尤其是那些隐藏在混淆代码中的调用。动态分析还能识别未解析的第三方库函数调用，使我们的BG更加全面。没有动态分析，这些关键行为将保持隐藏，大幅降低有效性。

总结。每个模块都对PROFMAL的整体有效性有所贡献，其中动态分析贡献最大，因为它揭示了静态分析所遗漏的运行时行为。

D. 有用性评估（*RQ3*）

RQ3 设置。我们在真实环境中部署了除 SOCKETAI 之外的所有检测器，以检测新发布的 NPM 包。SOCKETAI 在我们的有效性评估中产生了 161 美元的成本，由于费用高昂，使其在真实世界检测中不切实际，而 PROFMAL 仅产生了 18.1 美元的成本。我们在 2025 年 2 月至 2025 年 4 月期间每五

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

428

权使用仅限于：成都信息工程大学。于2026年11月02日 05:12:30 UTC 从 IEEE Xplore

TABLE IV: Results of Usefulness Evaluation

Tool	Detected	TP	FP Rate	FN	FN Rate
GUARDDOG	6,948	483	93.0%	46	8.7%
CEREBRO	1726	418	75.8%	111	21.0%
MALTRACKER	429	251	41.5%	278	52.6%
SPIDERSCAN	306	192	37.3%	337	63.7%
PROFMAL	692	**496**	**28.3%**	**33**	**6.2%**

minutes between February 2025 and April 2025, collecting a total of 153,361 packages for real-world detection.

Overall Usefulness Results. Table IV presents the real-world detection results. PROFMAL detected 692 potentially malicious packages, of which 496 were confirmed as malicious after manual review. These packages have been confirmed by NPM and removed. Among all detectors, PROFMAL identified the highest number of malicious packages while achieving the lowest false positive rate of 28.3%. Compared to state-of-the-art detectors, PROFMAL reduced the false positive rate by 24.1% to 70.0%. Besides, as NPM did not disclose a complete list of malicious packages, we used all the malicious packages detected by all detectors to compute false negatives. PROFMAL also achieved the lowest false negative rate of 6.2%. GUARDDOG had the lowest false negative rate among all baselines, but its false positive rate was extremely high. These results demonstrate the practical usefulness of PROFMAL in real-world.

False Positive Analysis. We summarized the causes of false positives. First, PROFMAL regards behaviors such as decompressing and executing binary files using base64, or downloading files via remote network access for execution as malicious. This occurs because PROFMAL focuses on the API level and does not consider low-level system calls. Incorporating tools that analyze system calls can help address this issue. Second, our malicious shell command detector treats the execution of binary files or other executable files as malicious, which leads to false positives. Third, while dynamic analysis helps assign more accurate sensitivity scores to some sensitive APIs, there exist packages that cannot be dynamically executed due to dependencies on specific environmental conditions or the existence of specific local files. In such cases, identical default sensitivity scores are assigned, leading to the misclassifications, especially in packages with extensive process execution operations.

False Negative Analysis. We also summarized the reasons of false negatives. First, PROFMAL focuses on behaviors triggered during installation and import time. Therefore, malicious code that activates only upon user invocation at run-time is out of our scope, directly resulting in false negatives. Second, we miss certain obfuscated malicious packages whose execution relies on either Windows platform or certain conditional checks (e.g., verifying the key of sensitive data), causing no observable behavior in our dynamic analysis. Differently, SPIDERSCAN directly flags obfuscation as malicious, which can find these obfuscated packages but at same time increase its false positives.

Summary. PROFMAL identified 496 new malicious packages in NPM over three months, achieving the lowest false positives and false negatives among state-of-the-arts, demonstrating its practical usefulness in real-world detection.

表 IV：有用性评估结果

Tool	已检测	TP	假阳性率	FN	假阴性率
GUARDDOG	6,948	483	93.0%	46	8.7%
CEREBRO	1726	418	75.8%	111	21.0%
MALTRACKER	429	251	41.5%	278	52.6%
SPIDERSCAN	306	192	37.3%	337	63.7%
PROFMAL	692	**496**	**28.3%**	**33**	**6.2%**

分钟监控所有新发布的 NPM 包，共收集了 153,361 个包用于真实世界检测。

整体有效性结果。表IV展示了真实世界的检测结果。PROFMAL检测出692个潜在恶意包，其中496个经人工审查后确认为恶意。这些包已被NPM确认并移除。在所有检测器中，PROFMAL识别出的恶意包数量最多，同时实现了最低的误报率28.3%。与最先进检测器相比，PROFMAL将误报率降低了24.1%至70.0%。此外，由于NPM未披露完整的恶意包列表，我们使用所有检测器检测到的所有恶意包来计算漏报率。PROFMAL还实现了最低的漏报率6.2%。GUARDDOG在所有基线中漏报率最低，但其误报率极高。这些结果证明了PROFMAL在真实世界中的实际有效性。

误报分析。我们总结了误报的原因。首先，PROFMAL将使用base64解压并执行二进制文件，或通过远程网络访问下载文件以执行等行为视为恶意。出现这种情况是因为PROFMAL关注API层面而未考虑底层系统调用。引入分析系统调用的工具可以帮助解决此问题。其次，我们的恶意shell命令检测器将执行二进制文件或其他可执行文件视为恶意，这导致了误报。第三，虽然动态分析有助于为某些敏感API分配更准确的敏感度分数，但存在一些包由于依赖特定环境条件或特定本地文件的存在而无法被动态执行。在这种情况下，会分配相同的默认敏感度分数，导致误分类，尤其是在具有大量进程执行操作的包中。

漏报分析。我们还总结了漏报的原因。首先，PROFMAL 关注的是在安装和导入时触发的行为。因此，仅在用户调用时于运行时激活的恶意代码不在我们的作用域内，这直接导致了假阴性。其次，我们遗漏了某些混淆恶意软件包，其执行依赖于 Windows平台或某些条件检查（例如，验证敏感数据的密钥），导致在我们的动态分析中没有可观察的行为。不同的是，SPIDERSCAN 直接将混淆标记为恶意，这可以找到这些混淆包，但同时增加了其误报。

总结。PROFMAL在三个月内于NPM中识别出496个新的恶意软件包，在现有最先进方法中实现了最低的误报和假阴性，展示了其在实际检测中的实用性。

428
wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply
ore 下载。适用限制。

TABLE V: Results of Efficiency Evaluation

Tool	Time (s)
GUARDDOG	33.9
SPIDERSCAN	98.6
CEREBRO	26.0
MALTRACKER	53.3
SOCKETAI	143.4
PROFMAL w/o Dynamic	115.7
PROFMAL	179.5 (↑ 55.1%)

E. Efficiency Evaluation (RQ4)

RQ4 Setup. We measured the execution time of each detector as well as the overhead introduced by PROFMAL's dynamic analysis. For SOCKETAI, based on ChatGPT's rate limits [33] and the average file counts of our dataset, we set the number of concurrently analyzed files per package to 10. All detectors ran on an Ubuntu server equipped with an Intel Xeon Silver 4316 CPU, 256 GB of RAM, and an RTX 3090 GPU.

Overall Efficiency Results. Table V shows the time overhead of all detectors. CEREBRO is the fastest, which takes 26.0 seconds per package, followed by rule-based detector GUARDDOG. PROFMAL without dynamic analysis takes 115.7 seconds per package, and incorporating dynamic analysis increases the execution time to 179.5 seconds, introducing a 55.1% overhead. Despite this overhead, PROFMAL effectively reveals dynamic behaviors undetected by purely static analysis, achieving the highest F1-score, and thus is worth the extra time consumption.

Summary. While dynamic analysis increases the time overhead of PROFMAL to 179.5 seconds per package, it uncovers behaviors missed by static analysis, which is worthwhile.

F. Threats and Limitations

Threats. Although dynamic analysis can augment static analysis, not all packages in our dataset were executable. Typical reasons include missing third-party commands in the standard Ubuntu environment, invalid URLs (often taken down after the disclosure of malicious packages), unavailable dependencies (no longer maintained or downloadable), platform constraints, and missing environment variables or user files (e.g., wallets). We only excluded obfuscated packages that failed to execute to ensure runnable experiments, since their analysis almost entirely relies on dynamic evidence. These removed packages might exhibit different malicious behaviors compared to the retained set. Besides, we manually analyzed malicious packages to filter out those whose malicious behaviors were triggered at run-time, this manual step introduced subjectivity, and the filtering may reduce diversity.

Limitations. No detector can be considered as a complete and isolated solution [31]. Likewise, there are some limitations of PROFMAL. First, PROFMAL currently targets behaviors executed during the installation and import time. Although most malicious behaviors occur within these phases, attacks outside this scope will evade detection. Second, despite employing object-sensitive analysis, PROFMAL does not address prototype pollution [39], enabling attackers to bypass detection through prototype manipulation. Third, PROFMAL flags executable files as malicious, affecting both code behavior

表 V：效率评估结果

Tool	时间（秒）
GUARDDOG	33.9
SPIDERSCAN	98.6
CEREBRO	26.0
MALTRACKER	53.3
SOCKETAI	143.4
PROFMAL w/o 动态分析	115.7
PROFMAL	179.5 (↑ 55.1%)

E. 效率评估（*RQ4*）

RQ4 设置。我们测量了每个检测器的执行时间以及PROFMAL动态分析引入的开销。对于SOCKETAI，基于ChatGPT的速率限制 [33]和我们数据集的平均文件数量，我们将每个包并发分析的文件数设置为10。所有检测器均在配备Intel Xeon Silver 4316 CPU、256 GB内存和RTX 3090 GPU的Ubuntu服务器上运行。

整体效率结果。表 V 展示了所有检测器的时间开销。CEREBRO 最快，每个包耗时 26.0 秒，其次是基于规则的检测器 GUARDDOG。不使用动态分析的 PROFMAL 每个包耗时 115.7 秒，而加入动态分析后执行时间增加到 179.5 秒，引入了 55.1% 的开销。尽管存在这一开销，PROFMAL 有效揭示了纯静态分析无法检测到的动态行为，取得了最高的F1分数，因此额外的耗时是值得的。

总结。尽管动态分析将PROFMAL的开销增加至每个包179.5秒，但它能发现静态分析遗漏的行为，这是值得的。

F. 威胁与局限性

威胁。尽管动态分析可以增强静态分析，但我们的数据集中并非所有包都是可执行的。典型原因包括标准Ubuntu环境中缺少第三方命令、无效的URL（通常在恶意包披露后被撤下）、不可用的依赖项（不再维护或无法下载）、平台限制，以及缺少环境变量或用户文件（例如钱包）。我们仅排除了执行失败的混淆包以确保实验可运行，因为对它们的分析几乎完全依赖于动态证据。这些被移除的包可能表现出与保留集不同的恶意行为。此外，我们手动分析了恶意包以过滤掉那些在运行时触发恶意行为的包，这一手动步骤引入了主观性，且过滤可能降低多样性。

局限性。没有任何检测器可以被视为完整且独立的解决方案 [31]。同样，PROFMAL 也存在一些局限性。首先，PROFMAL 目前针对的是在安装和导入时执行的行为。尽管大多数恶意行为发生在这些阶段，但此范围之外的攻击将逃避检测。其次，尽管采用了对象敏感分析，PROFMAL 并未解决原型污染 [39], ，使得攻击者能够通过原型操纵绕过检测。第三，PROFMAL 将可执行文件标记为恶意，同时影响代码行为

429
uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

and shell command analysis, leaving a need for system-level analysis in future. Fourth, while dynamic analysis augments static analysis by capturing runtime behaviors, effectiveness depends on successful execution under Linux, which some packages may not satisfy. This methodological limitation may lead to false positives or false negatives in practice. Finally, PROFMAL relies on third-party tools, inaccuracies in these tools may propagate into our analysis.

V. RELATED WORK

Several empirical studies [11, 20, 30, 56, 58, 59] have been conducted to understand malicious packages in various ecosystems. These studies provide good insights on designing detectors. Various approaches have also been developed to detect malicious NPM packages, and these tools can be broadly categorized into four types, rule-based [3, 6, 9, 15, 22, 23, 36, 45, 49, 53, 57], learning-based [7, 12, 14, 19, 25, 28, 29, 38, 50, 54], LLM-based [52], and differential testing-based [37, 41].

Rule-based approaches rely on predefined rules to analyze various package aspects, e.g., metadata [6, 45, 53], static behaviors [3, 6, 22, 23], and runtime behaviors [6]. MALOSS [6] combines metadata, static behaviors, and dynamic behaviors. Several protection tools [36, 49] defend malicious behaviors at runtime, but are not originally designed for malicious package detection. While rule-based approaches are simple, they suffer high false positives. To mitigate this, SPIDERSCAN [15] refines sensitive APIs into fine-grained types, builds behavior graphs, and performs dynamic analysis on targeted APIs to confirm maliciousness; and OSCAR [57] generates API behavior sequences dynamically, and applies enhanced system-level rules.

Learning-based approaches leverage metadata [12, 38], static behaviors [19, 38, 50], and dynamic behaviors [25] as features to learn a maliciousness classifier. Differently, Zhang et al. [54] and Huang et al. [14] propose to use API sequences to learn the behavior. Ohm et al. [28] evaluate the performance of various machine learning models in malicious packages detection. Recently, Zahan et al. [52] introduce SOCKETAI to investigate the potential of LLMs for malicious code detection.

These approaches suffer the limitation of imprecise modeling of program behavior as summarized in Sec. I and motivated in Sec. II. It is also worth mentioning that differential testing-based approaches [37, 41] identify discrepancies between source code and distributed artifacts or between updates. In contrast, PROFMAL only analyzes the source code of individual packages.

VI. CONCLUSIONS

We have presented PROFMAL, a malicious NPM package detector that learns from unified behavior graphs by synergy between static and dynamic analysis. Our experiments have demonstrated its effectiveness and usefulness, with 496 new malicious packages detected. In the future, we plan to introduce system-level analysis to further improve its effectiveness.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

和 Shell 命令分析，未来仍需要系统级分析。第四，虽然动态分析通过捕获运行时行为来增强静态分析，但其有效性取决于在 Linux 下成功执行，而某些包可能无法满足这一条件。这一方法学局限在实践中可能导致误报或假阴性。最后，PROFMAL 依赖第三方工具，这些工具的不准确性可能会传播到我们的分析中。

V. 相关工作

已有若干实证研究 [11, 20, 30, 56, 58, 59]用于理解各种生态系统中的恶意包。这些研究为设计检测器提供了良好的见解。多种方法也已被开发用于检测恶意NPM包，这些工具可大致分为四类：基于规则的 [3, 6, 9, 15, 22, 23, 36, 45, 49, 53, 57], 、基于学习的 [7, 12, 14, 19, 25, 28, 29, 38, 50, 54],、基于大语言模型的 [52], 以及基于差分测试的 [37, 41]。

基于规则的方法依赖预定义规则来分析包的各个方面，例如元数据 [6, 45, 53], 静态行为 [3, 6, 22, 23], 和运行时行为 [6]。MALOSS [6]结合了元数据、静态行为和动态行为。一些防护工具 [36, 49] 在运行时防御恶意行为，但最初并非为恶意包检测而设计。虽然基于规则的方法简单，但它们存在较高的误报率。为缓解这一问题，SPIDERSCAN [15] 将敏感API细化为细粒度类型，构建行为图，并对目标API执行动态分析以确认恶意性；OSCAR [57] 动态生成API行为序列，并应用增强的系统级规则。

基于学习的方法利用元数据 [12, 38], 静态行为 [19, 38, 50], 和动态行为 [25] 作为特征来学习恶意性分类器。不同的是，Zhang 等 [54]和Huang等[14] 提出使用API序列来学习行为。Ohm 等 [28] 评估了多种机器学习模型在恶意包检测中的性能。最近，Zahan等 [52] 引入了SOCKETAI来研究大语言模型在恶意代码检测方面的潜力。

这些方法存在对程序行为建模不精确的局限性，如第I节所总结并在第II节中阐述的动机。同样值得一提的是，基于差分测试的方法[37, 41] 识别源代码与分发制品之间或更新之间的差异。相比之下，PROFMAL仅分析单个包的源代码。

VI. 结论

我们提出了PROFMAL，一种恶意NPM包检测器，它通过静态与动态分析的协同从统一行为图中学习。我们的实验已证明其有效性和实用性，共检测出496个新的恶意软件包。未来，我们计划引入系统级分析以进一步提高其有效性。

致谢

本工作得到国家自然科学基金（资助号：62332005和62372114）的支持。

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

下载。适用限制。

REFERENCES

[1] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu, "Coca: improving and explaining graph neural network-based vulnerability detection systems," in *Proceedings of the 46th International Conference on Software Engineering*, 2024.

[2] Y. Cao, S. Wu, R. Wang, B. Chen, Y. Huang, C. Lu, Z. Zhou, and X. Peng, "Recurring vulnerability detection: How far are we?" in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2025.

[3] DataDog. (2022) Guarddog. [Online]. Available: https://github.com/DataDog/guarddog

[4] deepseek ai. (2024) Deepseek-v3. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-V3

[5] Docker. (2013) Docker: Accelerated container application development. [Online]. Available: https://www.docker.com/

[6] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium*, 2020.

[7] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Detecting suspicious package updates," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, 2019, pp. 13–16.

[8] GitHub. (2022) Github advisory database. [Online]. Available: https://github.com/github/advisory-database

[9] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, "Anomalicious: Automated detection of anomalous and potentially malicious commits on github," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, 2021, pp. 258–267.

[10] D. Grander. (2018) A post-mortem of the malicious event-stream backdoor. [Online]. Available: https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/

[11] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 166–177.

[12] S. Halder, M. Bewong, A. Mahboubi, Y. Jiang, M. R. Islam, M. Z. Islam, R. H. Ip, M. E. Ahmed, G. S. Ramachandran, and M. Ali Babar, "Malicious package detection using metadata information," in *Proceedings of the ACM on Web Conference*, 2024, p. 1779–1789.

[13] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of the 29th web conference*, 2020.

[14] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "Donapi: Malicious npm packages detector using behavior sequence knowledge mapping," in *Proceedings of the 33rd USENIX Security Symposium*, 2024.

[15] Y. Huang, R. Wang, W. Zheng, Z. Zhou, S. Wu, S. Ke, B. Chen, S. Gao, and X. Peng, "Spiderscan: Practical detection of malicious npm packages based on graph-based behavior modeling and matching," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1146–1158.

[16] D. Jayasuriya, V. Terragni, J. Dietrich, S. Ou, and K. Blincoe, "Understanding breaking changes in the wild," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1433–1444.

[17] Joern. (2019) Joern - the bug hunter's workbench. [Online]. Available: https://joern.io/

[18] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *Proceedings of the 44th IEEE Symposium on Security and Privacy*, 2023, pp. 1509–1526.

[19] P. Ladisa, S. E. Ponta, N. Ronzoni, M. Martinez, and O. Barais, "On the feasibility of cross-language detection of malicious packages in npm and pypi," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 71–82.

[20] P. Ladisa, M. Sahin, S. E. Ponta, M. Rosa, M. Martinez, and O. Barais, "The hitchhiker's guide to malicious third-party dependencies," in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 2023, pp. 65–74.

[21] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020.

[22] N. Li, S. Wang, M. Feng, K. Wang, M. Wang, and H. Wang, "Malwukong: Towards fast, accurate, and multilingual detection of malicious code

参考文献

[1] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu, "Coca：改进和解释基于图神经网络的漏洞检测系统，"发表于第46届软件工程国际会议论文集，2024年。[2] Y. Cao, S. Wu, R. Wang, B. Chen, Y. Huang, C. Lu, Z. Zhou, and X. Peng, "重复性漏洞检测：我们进展如何？"发表于第34届ACM SIGSOFT软件测试与分析国际研讨会论文集，2025年。[3] DataDog。（2022）Guarddog。 [在线]。可访问：https://github.com/DataDog/guarddog [4] deepseek ai。（2024）Deepseek-v3。 [在线]。可访问：https://github.com/deepseek-ai/DeepSeek-V3 [5] 2013 Docker。（）Docker：加速容器应用开发。[在线]。可访问：https://www.docker.com/ [6] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "面向衡量解释型语言包管理器上的供应链攻击，"发表于第28届年度网络与分布式系统安全研讨会论文集，2020年。[7] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. K¨astner, "检测可疑包更新，"发表于第41届国际软件工程会议：新思想与新兴成果论文集，2019年，第13–16页。[8] GitHub。（2022）GitHub咨询数据库。 [在线]。可访问：https://github.com/github/advisory-database [9] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Sch¨afer, "Anomalicious：在GitHub上自动检测异常且可能恶意的提交，"发表于第43届国际软件工程会议：软件工程实践论文集，2021年，第258–267页。[10] D. Grander。（2018）恶意event-stream后门的事后分析。 [在线]。可访问：https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/ [11] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "PyPI生态系统中恶意代码的实证研究，"发表于第38届IEEE/ACM自动化软件工程国际会议论文集，2023年，第166–177页。[12] S. Halder, M. Bewong, A. Mahboubi, Y. Jiang, M. R. Islam, M. Z. Islam, R. H. Ip, M. E. Ahmed, G. S. Ramachandran, and M. Ali Babar, "利用元数据信息进行恶意包检测，"发表于ACM Web会议论文集，2024年，第1779–1789页。[13] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "异构图Transformer，"发表于第29届Web会议论文集，2020年。[14] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "Donapi：使用行为序列知识映射的恶意npm包检测器，"发表于第33届USENIX安全研讨会论文集，2024年。[15] Y. Huang, R. Wang, W. Zheng, Z. Zhou, S. Wu, S. Ke, B. Chen, S. Gao, and X. Peng, "Spiderscan：基于图的行为建模与匹配的恶意npm包实用检测，"发表于第39届IEEE/ACM自动化软件工程国际会议论文集，2024年，第1146–1158页。[16] D. Jayasuriya, V. Terragni, J. Dietrich, S. Ou, and K. Blincoe, "理解实际场景中的破坏性变更，"发表于第32届ACM SIGSOFT软件测试与分析国际研讨会论文集，2023年，第1433–1444页。[17] Joern。（2019）Joern - 漏洞猎手工作台。 [在线]。可访问：https://joern.io/ [18] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK：开源软件供应链攻击分类体系，"发表于第44届IEEE安全与隐私研讨会论文集，2023年，第1509–1526页。[19] P. Ladisa, S. E. Ponta, N. Ronzoni, M. Martinez, and O. Barais, "关于在npm和PyPI中对恶意包进行跨语言检测的可行性，"发表于第39届年度计算机安全应用会议论文集，2023年，第71–82页。[20] P. Ladisa, M. Sahin, S. E. Ponta, M. Rosa, M. Martinez, and O. Barais, "恶意第三方依赖的搭车者指南，"发表于2023年软件供应链攻防研究与生态系统防御研讨会论文集，2023年，第65–74页。[21] A. LeClair, S. Haque, L. Wu, and C. McMillan, "通过图神经网络改进代码摘要，"发表于第28届程序理解国际会议论文集，2020年。[22] N. Li, S. Wang, M. Feng, K. Wang, M. Wang, and H. Wang, "Malwukong：面向快速、准确且多语言的恶意代码检测"

uthorized licensed use limited to: Chengdu University of Information Technology. Downl

权使用仅限：成都信息工程大学。于2026年11月02日05:12:30 UTC从IEEE Xplore下载

poisoning in oss supply chains," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1993–2005.
[23] Microsoft. (2020) Oss detect backdoor. [Online]. Available: https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor
[24] A. Møller. (2020) Javascript/typescript static analyzer for call graph construction, library usage pattern matching, and vulnerability exposure analysis. [Online]. Available: https://github.com/cs-au-dk/jelly
[25] T.-C. Nguyen, D.-L. Vu, and N. C. Debnath, "Classifying benign and malicious open-source packages using machine learning based on dynamic features." *International Journal of Computers and Their Applications*, p. 293, 2024.
[26] Node.js. (2014) Node.js documentation. [Online]. Available: https://nodejs.org/api/
[27] NPM. (2014) Node package manager (npm). [Online]. Available: https://www.npmjs.com/
[28] M. Ohm, F. Boes, C. Bungartz, and M. Meier, "On the feasibility of supervised machine learning for the detection of malicious software packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–10.
[29] M. Ohm, L. Kempf, F. Boes, and M. Meier, *Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors*. Gesellschaft für Informatik, Bonn, 2022.
[30] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2020, pp. 23–43.
[31] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, 2023, pp. 1–11.
[32] OpenAI. (2024) Gpt-4o-mini. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/
[33] ——. (2024) Rate limits. [Online]. Available: https://platform.openai.com/docs/guides/rate-limits
[34] OSV. (2022) A distributed vulnerability database for open source. [Online]. Available: https://osv.dev/
[35] phylum. (2020) Phylum.io — see phylum research blog & insights. [Online]. Available: https://blog.phylum.io/
[36] T. Pohl, M. Ohm, F. Boes, and M. Meier, "You can run but you can't hide: Runtime protection against malicious package updates for node.js," in *Sicherheit, Schutz und Zuverlässigkeit: Konferenzband der 12. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), Sicherheit 2024, Worms, Germany, April 9-11, 2024*, 2024, pp. 231–241.
[37] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci, "On the feasibility of detecting injections in malicious npm packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–8.
[38] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1681–1692.
[39] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node. js," in *Proceedings of the 32th USENIX Security Symposium*, 2023, pp. 5521–5538.
[40] snyk. (2023) Snyk vulnerability database. [Online]. Available: https://security.snyk.io/
[41] R. J. Sofaer, Y. David, M. Kang, J. Yu, Y. Cao, J. Yang, and J. Nieh, "Rogueone: Detecting rogue updates via differential data-flow analysis using trust domains," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
[42] sonatype. (2008) Sonatype: Software supply chain management. [Online]. Available: https://www.sonatype.com/
[43] Sonatype. (2024) 2024 state of the software supply chain. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/introduction
[44] H. Sun. (2019) Instrumentation framework for node.js compliant to ecmascript 2020 based on graalvm. [Online]. Available: https://github.com/Haiyang-Sun/nodeprof.js/
[45] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending against package typosquatting," in *Proceedings of the 14th International Conference on Network and System Security*, 2020, pp. 112–131.
[46] PROFMAL. (2025) PROFMAL. [Online]. Available: https://github.com/yiheng98/ProfMal

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

开源软件供应链中的投毒问题，"见2023年第38届IEEE/ACM自动化软件工程国际会议（ASE）论文集，2023年，第1993–2005页。[23] 微软。（2020）OSS后门检测。 [在线]。可访问：https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor [24] A. Møller。（2020）用于调用图构建、库使用模式匹配和漏洞暴露分析的JavaScript/TypeScript静态分析器。 [在线]。可访问：https://github.com/cs-au-dk/jelly [25] T.-C. Nguyen、D.-L. Vu和N. C. Debnath，"基于动态特征使用机器学习对良性及恶意开源包进行分类。"国际计算机及其应用期刊，第293页，2024年。[26] Node.js。（2014）Node.js文档。 [在线]。可访问：https://nodejs.org/api/ [27] NPM。（2014）Node包管理器（npm）。 [在线]。可访问：https://www.npmjs.com/ [28] M. Ohm、F. Boes、C. Bungartz和M. Meier，"关于监督式机器学习检测恶意软件包可行性的研究，"见第17届可用性、可靠性与安全性国际会议论文集，2022年，第1–10页。[29] M. Ohm、L. Kempf、F. Boes和M. Meier，通过恶意行为者的代码复用来检测恶意软件包。Gesellschaft f¨ur Informatik，波恩，2022年。[30] M. Ohm、H. Plate、A. Sykosch和M. Meier，"背刺者之刀收藏：开源软件供应链攻击综述，"见第17届入侵与恶意软件检测及漏洞评估国际会议论文集，2020年，第23–43页。[31] M. Ohm和C. Stuke，"SoK：软件供应链攻击的实用检测，"见第18届可用性、可靠性与安全性国际会议论文集，2023年，第1–11页。[32] OpenAI。（2024）GPT-4o-mini。 [在线]。可访问：
h
t
t
p
s://o
penai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ [33] ——。（2024）速率限制。 [在线]。可访问：https://platform.openai.com/docs/guides/rate-limits [34] OSV。（2022）面向开源软件的分布式漏洞数据库。[在线]。可访问：https://osv.dev/ [35] Phylum。（2020）Phylum.io — 参见Phylum研究博客与洞察。[在线]。可访问：https://blog.phylum.io/ [36] T. Pohl、M. Ohm、F. Boes和M. Meier，"你可以运行但无法隐藏：针对Node.js恶意包更新的运行时保护，"见Sicherheit, Schutz und Zuverl¨assigkeit：Gesellschaft f¨ur Informatik e.V.（GI）安全分会第12届年会会议论文集，Sicherheit 2024，德国沃尔姆斯，2024年4月9-11日，2024年，第231–241页。[37] S. Scalco、R. Paramitha、D.-L. Vu和F. Massacci，"关于检测恶意npm包中注入的可行性研究，"见第17届可用性、可靠性与安全性国际会议论文集，2022年，第1–8页。[38] A. Sejfia和M. Sch¨afer，"恶意npm包的实用自动化检测，"见第44届软件工程国际会议论文集，2022年，第1681–1692页。[39] M. Shcherbakov、M. Balliu和C.-A. Staicu，"寂静的春天：原型污染导致Node.js中的远程代码执行，"见第32届USENIX安全研讨会论文集，2023年，第5521–5538页。[40] Snyk。（2023）Snyk漏洞数据库。 [在线]。可访问：https://security.snyk.io/ [41] R. J. Sofaer、Y. David、M. Kang、J. Yu、Y. Cao、J. Yang和J. Nieh，"RogueOne：使用信任域通过差分数据流分析检测恶意更新，"见第46届IEEE/ACM软件工程国际会议论文集，2024年，第1–13页。[42] Sonatype。（2008）Sonatype：软件供应链管理。 [在线]。可访问：https://www.sonatype.com/ [43] Sonatype。（2024）2024年软件供应链现状。 [在线]。可访问：
h
t
t
p
s://
www.sonatype.com/state-of-the-software-supply-chain/introduction [44] H. Sun。（2019）基于GraalVM的符合ECMAScript 2020的Node.js插桩框架。 [在线]。可访问：https://github.com/Haiyang-Sun/nodeprof.js/ [45] M. Taylor、R. Vaidya、D. Davidson、L. De Carli和V. Rastogi，"防御包误植域名，"见第14届网络与系统安全国际会议论文集，2020年，第112–131页。[46] PROFMAL。（2025）PROFMAL。 [在线]。可访问：https://github.com/yiheng98/ProfMal

430

下载。适用限制条款。

[47] D. Trizna, “Shell language processing: Unix command parsing for machine learning,” *arXiv preprint arXiv:2107.02438*, 2021.
[48] L. Valentić. (2023) Typosquatting campaign delivers r77 rootkit via npm. [Online]. Available: https://www.reversinglabs.com/blog/r77-rootkit-typosquatting-npm-threat-research
[49] E. Wyss, A. Wittman, D. Davidson, and L. De Carli, “Wolf at the door: Preventing install-time attacks in npm with latch,” in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1139–1153.
[50] Z. Yu, M. Wen, X. Guo, and H. Jin, “Maltracker: A fine-grained npm malware tracker copiloted by llm-enhanced dataset,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1759–1771.
[51] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, “Malwarebench: Malware samples are not enough,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 728–732.
[52] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. A. Williams, “Leveraging large language models to detect npm malicious packages,” in *Proceedings of the 47th International Conference on Software Engineering*, 2025.
[53] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, “What are weak links in the npm supply chain?”

[47] D. Trizna，“Shell语言处理：面向机器学习的Unix命令解析，”*arXiv*预印本 *arXiv:2107.02438*，2021年。[48] L. Valenti´c。（2023）域名抢注活动通过npm投递r77 rootkit。 [在线]。可访问：
h
t
t
p
s://
www.reversinglabs.com/blog/r77-rootkit-typosquatting-npm-threat-research [49] E. Wyss、A. Wittman、D. Davidson和L. De Carli，“门口的狼：使用latch防止npm中的安装时攻击，”见*ACM*亚洲计算机与通信安全会议论文集，2022年，第1139–1153页。[50] Z. Yu、M. Wen、X. Guo和H. Jin，“MALTRACKER：由大语言模型增强数据集辅助的细粒度npm恶意软件追踪器，”见第*33*届*ACM SIGSOFT*软件测试与分析国际研讨会论文集，2024年，第1759–1771页。[51] N. Zahan、P. Burckhardt、M. Lysenko、F. Aboukhadijeh和L. Williams，“Malwarebench：恶意软件样本还不够，”见第*21*届挖掘软件仓库国际会议论文集，2024年，第728–732页。[52] N. Zahan、P. Burckhardt、M. Lysenko、F. Aboukhadijeh和L. A. Williams，“利用大语言模型检测npm恶意包，”见第*47*届软件工程国际会议论文集，2025年。[53] N. Zahan、T. Zimmermann、P. Godefroid、B. Murphy、C. Maddila和L. Williams，“npm供应链中的薄弱环节是什么？”

431

uthorized licensed use limited to: Chengdu University of Information Technology. Downl
权使用仅限：成都信息工程大学。于2026年11月2日05:12:30 UTC从IEEE Xplore下载

in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 331–340.
[54] J. Zhang, K. Huang, Y. Huang, B. Chen, R. Wang, C. Wang, and X. Peng, "Killing two birds with one stone: Malicious package detection in npm and pypi using a single model of malicious behavior sequence," *ACM Transactions on Software Engineering and Methodology*, 2024.
[55] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proceedings of the 30th IEEE/ACM international conference on program comprehension*, 2022.
[56] Y. Zhang, X. Zhou, H. Wen, W. Niu, J. Liu, H. Wang, and Q. Li, "Tactics, techniques, and procedures (ttps) in interpreted malware: A zero-shot generation with large language models," *arXiv preprint arXiv:2407.08532*, 2024.
[57] X. Zheng, C. Wei, S. Wang, Y. Zhao, P. Gao, Y. Zhang, K. Wang, and H. Wang, "Towards robust detection of open source software supply chain poisoning attacks in industry environments," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1990–2001.
[58] X. Zhou, F. Liang, Z. Xie, Y. Lan, W. Niu, J. Liu, H. Wang, and Q. Li, "A large-scale fine-grained analysis of packages in open-source software ecosystems," *arXiv preprint arXiv:2404.11467*, 2024.
[59] X. Zhou, Y. Zhang, W. Niu, J. Liu, H. Wang, and Q. Li, "Oss malicious package analysis in the wild," *arXiv preprint arXiv:2404.04991*, 2024.

wnloaded on November 02,2026 at 05:12:30 UTC from IEEE Xplore. Restrictions apply

载于第*44*届国际软件工程会议：软件工程实践论文集，2022年，第331–340页。[54] J. Zhang、K. Huang、Y. Huang、B. Chen、R. Wang、C. Wang与X. Peng，“一石二鸟：利用单一恶意行为序列模型检测npm和pypi中的恶意包”，*ACM*软件工程与方法学汇刊，2024年。[55] K. Zhang、W. Wang、H. Zhang、G. Li与Z. Jin，“利用异构图学习表示程序”，载于第*30*届*IEEE/ACM*程序理解国际会议论文集，2022年。[56] Y. Zhang、X. Zhou、H. Wen、W. Niu、J. Liu、H. Wang与Q. Li，“解释型恶意软件中的战术、技术与程序（TTP）：基于大语言模型的零样本生成”，*arXiv*预印本*arXiv:240708532*，2024。。[57] X. Zheng、C. Wei、S. Wang、Y. Zhao、P. Gao、Y. Zhang、K. Wang与H. Wang，“面向工业环境中开源软件供应链投毒攻击的鲁棒检测”，载于第*39*届*IEEE/ACM*自动化软件工程国际会议论文集，2024年，第1990–2001页。[58] X. Zhou、F. Liang、Z. Xie、Y. Lan、W. Niu、J. Liu、H. Wang与Q. Li，“开源软件生态系统中包的大规模细粒度分析”，*arXiv*预印本*arXiv:2404.11467*，2024年。[59] X. Zhou、Y. Zhang、W. Niu、J. Liu、H. Wang与Q. Li，“真实环境中的开源软件恶意包分析”，*arXiv*预印本*arXiv:2404.04991*，2024年。

431

载。适用限制。