PROFMAL: Detecting Malicious NPM Packages by the Synergy between Static and Dynamic Analysis

Yiheng Huang*, Wen Zheng*, Susheng Wu*, Bihuan Chen*†,
You Lu*, Zhuotong Zhou*, Yiheng Cao*, Xiaoyu Li*, Xin Peng*
*College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China
†Corresponding Author

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 ecosystem, 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-

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 *bitsoex_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 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**.

```
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*

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  const os = require("os");
2  function tryGet(toCall){
3    try {
4      return toCall();
5    } catch (e) {
6      return "err";
7    }
8  }
9  data = {
10   p: package,
11   h: tryGet(os.hostname),
12   d: tryGet(os.homedir),
13   c: __dirname,
14 };
15 sendToServer(data);
```

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

```
1  const os = require("os")
2  const { exec } = require("child_process");
3  const https = require("https");
4  function printPlatform() {
5    const platform = os.platform();
6    console.log(platform);
7  }
8
9  function printMountedDisks() {
10   const platform = os.platform();
11   const cmd = platform === "win32" ?
12         "wmic logicaldisk 11  get name" : "df -h";
13   exec(cmd, (err, stdout, stderr) => {
14     console.log(stdout.trim());
15   });
16 }
17
18 function fetchSearch() {
19   https.get("https://user_defined.com", (res) => {
20     console.log("Status Code:", res.statusCode);
21  })
22 }
23
24 printPlatform()
25 printMountedDisks()
26 fetchSearch()
```

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

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**.

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

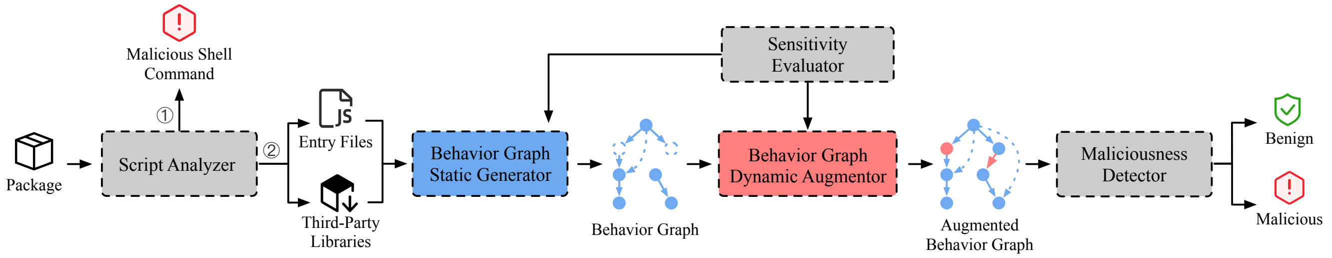

Fig. 4: Approach Overview of ProfMal

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 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

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:

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

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.

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 $propMap[\langle obj_{last}, p_{last} \rangle] = obj_r$.

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 $propMap[\langle obj_{a.b}, \text{c} \rangle] = obj_r$.

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

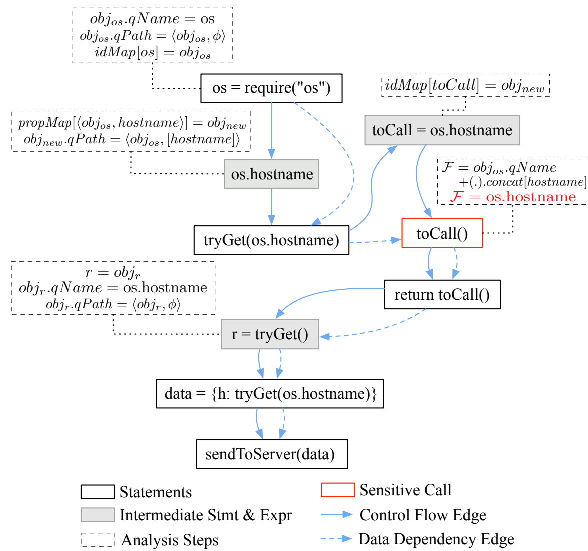

Fig. 6: Behavior Graph of *automation.samples_0.1.15*

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 $qPath = \langle obj_{os}, [hostname] \rangle$.

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.

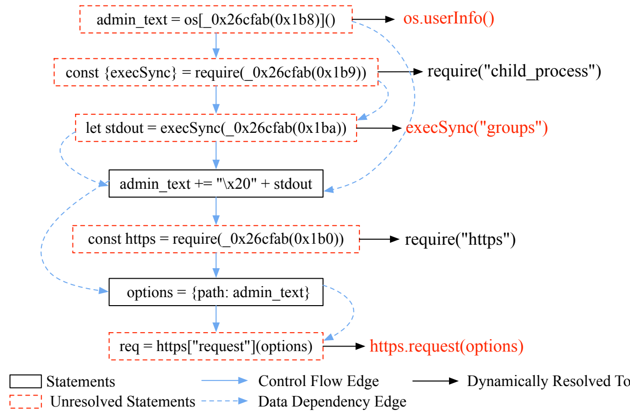

Fig. 7: Augmented BG of *bitsoex_react-design-system_14.1.4*

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 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. $\varphi(e)$

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:

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

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 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.

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

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%**

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.

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 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).

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 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

[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?” 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.