What You See Is Not What You Execute: Memory-Based Runtime SBOM Generation for Supply Chain Security

Hala Ali[a,*], Andrew Case[b] and Irfan Ahmed[a]

[a]Department of Computer Science, Virginia Commonwealth University, USA
[b]Volatility Foundation, USA

ARTICLE INFO

Keywords:
Supply Chain Security
SBOM
Memory Forensics
Volatility 3

ABSTRACT

Modern software development is heavily dependent on third-party components and public repositories, making software supply chain security a persistent challenge. To improve security and transparency, the use of Software Bill of Materials (SBOM) has emerged as a standardized technology. SBOM systems enumerate and record the components of software packages, and organizations use this information to discover outdated or vulnerable software within their environments. To be useful, SBOM documents must contain complete listings of software components. Unfortunately, SBOMs derived from existing tools are very limited as they only capture software packages that are referenced from static, install-time information. This approach fails to capture dynamically loaded modules, environment-specific dependencies, and version drift. To significantly increase the value of SBOM and to detect runtime attacks, we developed *MEM-SBOM*, a memory forensics framework that generates SBOMs directly from the runtime state of Python applications. Through this approach, *MEM-SBOM* precisely reveals the software modules and versions active on a system. Beyond enumeration, *MEM-SBOM* also analyzes function bytecodes to construct execution-aware dependency graphs and to perform fine-grained vulnerability reachability analysis. *MEM-SBOM* is implemented as Volatility 3 plugins and was tested on 51 Python applications and achieved 100% accuracy in recovering installed, imported, and dynamically loaded packages, and it constructed accurate dependency graphs, outperforming eight state-of-the-art SBOM tools. In a case study of the *tornado* package, *MEM-SBOM*'s reachability analysis reduced package-level vulnerability detections by 83%. These capabilities make *MEM-SBOM* a practical foundation for securing software supply chains, proactive defense, and incident response by providing a verifiable, runtime-grounded view of what is executed on a system.

1. Introduction

The widespread adoption of third-party libraries and frameworks from public repositories has become integral to modern software development, enabling rapid feature integration while simultaneously expanding the software supply chain attack surface (Nahum et al., 2024). Adversaries are increasingly exploiting these repositories to distribute malicious packages that compromise downstream applications (Vu et al., 2023). In 2024, more than 700,000 malicious packages were discovered in public registries, a 156% increase over the previous year (Sonatype, 2025). The official Python repository, *PyPI*, has repeatedly faced such incidents, with more than 12,000 malicious packages removed in 2022 alone (Mike Fiedler, 2023), yet malicious uploads persist despite repository-level scanning (Samaana et al., 2025; Kampourakis et al., 2025). Examples include `zlibxjson v8.2`, which embedded credential-stealing scripts that exfiltrated cookies and authentication tokens (Wang, 2024), and `fshec2`, which concealed credential theft and remote command handling within compiled bytecode (`.pyc`) to evade detection (Nelson, 2023). In April 2025, additional malicious packages (`bitcoinlibdbfix`, `bitcoinlib-dev`, and `disgrasya`) were downloaded more than 39,000 times before being removed, with the latter embedding an automated script for credit-card fraud (Lakshmanan, 2025). Even trusted frameworks can introduce critical vulnerabilities, such as the recent SQL-injection flaw in *Django* (CVE-2025-64459), with affected versions still publicly available on PyPI (Bidart, 2025) at the time of writing.

These recurring incidents have highlighted the urgent need for software supply chain transparency, prompting regulatory and industry efforts to improve visibility into software composition. In response, initiatives led by the Cybersecurity and Infrastructure Security Agency (CISA) and the National Telecommunications and Information Administration (NTIA) have advanced the Software Bill of Materials (SBOM) as a standardized mechanism for software transparency and risk management (Office of the Director of National Intelligence (ODNI) et al., 2023). An SBOM is a structured inventory of software components and dependencies, specifying their metadata (e.g., name, version, and license) and relationships (Tooling and Implementation Working Group, 2024). Based on CISA's guidance, SBOMs have become an increasingly expected requirement of federal software procurement (Cybersecurity and Infrastructure Security Agency (CISA), 2025). As a result, several tools have emerged to automate SBOM generation across ecosystems. However, these tools remain constrained by their reliance on static metadata, such as build manifests and package files (e.g., `requirements.txt`, `pyproject.toml`, and `setup.py`), which describe what was built rather than what

*Corresponding author
alih16@vcu.edu (H. Ali); andrew@dfir.org (A. Case); iahmed3@vcu.edu (I. Ahmed)
ORCID(s):

actually executes. Prior studies show that these metadata-driven tools generate incomplete and inaccurate SBOMs due to inconsistent format support, parser divergence, and weak handling of dynamic or environment-specific dependencies (Yu et al., 2024; Dalia et al., 2024; Cofano et al., 2024; Xia et al., 2023; Stalnaker et al., 2024). Furthermore, metadata files themselves can be compromised, as attackers may modify dependency specifications to mislead SBOM tools into producing incorrect component and version information Zahan et al. (2023).

Recent efforts have extended SBOM generation to the software deployment phase, including container-level analysis (Kawaguchi and Hart, 2024), package-manager integration (Benedetti et al., 2025), and integrity verification of deployed components (Sharma et al., 2024). Although these approaches improve accuracy compared to static build-phase methods, they still rely on filesystem artifacts and package managers, which reflect the software's state at install. This limitation becomes even more evident in dynamic, interpreted languages where dependencies are resolved during execution rather than at build time. Python presents a particularly challenging and representative example of such dynamic ecosystems due to its highly flexible import mechanism, which allows modules to be loaded at runtime through conditional, delayed, or reflective imports (Ali et al., 2025b). This flexibility causes different frameworks and applications to load and retain dependencies in distinct ways. For example, when upgrading shared dependencies such as *asgiref*, *Django*'s *StatReloader* dynamically loads the newer version into memory while running, whereas *Celery*, lacking an auto-reload mechanism, continues executing the older version already loaded at startup.

While runtime SBOMs aim to capture the active components of running applications by instrumenting the executing system (Mirakhorli et al., 2024), this approach assumes that the system remains operational and monitorable, an assumption that often fails in adversarial or failure scenarios. A malicious dependency executed at runtime can cause the system to crash entirely or make it inaccessible through corruption, destruction, or ransomware encryption Cybersecurity and Infrastructure Security Agency (CISA) (2021); National Counterintelligence and Security Center (NCSC) (2021); Wu and Xu (2016). In such cases, volatile memory becomes the only reliable source for recovering the actual runtime components and dependencies, and thus generating accurate runtime SBOMs for forensic analysis.

Therefore, in this paper, we propose *MEM-SBOM*, a memory-based framework that generates the SBOM directly from the runtime state of Python applications. *MEM-SBOM* systematically traverses the interpreter's internal data structures, including interpreter registries, thread contexts, the garbage collector, arena allocators, and heap regions, to extract all modules resident in memory. This multi-layered architecture ensures completeness and forensic soundness even under evasive conditions, such as module deletion, import-hook exploitation, sub-interpreter isolation, and garbage collector manipulation. *MEM-SBOM* then cross-validates the extracted modules across all layers and application processes, organizes them by their parent packages, and excludes the built-in and standard libraries. For resolving version information, *MEM-SBOM* inspects module attributes (e.g., `__version__`), correlates with installation metadata, and queries PyPI when necessary. The recovered third-party packages are serialized into a CycloneDX-compliant SBOM. Beyond generating the SBOM, *MEM-SBOM* analyzes the Python objects associated with each package and disassembles the bytecode of its functions to construct execution-aware dependency graphs that capture both direct and transitive runtime relationships. Leveraging the resulting SBOMs and graphs, along with integrating public vulnerability data, the framework performs fine-grained, function-level reachability analysis to distinguish only those active, exploitable code paths in memory.

We implemented *MEM-SBOM* as a suite of new plugins for Volatility 3 (Volatility Foundation, 2025). We then evaluated it across 51 real-world Python applications spanning 23 diverse categories, including CLI tools, web frameworks, machine learning platforms, workflow schedulers, and visualization tools. Validated against a dual static and runtime ground truth, *MEM-SBOM* accurately recovers all application packages, including those dynamically loaded at runtime, as well as the cached metadata embedded in installed packages, across heterogeneous execution environments. It further identifies ten cases where the version embedded in the loaded code differs from the version declared at installation time. In a *tornado* case study, function-level reachability analysis shows that only `Streamlit`, among six *tornado*-dependent applications flagged as vulnerable at the package level, actually invokes the affected routines at runtime, eliminating 83.3% of false-positive vulnerability reports. Finally, our comparison against eight state-of-the-art SBOM tools shows that none of them support all Python metadata configurations or capture dynamically loaded packages, resulting in incomplete SBOMs, partially constructed dependency graphs, and ultimately inaccurate vulnerability reports.

Our contributions are as follows:

- We propose *MEM-SBOM*, the first framework to generate SBOMs directly from the runtime state of Python applications, providing an execution-grounded view of software components.
- We characterize evasion techniques that conceal Python modules during execution and design a robust multi-layered extraction architecture capable of detecting and recovering all modules.
- We construct runtime dependency graphs and perform function-level vulnerability reachability analysis, identifying active exploitable code paths.
- We evaluate *MEM-SBOM* across 51 real-world Python applications, achieving higher accuracy and vulnerability precision than existing SBOM tools.

The remainder of this paper is organized as follows. Section 2 provides the background, and Section 3 reviews

related work. Section 4 outlines the key challenges, while Section 5 explains the methodology. Section 6 presents the evaluation, Section 7 discusses limitations and future directions, and Section 8 concludes the paper.

2. Background

This section presents the SBOM types and standards and illustrates the import mechanism and memory analysis of the Python runtime.

2.1. SBOM Fundamentals

SBOMs can be generated at different stages of the software lifecycle, each providing a distinct level of visibility into software composition (Cybersecurity and Infrastructure Security Agency (CISA), 2023). **Build SBOM**s describe the components, dependencies, and build metadata incorporated during compilation or packaging. **Deployed SBOMs** enumerate the components installed and configured within operational environments. **Runtime SBOMs**, in turn, capture the components actively loaded during execution, revealing dynamic and environment-specific dependencies that are invisible to static analysis. SBOMs are commonly represented using two standards. **SPDX**, maintained by the Linux Foundation, focuses on license compliance, copyright attribution, and file-level provenance (Software Package Data Exchange (SPDX), 2025). **CycloneDX**, developed by OWASP, is a lightweight, security-oriented standard that emphasizes dependency relationships, component provenance, and vulnerability correlation (CycloneDX Project, 2025b).

2.2. Import Mechanism of Python

The import mechanism of Python follows a multi-stage resolution process formalized in *PEP 451* and *PEP 420* (Snow, 2013; Smith, 2012). When the interpreter encounters an `import` statement, it first checks its per-interpreter module registry (`sys.modules`), which maps module names to loaded `PyModuleObject` instances. If the module is not found, the import machinery proceeds through finder objects listed in `sys.meta_path`. Each finder implements `find_spec()` to locate the requested module and, upon success, returns a `ModuleSpec` that encapsulates the module's name, origin, loader, and other import metadata. Using this specification, the interpreter calls `loader.create_module(spec)` if provided, or instantiates a new module object otherwise, registers it in `sys.modules`, and executes it via `loader.exec_module(module)` to populate its namespace. Built-in and frozen modules use non-filesystem origins (e.g., "built-in" or "frozen"), whereas file-based modules populate attributes, such as `__file__` and `__cached__`. This spec-driven model replaces the legacy `load_module()` API, cleanly separating module discovery (`find_spec()`) from execution (`exec_module()`) and exposing import state through the `__spec__` attribute for reliable runtime introspection.

2.3. Memory Analysis of Python

Python's runtime architecture consists of interdependent memory management subsystems that control object allocation and persistence in memory. The global `_PyRuntime` structure, of type `_PyRuntimeState`, represents the entry point to the runtime, maintaining interpreter registries, thread states, and the configuration of the garbage collector. This structure is located in the dynamic symbol table of *ELF* binaries on Linux or the export table of *PE* binaries on Windows (Ali et al., 2025a). Each interpreter instance (`PyInterpreterState`) maintains its own module registry, import machinery, and a set of thread contexts (`PyThreadState`), where each thread holds a stack of execution frames containing references to active function objects (Ali et al., 2025b). Memory allocation follows a hybrid scheme. Small objects ($\leq$512 bytes) are managed by the `pymalloc` allocator within 256 KB arenas subdivided into 4 KB pools (Golubin, 2019). Larger objects are handled by the system allocator via `brk()` for the process heap and `mmap()` for separate anonymous memory mappings, depending on allocation size and system configuration. The garbage collector tracks objects across three generations using doubly-linked lists via `gc_prev` and `gc_next` pointers. Together, these mechanisms define the spatial organization of Python objects in memory, enabling the reconstruction of complete runtime SBOMs from memory artifacts.

3. Related Work

Evaluating Existing SBOM Tools. Prior studies primarily evaluated SBOM tools, standardization challenges, and adoption barriers. Dalia *et al.* (Dalia et al., 2024) evaluated *Syft*, *Microsoft SBOM Tool*, *Tern*, and *CycloneDX* tools in terms of multi-platform support, format compatibility, and usability, demonstrating that these tools remain static, metadata-driven, and lacking operational readiness. Mirakhorli *et al.* (Mirakhorli et al., 2024) analyzed 84 tools and observed heavy reliance on package managers, inconsistent component identification, and limited cross-language coverage, urging the need for standardized dependency semantics and validation frameworks. Halbritter and Merli (Halbritter and Merli, 2024) further showed that even the most accurate tools (*CycloneDX-Npm* and *CycloneDX-Conan*) fail to meet the *NTIA*'s minimum-element requirements, frequently omitting dependencies or misreporting versions. From an adoption perspective, Xia *et al.* (Xia et al., 2023) found that standardization issues, inadequate validation, and information-sensitivity concerns hinder adoption. Stalnaker *et al.* (Stalnaker et al., 2024) identified specification complexity, tooling insufficiency, and verification issues as major barriers, and highlighted the need for language-specific and verifiable SBOM frameworks, particularly for emerging AI/ML domains lacking formal standards. Bi *et al.* (Bi et al., 2024) analyzed 4,786 SBOM-related GitHub discussions, linking poor SBOM quality to technical debt and governance deficiencies, highlighting the need for continuous assurance integration. Complementary studies by Cofano *et al.* (Cofano et al., 2024) and Yu *et al.* (Yu et al., 2024)

highlighted consistent reliability issues in *Trivy*, *Syft*, *CDXGen*, *ORT*, *Microsoft SBOM Tool*, and *GitHub Dependency Graph*, attributing inaccuracies to flawed metadata parsing, ambiguous naming and version-resolution, and incomplete dependency resolution, reflecting the inherent limitations of static, metadata-driven analysis.

Proposing New SBOM Techniques. Recent efforts have extended SBOM generation to the deployment phase of software. Sharma *et al.* (Sharma et al., 2024) introduce *SBOM.EXE*, which prevents dynamic code injection in Java by maintaining a Bill of Materials Index (BOMI) of legitimate class checksums. This approach is inherently limited since it requires complete test coverage and cannot handle non-deterministic code generation or hidden classes. Benedetti *et al.* (Benedetti et al., 2025) propose *PIP-SBOM*, which integrates SBOM creation into the Python package manager *pip* using its native resolver. However, this approach reports versions that would be freshly installed rather than actual deployed versions, and inherits pip's limitations in failing to capture runtime-only dependencies and dynamic imports. Kawaguchi and Hart (Kawaguchi and Hart, 2024) developed *C3DRS*, a consumer-side vulnerability management system that generates SBOMs through dynamic container inspection. Although it improves detection accuracy, its analysis remains limited to filesystem artifacts.

4. Challenges

This section identifies six challenges that *MEM-SBOM* is designed to overcome.

C1: Evasive manipulation of the Python import system. Python's dynamic runtime allows introspection and modification of its import system at execution time. The same flexibility that enables extensibility, via monkey patching, custom import hooks, or dynamic loaders, can be exploited to conceal or alter modules in memory. Attackers may delete or overwrite registry entries in *sys.modules*, replace legitimate modules with malicious counterparts, inject hidden modules through secondary sub-interpreters, or use low-level APIs (e.g., `ctypes`, `importlib`, or `runpy`) to manipulate the garbage collector and create unregistered modules. Such behaviors complicate the recovery of a complete and trustworthy module set from memory.

C2:Fragmented runtime state in multi-process applications. Complex Python applications (e.g., *Airflow*, *FastAPI*, *MLflow*) comprise multiple cooperating processes, each maintaining its own interpreter state and dependency set. These processes may load different or conflicting versions of shared packages depending on the deployment environment. To generate an accurate, application-level SBOM, these fragmented module views must be merged while resolving version conflicts and cross-process duplications.

C3: Distinguishing third-party packages from internal and standard libraries. When a Python application runs, the interpreter instantiates not only the application's modules but also a wide range of internal and standard-library modules required for its own operations. In memory, all of these appear as `PyModuleObject` instances within the same runtime, making it difficult to isolate third-party components of the application. Differentiating these modules is essential for constructing an accurate SBOM that reflects only supply chain dependencies.

C4: Incomplete version information in memory. Accurately resolving package versions from memory poses a major challenge for memory-based SBOM generation. A `PyModuleObject` may expose version-related attributes, such as `__version__`, `VERSION`, `__VERSION__`, `version_info`, `__version_info__`. However, these attributes lack standardization, exhibit inconsistent naming, and are not guaranteed to be present in memory. This challenge complicates the generation of accurate memory-based SBOMs, for which package name and version are essential elements.

C5: Structural over-approximation of dependency relationships. Python's eager import mechanism loads entire namespaces into memory, making it difficult to distinguish between modules that are merely loaded and those actually used. When a module imports another, the callee's modules, classes, and functions are instantiated in memory, even if most remain unreferenced. Recursive imports further amplify this effect by pulling additional transitive dependencies into memory. Traversing these imports results in deep transitive chains that populate memory with modules the application never invokes, leading to inflated SBOMs with high false-positive rates. This, in turn, causes the dependency graph to over-approximate the true runtime state, reporting relationships that are syntactically derived but semantically invalid. Therefore, with such structural over-approximation, constructing precise and execution-aware runtime SBOMs poses another challenge.

C6: Identifying semantically reachable and exploitable code paths. Beyond C5's over-approximation at the module level, this challenge extends to the internal objects of each module. Once imported, a module's functions, classes, and attributes coexist in memory as a static namespace, yet only a subset is ever executed. Treating all resident objects as active inflates the application's logical footprint and obscures its actual runtime behavior, complicating any assessment of vulnerability impact. Vulnerabilities residing in unreferenced portions of loaded modules may be structurally visible yet semantically unreachable in practice. Accurately identifying actively invoked objects and enumerating exploitable call paths across module objects is therefore a core technical challenge.

5. Methodology

Figure 1 illustrates the *MEM-SBOM* pipeline. After identifying the application processes and locating the interpreter internals within the memory dump (Ali et al., 2025b), the framework extracts the runtime module state to produce a CycloneDX-compliant SBOM, a runtime dependency graph, and function-level exploitation paths.

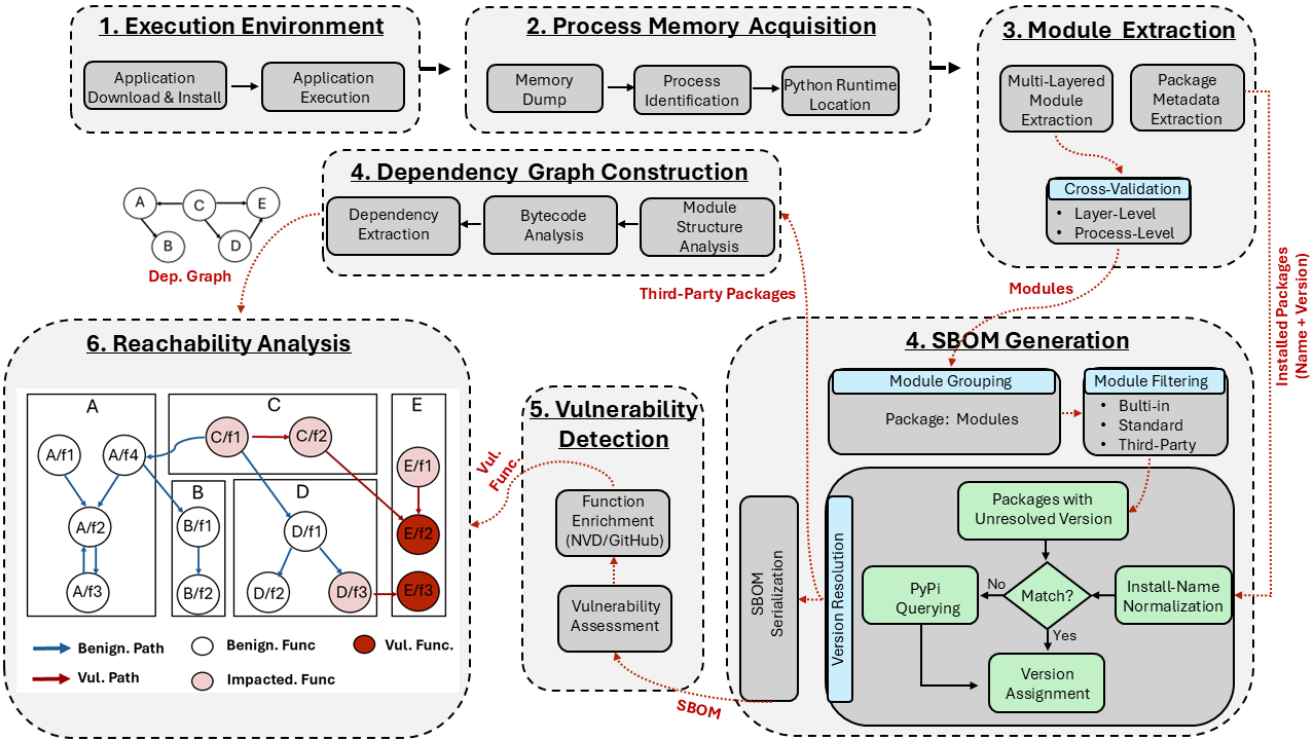

Figure 1: Overview of the *MEM-SBOM* Framework

5.1. Module Extraction

5.1.1. Package Metadata Extraction

To identify installed packages without filesystem access, MEM-SBOM inspects CPython's import system cache, `sys.path_importer_cache`, which maps filesystem paths to finder objects responsible for module discovery (see Section 2.2). Each FileFinder instance maintains a persistent `_path_cache` containing all recognized directory entries, including source (`.py`), bytecode (`.pyc`), extension (`.so`, `.pyd`), and package-metadata directories (`.dist-info`, `.egg-info`). This cache is constructed once at import time, providing a stable record of the interpreter's package environment. *MEM-SBOM* extracts package metadata names and versions from cached directory entries that match the *PEP 427* wheel format (`package_name-version.dist-info`) or legacy `egg-info` naming scheme. The framework normalizes the recovered names, associates version numbers, and identifies their installation scope based on directory paths.

5.1.2. Multi-Layered Module Extraction

Python's runtime structures can be exploited to conceal malicious module objects through various evasion techniques, as discussed next.

A1. Module Deletion. Deletes entries from `sys.modules`, hiding modules that remain live through residual references elsewhere in memory. **A2. Module Overwrite.** Overwrites or aliases legitimate entries in `sys.modules` (e.g., mapping a benign name to a malicious module object) to hide the module behind a trusted identifier. **A3. Import-Hook Interception.** Inserts a malicious finder into `sys.meta_path` that redirects a targeted module import to a custom loader, which executes a malicious module during initialization. **A4. Import-Free Module Creation.** Creates modules via low-level `importlib` and `runpy` APIs without registering them in `sys.modules`. **A5. Sub-Interpreter Isolation.** Injects malicious modules into secondary interpreters (sub-interpreters) within the same process, making them invisible to scans of the main interpreter's registry. **A6. Garbage-Collector Manipulation.** Manipulates the pointers of GC-tracked objects (e.g., via `ctypes`) to unlink malicious module objects from the GC's linked lists, evading GC-based detection.

To address these evasion techniques, *MEM-SBOM* employs a robust multi-layered extraction approach that overcomes Challenge **C1** by incrementally expanding coverage from interpreter registries to raw heap regions. Layers 1–3 provide authoritative and context-rich evidence with minimal false positives, ideal for benign module recovery. Layers 4–5 extend extraction to raw memory arenas and heap regions. Although these layers require object-type validation, they are essential for achieving completeness, as all modules with active references remain allocated in memory. **L1. Interpreter Registry.** Traverses `sys.modules` of each interpreter, providing the fastest extraction mechanism. **L2. Thread Execution States.** Inspects the global namespace and local variables of the stack frames to identify their modules that might be hidden from the interpreter registry. **L3. Garbage Collector Traversal.** Traverses the doubly-linked lists of objects managed by the process-level garbage collector, which is shared across all interpreters. **L4. Arena-Aware Analysis.** Scans arena pools to identify

valid module objects within allocated blocks by verifying their reference count (`ob_refcnt`) and type (`ob_type`). **L5. Heap Areas Scanning.** Recovers large module objects in 8-byte aligned increments, interpreting 16-byte sequences as `ob_refcnt` and `ob_type`, validating the `PyObject` header, dereferencing the type pointer if it corresponds to a module type, and confirming module identity.

Cross-validation and de-duplication. To address Challenge **C2**, *MEM-SBOM* implements two levels of cross-validation. First, within each application process, it deduplicates modules discovered across all five extraction layers by name. Second, for multi-process applications, per-process module sets are merged to generate the application's complete module set.

5.2. SBOM Generation

After module extraction, *MEM-SBOM* aggregates the recovered modules into packages, excludes built-in and standard-library components, resolves third-party package versions, and serializes the resulting packages into a CycloneDX-compliant SBOM for further analysis.

5.2.1. Module Grouping

A Python application consists of a top-level package and its dependent packages, each composed of one or more `.py` files represented in memory as module objects (Snow, 2013; Smith, 2012). The application entry point appears as the `__main__` module, distinguishing it from all other modules. Each module object is identified by its fully qualified name (FQN), which encodes its position within the package hierarchy. For example, the package `requests` with its modules (`adapters`, `models`, and `sessions`) are represented in memory as `requests`, `requests.adapters`, `requests.models`, and `requests.sessions`. *MEM-SBOM* groups all related modules under their top-level package, which is defined as the substring preceding the first dot (`.`) in the module name. This produces a one-to-many mapping between each package and the modules it contains, enabling package-level representation in the final SBOM. Notably, built-in modules appear as single identifiers, whereas standard-library and third-party packages use hierarchical names separated by dots.

5.2.2. Module Filtering

Given Challenge **C3**, the resulting packages are classified as built-in, standard, or third-party to distinguish trusted interpreter libraries from the application components, allowing SBOMs to focus primarily on third-party packages. Third-party packages are identified by their installation paths, such as `site-packages` or `dist-packages`, and corresponding vendor-specific directories, such as `vendor-packages` or `local-packages` when present in the runtime environment. Built-in libraries (e.g., `_signal`, `marshal`, and `_imp`) are identified through `sys.builtin_module_names`. Standard libraries are recognized by name and installation path patterns derived from the interpreter's installation directories (e.g., `/usr/lib/python3.x/`, `/lib/python3.x/`). Moreover, compiled extensions (`.so`, `.pyd`) located within these directories are also filtered out. Accordingly, *MEM-SBOM* relies on installation paths to ensure precise classification of package categories.

Some packaging tools bundle their own dependencies to ensure version consistency, creating namespace-isolated copies of third-party libraries. For instance, *pip* embeds its dependencies under `pip._vendor` (e.g., `pip._vendor.requests`), while *setuptools* uses `setuptools._vendor`. *MEM-SBOM* identifies and excludes these bundled copies by matching common namespace conventions (e.g., `._vendor.`, `.extern.`) and directory patterns (e.g., `/_vendor/`, `/extern/`). Internal utility packages, such as `_distutils_hack`, are similarly omitted to ensure the SBOM reflects only the application dependencies.

5.2.3. Version Resolusion

To identify the version of each recovered third-party package, *MEM-SBOM* analyzes their `PyModuleObject` structures to inspect their version attributes (e.g., `__version__`, `VERSION`, `version_info`). As noted in Challenge **C4**, these attributes are not mandatory and therefore might not appear in memory. When version fields cannot be located, *MEM-SBOM* attempts to map each import name to its installation record in `sys.path_importer_cache` to retrieve the associated version (§5.1.1). However, distribution names in installation records differ from their import counterparts due to the normalization rules defined in *PEP 503* (Stufft, 2015). For example, the distribution `email-validator` corresponds to the import name `email_validator`. *MEM-SBOM* resolves such discrepancies by applying *PEP 503* normalization (i.e., case folding and hyphen-to-underscore transformation) before matching import and installation names.

A more complex case arises when normalization fails to establish a correspondence between the import names observed in memory and the installation names in the local cache (`sys.path_importer_cache`). For instance, `sys.modules` may contain `cv2`, while the cache lists `opencv-contrib-python`, and `opencv-python-headless`. Although these distributions originate from the same upstream *OpenCV* project, they install distinct packages that all expose the same import name (`cv2`), making it impossible to infer from the local cache alone which distribution provided the loaded module.

To resolve such ambiguity, and given that the PyPI API operates on distribution identifiers rather than import names, *MEM-SBOM* queries PyPI using the previously unmatched installation names discovered in the local cache. The framework retrieves package metadata through the public JSON API, extracting import names from the `.dist-info/top_level.txt` file for wheel distributions and from the `.egg-info/top_level.txt` file for source distributions. This file enumerates the top-level importable names defined by each distribution. This process allows *MEM-SBOM* to recover the complete set of import names corresponding to the locally installed distributions. The framework then matches these recovered import names against the modules observed in memory to determine their originating distributions. Once a match is found, the version information is obtained from the local cache, ensuring

that the generated SBOM reflects the actual environment state. A final case involves packages dynamically loaded at runtime or bundled within the application directory, which do not appear in the installation cache and may lack version attributes in their module objects. Ultimately, when the version attribute remains indeterminate, *MEM-SBOM* records its value as `"unknown"`, consistent with CISA's SBOM minimum-element specification (Cybersecurity and Infrastructure Security Agency (CISA), 2025).

5.2.4. SBOM Serialization

After resolving package versions, *MEM-SBOM* serializes the recovered package information into a CycloneDX-compliant SBOM in JSON format. We selected this standard due to its security-oriented design and support for vulnerability correlation, which aligns with the cybersecurity and digital forensics support objectives of our work. Following CISA's guidance (Tooling and Implementation Working Group, 2024), we focus on attributes essential to incident response (e.g., component identity, version, and dependency relationships) rather than licensing or copyright metadata. The generated SBOM consists of four primary sections. **Metadata.** Defines SBOM provenance using a UUID-based serial number, extraction timestamp, and tool identifier to ensure extraction traceability. **Components.** Lists each recovered package with fields `name`, `version`, and package URL (PURL) following (`pkg:pypi/package@version`) format, enabling direct mapping to vulnerability databases such as National Vulnerability Database (NVD), Open Source Vulnerabilities (OSV), and GitHub Security Advisories (GHSA). Component hashes are omitted because memory-resident modules lack stable file artifacts, consistent with CISA's allowance for inaccessible sources. Additional forensic metadata, such as `extracted_from: memory` to distinguish runtime from manifest-based SBOMs, and `file_path` for filesystem correlation, are preserved in the `properties` of the component. **Dependencies.** Lists the direct dependencies of the application packages. **Annotations.** Includes the annotator identity, creation timestamp, extraction source, and package count, allowing analysts to assess completeness and maintain provenance across multiple SBOMs.

5.3. Dependency Graph Construction

This section describes the dependency graph construction, which represents the relationships between the recovered third-party packages while considering Challenge **C5**.

5.3.1. Module Structure Analysis

The namespace dictionary (`__dict__`) of each `PyModuleObject` contains references to the module's own objects as well as those imported from other modules. For module objects, the defining module is identified through the `md_name` attribute; for class objects, through the `__module__` attribute; and for callable objects (e.g., functions, generators, coroutines, asynchronous generators, instance methods, class methods, and static methods), through the `func_module` attribute. The analysis is applied recursively to capture dependencies within nested classes and inner functions.

5.3.2. Bytecode Analysis

To extract import statements embedded within the function body, we disassemble its compiled bytecode and apply a stride-1 sliding window of width four over the opcode sequence to capture the complete instruction patterns associated with import variants. During traversal, non-semantic scaffolding instructions (e.g., `RESUME`, prologue assignments, and constant initializations) are excluded to reduce noise. Each window is then compared against the opcode patterns summarized in Table 1.

Standard Import Patterns (P1-P3). Basic import statements (`import module`) generate an `IMPORT_NAME` instruction, followed by a scope-dependent storage instruction (`STORE_FAST` for function scope or `STORE_NAME` for module/-class scope). Aliased and multi-module imports yield equivalent opcode sequences repeated per module, with the module name extracted from `IMPORT_NAME.argval` field.

From-Import Patterns (P4-P8). The `from module import item` statement generates an `IMPORT_NAME` followed by one or more `IMPORT_FROM` instructions and concluding with `POP_TOP`. The analysis extracts only the base module name, as imported symbols do not influence dependency relationships. The star-import variant (`from module import *`) replaces `IMPORT_FROM` with `IMPORT_STAR` and omits `POP_TOP`.

Dynamic Import Patterns (P9-P11). Function-based imports, called via `exec()`, `importlib.import_module()`, or `__import__()`, are detected by recognizing their characteristic opcode call sequences rather than dedicated import opcodes. For `exec()`, the analysis inspects the `LOAD_CONST` string operand and applies regular expressions to extract embedded import statements. In contrast, `importlib` and `__import__()` embed the target module name as a string constant within the `LOAD_CONST` instruction.

Note. The bytecode analysis maintains cross-version compatibility by handling instruction-set variations across Python releases. Bound-method calls compiled as `LOAD_METHOD` → `CALL_METHOD` in Python 3.6–3.10 are replaced in Python 3.11 by `LOAD_ATTR`→ `CALL`, optionally followed by `KW_NAMES` to handle keyword arguments. For name resolution, module-level code uses `LOAD_NAME`→ `STORE_NAME`, while function scopes use `LOAD_GLOBAL`→ `STORE_GLOBAL` for global variables, `LOAD_FAST` → `STORE_FAST` for local variables, and `LOAD_DEREF` → `STORE_DEREF` for closure variables. When `LOAD_CONST` instructions have a code object as their argument value, the analysis recursively inspects them to recover imports from all inner scopes.

5.3.3. Dependency Extraction

Algorithm 1 constructs the dependency graph by integrating namespace analysis and bytecode inspection. For each third-party package $p \in \mathcal{P}_{3rd}$, let $\mathcal{M}_p$ denote the set of modules belonging to p. For each module $m \in \mathcal{M}_p$, the algorithm initializes an empty import set $\mathcal{I}_m$ and traverses the module's dictionary. For each object o in the dictionary, it determines the object type τ through the `ob_type` field. When τ is `module` and the referenced module name belongs to a different package than p, it is identified as an external

Table 1
Bytecode patterns corresponding to Python import constructs.

Pattern	Python Code Example	Bytecode Sequence Pattern
P1: Standard Import	`import math`	`IMPORT_NAME(math)` → `STORE_FAST(math)` / `STORE_NAME(math)`
P2: Import with Alias	`import math as m`	`IMPORT_NAME(math)` → `STORE_FAST(m)`/ `STORE_NAME(m)`
P3: Multiple Imports	`import os, sys`	Repeat: (`IMPORT_NAME` → `STORE_FAST` / `STORE_NAME`) for each module
P4: From Import (Single)	`from math import sqrt`	`IMPORT_NAME(math)` → `IMPORT_FROM(sqrt)` → `STORE_FAST(sqrt)`/ `STORE_NAME(sqrt)` → `POP_TOP`
P5: From Import (Multiple)	`from math import sqrt, pi, cos`	`IMPORT_NAME(math)` → Repeat: (`IMPORT_FROM` → `STORE_FAST` / `STORE_NAME`) for each item → `POP_TOP`
P6: From Import with Alias	`from math import sqrt as sq`	`IMPORT_NAME(math)` → `IMPORT_FROM(sqrt)` → `STORE_FAST(sq)` / `STORE_NAME(sq)` → `POP_TOP`
P7: From Import Star	`from math import *`	`IMPORT_NAME(math)` → `IMPORT_STAR`
P8: Nested module Import	`from os.path import join`	`IMPORT_NAME(os.path)` → `IMPORT_FROM(join)` → `STORE_FAST(join)` / `STORE_NAME(join)` → `POP_TOP`
P9: Dynamic Import via `exec()`	`exec('import math')`	`LOAD_GLOBAL/LOAD_NAME(exec)` → `LOAD_CONST(string)` → `CALL_FUNCTION/CALL`

P10: Dynamic Import via `importlib`	`import importlib`
	`m = importlib.`
	`import_module('math')`
		`IMPORT_NAME(importlib)` → ... → `STORE_FAST(importlib)` / `STORE_NAME(importlib)` → `LOAD_METHOD/LOAD_ATTR(import_module)` → `LOAD_CONST('math')` → `CALL_METHOD/CALL`

P11: Dynamic Import via `__import__()`	`m = __import__('math')`	`LOAD_GLOBAL/LOAD_NAME(__import__)` → `LOAD_CONST('math')` → `CALL_FUNCTION/CALL`

dependency and appended to $\mathcal{I}_m$. For class objects, it extracts the `__module__` attribute and recursively processes the class dictionary to capture nested dependencies. For callable objects, it extracts the `func_module` reference, validates it as an external dependency, analyzes its bytecode, and merges the results into $\mathcal{I}_m$. After computing $\mathcal{I}_m$ for all modules of package p, the algorithm aggregates and deduplicates imports across $\mathcal{M}_p$, and creates a directed edge $E(p, q)$ for each distinct imported package q, yielding a package-level dependency graph.

5.4. Vulnerability Detection

To address challenge **C6**, *MEM-SBOM* integrates the SBOM, dependency graph, and public vulnerability data to identify vulnerable packages within the application, determine their impact scope, and trace all reachable paths to them. To identify vulnerabilities, we use *Grype* (v0.100.0) as the vulnerability scanner due to its accuracy, ecosystem coverage, and competitive performance compared to alternative scanners (Anchore, 2025a; Benedetti et al., 2025; Bufalino et al., 2025). This tool maps SBOM components (via their PURLs) to known CVEs using aggregated advisory feeds (e.g., NVD, OSV, and GitHub Security Advisories). For each SBOM component, it produces a structured set of matching vulnerabilities, including identifiers, affected version ranges, and severities, which *MEM-SBOM* attaches to the corresponding packages as input to the function-level analysis.

5.4.1. Function-Level Enrichment

Given the over-approximation inherent in package-level vulnerability detection, as discussed in Challenges **C5** and **C6**, *MEM-SBOM* augments CVE matches with function-level metadata extracted from public vulnerability advisories. This enrichment enables us to distinguish vulnerable code paths within the application. For each CVE reported by *Grype*, we query NVD and GitHub Security Advisories to obtain additional context. From NVD, we retain entries whose CPE configurations match the affected package and apply heuristics over the free-text CVE description to extract function identifiers. From GitHub advisories, when remediation commits or pull requests are available, we analyze the corresponding diffs to identify updated function definitions. When function-level attribution cannot be reliably determined, either due to incomplete advisory descriptions or genuinely package-wide impact, we label the CVE with package-wide scope and leave finer-grained localization to manual analysis.

5.5. Reachability Analysis

MEM-SBOM determines which parts of the application are exposed to identified CVEs by tracing exploitation paths through backward taint propagation. Given a vulnerability sink $S = (p, m, f)$, where p denotes the vulnerable package, m the module, and f the function, it traces backward from f to identify all callers that eventually reach the application entry point. This produces taint propagation paths $\subsetneqq =$

Algorithm 1 *Dependency Graph Construction*

Input: $\mathcal{P}_{3rd}$ (third-party packages)
Output: $\mathcal{D}_{graph}$ (dependency graph)
Initialize $\mathcal{D}_{graph} \leftarrow \emptyset$
for each package $p \in \mathcal{P}_{3rd}$ **do**
  Initialize $\mathcal{I}_p \leftarrow \emptyset$ {Imports observed for package p}
  for each module $m \in \mathcal{M}_p$ **do**
    Initialize $\mathcal{I}_m \leftarrow \emptyset$
    ProcessDict$(p, m.__dict__, \mathcal{I}_m)$
    $\mathcal{I}_p \leftarrow \mathcal{I}_p \cup \mathcal{I}_m$
  end for
  for each imported package $q \in \mathcal{I}_p$ **do**
    $\mathcal{D}_{graph} \leftarrow \mathcal{D}_{graph} \cup \{E(p, q)\}$
  end for
end for

Procedure ProcessDict$(p, dict, \mathcal{I}_m)$
for each object $o \in dict$ **do**
  $\tau \leftarrow o$.ob_type.tp_name
  if $\tau =$ module **then**
    $mod\_name \leftarrow o.\_\_name\_\_$
    if $pkg(mod\_name) \neq p$ **then**
      $\mathcal{I}_m \leftarrow \mathcal{I}_m \cup \{mod\_name\}$
    end if
  else if $\tau =$ class **then**
    $class\_mod \leftarrow o.\_\_module\_\_$
    if $pkg(class\_mod) \neq p$ **then**
      $\mathcal{I}_m \leftarrow \mathcal{I}_m \cup \{class\_mod\}$
    end if
    ProcessDict$(p, o.__dict__, \mathcal{I}_m)$ {Recurse into class}
  else if $\tau \in Callable\_Types$ **then**
    $func\_mod \leftarrow o$.func_module
    if $pkg(func\_mod) \neq p$ **then**
      $\mathcal{I}_m \leftarrow \mathcal{I}_m \cup \{func\_mod\}$
    end if
    $\mathcal{I}_{bytecode} \leftarrow$ BytecodeAnalysis$(o$.func_code$)$
    $\mathcal{I}_m \leftarrow \mathcal{I}_m \cup \mathcal{I}_{bytecode}$
  end if
end for

$\langle f_1, f_2, \ldots, f_n \rangle$, where $f_n = f$ is the vulnerable function and f_1 belongs to the main application package. The sink is considered reachable if at least one such path exists.

The analysis proceeds in two stages. First, within the vulnerable package, we inspect all modules, classes, and functions to identify local callers of f. Any function that invokes f, either directly or through an internal call chain within the same package, is added to the sink set. Second, for each package that imports the vulnerable package, we detect functions that directly call any element of the current sink set and mark them as vulnerable. We then apply the same intra-package analysis to these caller packages to identify internal call chains that reach those functions. The resulting set of direct and internal callers forms the sink set for the next iteration. This recursive process continues until reaching the application entry point. A vulnerability is considered reachable when at least one such backward call path exists. This process requires analyzing function bytecode to extract call relationships. Therefore, we extend the bytecode analysis introduced in Section 5.3.2 to detect the call patterns shown in Table 2, including inner-function calls, closure and nonlocal calls, decorator-wrapper patterns, and instance, class, and module-level function calls.

6. Evaluation

This section first validates the capabilities of *MEM-SBOM* in generating accurate runtime SBOMs across heterogeneous installation environments, constructing dependency relationships, and enabling precise vulnerability impact assessment. It also demonstrates its superiority over existing state-of-the-art tools through a set of experiments in terms of runtime package recovery, dependency-graph completeness, and vulnerability correlation.

6.1. Experimental Setup

We implemented *MEM-SBOM* as a suite of Volatility 3 (v2.26.2) plugins running on Ubuntu 20.04.6 LTS with 8 GB of RAM and Python 3.8.10. Application processes were identified using Volatility's `pslist` and `pstree` plugins. The reachability analysis plugin takes as input the application's process ID, main package (source), vulnerable package (target), vulnerable function, and the dependency graph, returning all runtime paths leading to the vulnerable code as output. The evaluated applications rely on Python versions ranging from 3.6 to 3.12. While the memory architecture remains stable across these versions, performance-oriented refinements introduce variations in internal offsets (e.g., the addresses of `_PyRuntime` and `arenas`). *MEM-SBOM* considers these version-specific offsets to maintain compatibility across all supported Python versions.

6.1.1. Application Deployment

We installed each of the applications using `pip install` into an isolated Python virtual environment created on Ubuntu 20.04 LTS virtual machines with 4 vCPUs and 4 GB RAM. This isolation ensures that each application is evaluated with its own dependency set, preventing any cross-application interference and guaranteeing that all modules recovered by *MEM-SBOM* belong solely to the target application. Memory dumps were captured by snapshotting the virtual machine immediately after the application completed its initialization phase, allowing sufficient time for it to complete startup and reach a steady execution state in which all the third-party packages used by the application were resident in memory. Modules related to environment management and packaging utilities (e.g., `pip`, `pkg_resources`, `setuptools`, `wheel`, `distlib`, and `virtualenv`) as well as application-specific configuration files were excluded from comparisons to ensure fair comparison between the *MEM-SBOM* and SBOM tools.

Table 2
Bytecode patterns corresponding to Python function call constructs.

Pattern	Python Code Example	Bytecode Sequence Pattern

F1: Inner Function Call	`def outer():`
	`def inner(): ...`
	`inner()`
		Definition: `LOAD_CONST(inner code object)` → `MAKE_FUNCTION` → `STORE_FAST(inner)`
		Call: `LOAD_FAST(inner)` → `CALL_FUNCTION` / `CALL` / `CALL_FUNCTION_KW` / `CALL_FUNCTION_EX`

F2: Closure Function Call	`def outer():`
	`x = 10`
	`def inner():`
	`return x`
	`inner()`
		Definition: `LOAD_CLOSURE(x)` → `LOAD_CONST(inner code object)` → `MAKE_FUNCTION(closure)` → `STORE_FAST(inner)`
		Call: `LOAD_FAST(inner)` → `CALL_FUNCTION` / `CALL` / `CALL_FUNCTION_KW` / `CALL_FUNCTION_EX`

F3: Closure/Nonlocal Call	`def outer():`
	`def helper(): ...`
	`def inner():`
	`helper()`
		`LOAD_DEREF(helper)` → `CALL_FUNCTION` / `CALL` / `CALL_FUNCTION_KW` / `CALL_FUNCTION_EX`

F4: Decorator Wrapper Pattern	`def wrapper(*args, **kwargs):` `return func(*args, **kwargs)`	**Definition:** `LOAD_CONST(inner code object)` → `MAKE_FUNCTION`(closure) → `STORE_FAST(wrapper)` **Inside wrapper - calling original:** `LOAD_DEREF(func)` → `LOAD_FAST(args)` → `LOAD_FAST(kwargs)` → `CALL_FUNCTION_EX`
F5: Function Call (instance/class/module)	`obj = MyClass()`, `obj.method()` `module.func()`	`LOAD_METHOD()` / `LOAD_ATTR()` → ... → `CALL_METHOD`/ `CALL` / `CALL_FUNCTION_KW` / `CALL_FUNCTION_EX`

6.1.2. Dataset

We evaluated 51 open-source Python applications from the *Awesome-Python* repository (Chen, 2025). Each selected application constitutes a standalone system with its own dependency tree, metadata configuration, and runtime context, thereby capturing the diversity of Python packaging and execution models observed in practice. The dataset covers a wide range of 23 categories, including CLI tools (e.g., *iRedis*, *LiteCLI*), web frameworks (e.g., *Django*, *Flask*, *FastAPI*), machine learning platforms (e.g., *MLflow*, *BentoML*, *Rasa*), task schedulers (e.g., *Apache Airflow*, *Celery*, *Prefect*), and data visualization tools (e.g., *Apache Superset*, *Orange*). This diversity reflects the heterogeneity of real-world Python deployments and enables evaluating *MEM-SBOM* under realistic operational conditions. The complete dataset is presented in Table 12 of Appendix A.

Note. The versions installed for each application and its dependencies are determined by the underlying operating system and environment configuration. Consequently, the versions resolved by `pip` may not correspond to the most recent releases available on PyPI. Our dataset, therefore, reflects realistic deployment environments rather than enforcing the latest releases.

6.1.3. Ground-Truth Generation

To evaluate the correctness of *MEM-SBOM* in recovering both installed package metadata and packages actively used at runtime, we constructed two complementary forms of ground truth: a static ground truth and a runtime ground truth. The static ground truth captures the set of packages installed in each application environment. We obtained this information using `pip freeze`, which enumerates all installed distributions and their resolved versions. This dataset serves as the reference for validating *MEM-SBOM*'s ability to reconstruct package names and versions directly from memory without filesystem access.

The runtime ground truth reflects the set of packages loaded by the Python interpreter during application execution. It is derived from the interpreter's module registry (`sys.modules`), which provides the authoritative record of legitimately imported modules in benign, non-adversarial deployments (see Section 5.1.2). Each application is instrumented using Python's `sitecustomize` module, a built-in startup mechanism that the interpreter automatically imports before any application-level code (Python Software Foundation, 2024), ensuring that our monitoring logic executes at interpreter initialization. The instrumentation captures the complete contents of `sys.modules` after the application reaches a steady execution state. It also reinitializes automatically in child processes spawned at runtime, providing complete coverage for multi-process applications. When triggered, each process serializes its module registry into a process-specific JSON file. Memory snapshots were acquired immediately thereafter, ensuring temporal alignment between the live module state and its in-memory representation. The same processing steps described in Section 5.2 (i.e., module grouping and filtering) are applied to the runtime ground-truth data to isolate third-party package modules. This dual-ground-truth strategy provides complementary visibility into installed distributions and actively used packages, enabling rigorous assessment of *MEM-SBOM*'s module-extraction accuracy. After validating *MEM-SBOM* against both views, we used it as the authoritative runtime baseline to evaluate the SBOM tools, while the static ground

truth continues to serve as the reference for installed-package accuracy comparisons.

6.1.4. SBOM Tools

We evaluated widely used SBOM tools: *Syft* (v1.27.1) (Anchore, 2025b), *Trivy* (v0.65.0) (Aqua Security, 2025), *CDXGen* (v11.4.4) (CycloneDX Project, 2025a), *CycloneDX-Python* (v7.1.0) (CycloneDX Project, 2025c), *SBOM4Python* (v0.12.4) (Harrison, 2025), *ORT* (v68.2.0) (OSS Review Toolkit, 2025), *Jake* (v3.0.14) (Sonatype Nexus Community, 2025), and *PIP-SBOM* (Benedetti et al., 2025). For metadata-based evaluation, all eight tools were assessed. For deployment-based evaluation, we restricted the comparison to *Syft*, *CDXGen*, *CycloneDX-Python*, and *PIP-SBOM*, as these tools can analyze an existing installed environment or construct a synthetic one for SBOM generation. *Syft* and *CycloneDX-Python* directly inspect the evaluated environment by enumerating installed distributions from the `site-packages` directory. *CDXGen*, in contrast, simulates an installation environment by creating a temporary virtual environment, resolving and installing all dependencies declared in the application's manifests, and enumerating the resulting `site-packages` directory. *PIP-SBOM* does not inspect installed environments; instead, it downloads the application's distribution into a temporary directory, extracts its metadata, and resolves dependencies using PIP's native logic. As a result, it reports the versions PIP would install, not the exact versions present in the evaluated environment.

6.2. Evaluation Research Questions

Our evaluation addresses the following research questions:

RQ1: How accurately can *MEM-SBOM* recover the runtime state of Python applications, including installed, imported, and dynamically loaded packages?

RQ2: Can MEM-SBOM determine the actual reachability of known vulnerabilities within running applications?

RQ3: How do existing SBOM generation tools perform across diverse metadata configurations and real deployment environments, and how do their SBOMs impact the vulnerability detection accuracy compared to *MEM-SBOM*?

6.2.1. Evaluation Metrics

We evaluate *MEM-SBOM* and existing SBOM tools using four complementary categories of metrics that capture the package identification, dependency graph construction, vulnerability detection accuracy, and vulnerability impact reachability. All metrics described in this section are computed per application, and their mean values are calculated across the full evaluation dataset to summarize the aggregate performance of the tools and *MEM-SBOM*.

Package Identification. Let P_S, P_T, and P_M denote the sets of packages obtained from the static ground truth (`pip freeze`), the evaluated SBOM tool, and *MEM-SBOM*, respectively. We compute:

$$C_P = \frac{|P_T \cap P_M|}{|P_M|} \times 100\%, \quad M_P = \frac{|(P_S \cap P_M) \setminus P_T|}{|P_M|} \times 100\%,$$

$$R_P = \frac{|P_M \setminus (P_S \cup P_T)|}{|P_M|} \times 100\%$$

C_P (*Coverage*) measures the proportion of runtime packages correctly detected by the tool, M_P (*Missing*) quantifies packages that exist in both static and runtime environments but are missing from the tool output, and R_P (*Runtime-Only*) captures runtime-only dependencies invisible to static analysis.

Version accuracy. For every package present in all three sets (static ground truth P_S, tool output P_T, and runtime set recovered by *MEM-SBOM* P_M), we evaluate whether the version reported by the tool ($v_T(p)$) or *MEM-SBOM* ($v_M(p)$) matches the static ground-truth version ($v_S(p)$). Version accuracy is the proportion of such matched versions and is defined as follows:

$$V_{acc} = \frac{|\{\, p \in (P_S \cap P_T \cap P_M) \mid v_X(p) = v_S(p) \,\}|}{|P_S \cap P_T \cap P_M|} \times 100\% \quad X \in \{T, M\}$$

Dependency Graph Construction. For each package p, let $D_T(p)$ and $D_M(p)$ denote its direct dependencies in the graphs generated by the evaluated SBOM tool and *MEM-SBOM*, respectively, and let E_T and E_M be the corresponding sets of directed edges. We define the common-edge domain (E_{comm}) as:

$$E_{\text{comm}} = \{\, (u \to v) \in E_M \;:\; u, v \in V_T \cap V_M \,\}$$

where V_T and V_M are the node sets of the tool and *MEM-SBOM* graphs, respectively. Using these definitions, we evaluate structural accuracy through three complementary metrics defined as follows:

$$D_{acc} = \frac{\sum_{p \in P_M} |D_T(p) \cap D_M(p)|}{\sum_{p \in P_M} |D_M(p)|} \times 100\%, \quad E_{acc} = \frac{|E_T \cap E_M|}{|E_{\text{comm}}|} \times 100\%,$$

$$G_{complete} = \frac{|E_T \cap E_M|}{|E_M|} \times 100\%$$

D_{acc} (*Dependency Accuracy*) measures how precisely a tool recovers direct runtime dependencies for packages observed in memory. E_{acc} (*Edge Accuracy*) measures how accurately the tool identifies runtime dependency edges for the packages that both the tool and *MEM-SBOM* have in common (E_{comm}), while $G_{complete}$ (*Graph Completeness*) evaluates global coverage by comparing all runtime edges in memory with those reconstructed by the tool, regardless of node presence.

Vulnerability Detection. Let V_S, V_T, and V_M denote the sets of vulnerabilities reported by *Grype* when analyzing SBOMs generated using `pip-audit` (static ground truth), the evaluated tool, and *MEM-SBOM*, respectively. The static

vulnerability baseline is obtained by executing `pip-audit` on the packages reported by `pip freeze`.

$$C_V = \frac{|V_T \cap V_M|}{|V_M|} \times 100\%, \quad P_V = \frac{|V_T \cap V_M|}{|V_T|} \times 100\%,$$

$$F_V = 2 \cdot \frac{C_V \cdot P_V}{C_V + P_V}, \quad M_V = \frac{|(V_S \cap V_M) \setminus V_T|}{|V_M|} \times 100\%,$$

$$R_V = \frac{|V_M \setminus (V_S \cup V_T)|}{|V_M|} \times 100\%.$$

C_V (Coverage) measures the proportion of runtime vulnerabilities (i.e., those identified by *MEM-SBOM*) that are correctly identified by the tool. P_V (*Precision*) measures the proportion of tool-reported vulnerabilities that are present at runtime. F_V (F1 Score) provides the harmonic mean of precision and recall, capturing the overall balance between detection completeness and correctness. M_V (*Missing*) measures vulnerabilities that appear in both the static ground truth and memory but are missing from the tool's output. Finally, R_V (*Runtime-Only*) represents the fraction of vulnerabilities associated with runtime-only packages.

Vulnerability Impact and Reachability. To measure how a vulnerable dependency propagates through the application's runtime graph, we employ three structural metrics. **DEC (Direct Exposure Count)** measures the number of packages that directly depend on the vulnerable package, indicating the extent of immediate exposure. **RP (Reachability Percentage)** quantifies the fraction of all packages in the runtime graph that have a direct or transitive path to the vulnerable package, capturing the scope of potential impact. **MDD (Minimum Dependency Distance)** reports the shortest path from the main application package (source) to the vulnerable package (target), representing the depth of remediation required.

6.3. Evaluation Results

This section presents and discusses the experimental results that answer the evaluation research questions.

6.3.1. RQ1: Package Recovery

Table 3 summarizes the evaluation results of *MEM-SBOM*'s extraction accuracy using 23 representative applications, each chosen to illustrate a distinct category from the complete set of 51 applications. The results show the exact matching between the package metadata recovered from memory by inspecting the `_path_cache` structure and the static ground truth obtained from `pip freeze`. This validates the ability of *MEM-SBOM* to accurately reconstruct the name and version of the packages installed in the deployment environment without relying on filesystem artifacts. Similarly, the comparison between the runtime ground truth and the modules recovered from `sys.modules` yields a perfect match, demonstrating the accuracy of *MEM-SBOM* in capturing the subset of installed packages that the application actually imports during execution. As anticipated, not all installed packages are imported at runtime. This discrepancy arises from optional and conditional dependencies, platform-specific packages, and build-time utilities that are never used during execution. For instance, in system-wide installations, the Python environment includes numerous OS-bundled packages that are present on the filesystem but never imported by the application.

Table 3
Accuracy of *MEM-SBOM* in recovering installed, imported, and dynamically loaded packages.

Application	Static Packages		Imported Packages		Dynamic
	Static Ground Truth	Memory (_path_cache)	Runtime Ground Truth	Memory (sys.modules)	Modules
Ajenti-Panel	50	50	40	40	0
Apache Superset	136	136	102	102	2
APScheduler	10	10	9	9	0
Beets	21	21	20	20	1
Daphne	21	21	16	16	0
Datasette	27	27	21	21	1
DevPi-Server	66	66	45	45	0
Django	5	5	5	5	0
Errbot	33	33	30	30	0
FScociety	14	14	12	12	0
Glances	5	5	35	35	31
iRedis	14	14	14	14	0
JupyterLab	91	91	69	69	0
Lektor	24	24	23	23	0
Locust	30	30	26	26	0
Mayan-EDMS	146	146	110	110	2
MLflow	64	64	29	29	0
Modoboa	86	86	60	60	2
RPyC	4	4	2	2	0
RQ	4	4	4	4	0
Scapy	1	1	1	1	0
Scrapy	37	37	28	28	1
Trytond	15	15	14	14	0

The results also illustrate a set of dynamic package modules that appear only at runtime. Such modules typically originate from packages dynamically loaded via `importlib.import_module()` or `__import__()`, side-loaded packages bundled directly within the application directory, or plugin frameworks that introduce additional modules during execution. *Glances* shows the largest runtime expansion (31 modules) due to its plugin architecture, which dynamically imports monitoring extensions from `glances/plugins/*.py` as independent top-level modules (e.g., `glances_cpu`, `glances_network`) via `importlib`, rather than hierarchical modules. Meanwhile, *Errbot*, *Apache Superset*, and *Modoboa* inject deployment-specific configurations through direct imports; *Datasette*, *Scapy*, and *Mayan-EDMS* expose backward-compatibility wrappers that create duplicated namespaces at runtime (e.g., `multipart`/`python_multipart`, `attr`/`attrs`).

Table 4 highlights the robustness of *MEM-SBOM* across six heterogeneous installation environments for the *Flask* web framework, representing diverse packaging ecosystems. Despite substantial variation in dependency resolution and metadata formats (e.g., `.dist-info` and `.egg-info`) across `pip`, `Conda`, `Poetry`, and system-level package managers, *MEM-SBOM* consistently enumerated the same eight imported packages of the application (i.e., `Flask`, `Itsdangerous`, `Jinja2`, `Blinker`, `click`, `Markupsafe`, `zipp`, and `Werkzeug`), demonstrating environment-independent extraction accuracy. However, the small differences in package counts reflect expected installation artifacts. Source-based installations introduce build tools (e.g. `wheel`) as the result of compiling the source archives by the package manager `pip`. *Conda* inherits 78 user-scope packages unless `PYTHONNOUSERSITE=1` is set. *Poetry*

Table 4
Accuracy of *MEM-SBOM* in recovering the *Flask*'s packages across heterogeneous installation environments.

Install Method	Environment Type	Static Packages Env.	Static Packages System	Imported Packages	Flask Version
pip (wheel, PyPI)	venv	12	0	8	3.0.3
pip (source, PyPI)	venv	13	0	8	3.0.3
pip (GitHub)	venv	12	0	8	3.0.3
Conda (user inherit)	Conda + user scope	0	78	8	3.0.3
Poetry	poetry venv	12	0	9	3.0.3
System (APT)	System-wide	0	146	9	1.1.1

introduces a virtual infrastructure module (`_virtualenv`) for environment isolation while *APT*-based installations injects `apport_python_hook`, a system-level crash-reporting module. However, *APT*-based installations show the highest heterogeneity, combining 76 user-level and 70 system-wide packages across `site-packages` and `dist-packages`. When Flask is installed system-wide, its version and dependency set depend on the operating system's bundled Python packages. As a result, the application inherits legacy builds and distribution-specific variants (e.g., `Flask 1.1.1`) as distributed with Ubuntu 20.04, illustrating version divergence and mixed provenance that *MEM-SBOM* effectively exposes.

Beyond reconstructing an accurate runtime state from memory, *MEM-SBOM* identifies inconsistencies between the declared package version at installation and the version embedded in the loaded code. Table 5 highlights ten representative cases in our evaluation set where *MEM-SBOM* revealed such mismatches that metadata-based tools failed to detect. These inconsistencies arise for several reasons. First, version attributes embedded in the source code may remain unchanged across releases, leading to incorrect version values in memory. Second, development builds may contain prerelease identifiers that are not reflected in the package manifests. Third, wrapper or alias artifacts may expose imported module names that diverge from the identifiers of the underlying distribution packages.

The Ajenti-Panel case exemplifies the third category, where `beautifulsoup4==4.13.4` provides the `bs4` import namespace, while a separate compatibility package `bs4==0.0.2` is installed in the same environment. This results in two distribution packages mapping to a single runtime namespace. Such mismatches can conceal vulnerable versions or enable dependency confusion, misleading tools that rely solely on static metadata. By validating version information directly from memory, *MEM-SBOM* can prove whether the installed packages align with their declared provenance, a critical capability for generating accurate SBOMs.

Answer to RQ1

MEM-SBOM achieves 100% coverage of the installed, imported, and dynamically loaded packages across heterogeneous deployment environments.

Table 5
Version inconsistencies between package metadata and embedded code as detected by *MEM-SBOM*.

Application	Dependency	Install Version	Memory Version
Ajenti-Panel	beautifulsoup4	0.0.2	4.13.4
DevPi-Server	strictyaml	1.7.3	1.6.2
Jet-Bridge	regex	2024.11.6	2.5.148
Lektor	lazy_object_proxy	1.11.0	1.10.0
Mayan-EDMS	regex	2025.7.34	2.5.159
Modoboa	oath	1.4.4	1.4.0
MyCLI	pymysql	1.1.2	1.4.6
Prefect	regex	2025.9.1	2.5.161
Rasa	terminaltables	3.1.10	3.1.0
Spyder	textdistance	4.6.3	4.6.2

Table 6
Evaluation of vulnerability propagation for the *tornado* package across the evaluated applications.

Application	Version	#Pkg	DEC	RP (%)	MDD	Dep. Type
BentoML	1.3.5	54	2	7.4	**2**	Transitive
Flower	2.0.1	19	1	5.3	1	Direct
Jet-Bridge	1.12.1	35	1	2.9	1	Direct
JupyterLab	4.3.8	70	**8**	**14.3**	1	Direct
Spyder	6.0.8	105	3	6.7	**2**	Transitive
Streamlit	1.40.1	17	1	5.9	1	Direct

6.3.2. RQ2: Vulnerability Reachability

Using *Grype*, *MEM-SBOM* identifies all known vulnerabilities present in the evaluated applications and their dependent packages, including *urllib3*, *tornado*, and *werkzeug* (Table 13, Appendix A). As a representative case study, we focus on the *tornado* package, which appears across multiple applications and includes vulnerable routines located in the `tornado.httputil` module, specifically, `_unquote_cookie()`, `parse_cookie()`, `parse_body_arguments()`, and `parse_multipart_form_data()`. These functions are affected by *CVE-2024-52804* (unsafe HTTP header parsing) and *CVE-2025-47287* (unsafe multipart parsing). Despite the availability of patched releases, several applications continue to rely on outdated versions. We found that *BentoML*, *Flower*, *Jet-Bridge*, *Spyder*, and *Streamlit* embed *tornado* 6.4.2, while *JupyterLab* bundles *tornado* 5.1.1.

To illustrate how this vulnerable dependency propagates through each application, Table 6 reports the DEC (Direct Exposure Count), RP (Reachability Percentage), and MDD (Minimum Dependency Distance). All applications except *BentoML* and *Spyder* depend on *tornado* directly (MDD=1). *JupyterLab* exhibits the widest reach, with eight of its packages linked to *tornado* (DEC=8, RP=14.3%), whereas smaller frameworks such as *Streamlit* and *Jet-Bridge* show limited propagation (RP=2.9% and 5.9%, respectively). These variations illustrate how an application's architecture determines the extent to which a vulnerable dependency propagates through its components. However, the presence of a vulnerable package does not necessarily imply that its vulnerable functionality is invoked at runtime.

Package-level scanning, therefore, overestimates risk. Figure 2 illustrates that although `spyder` has a transitive dependency path to `tornado.ioloop` module via the `zmq`

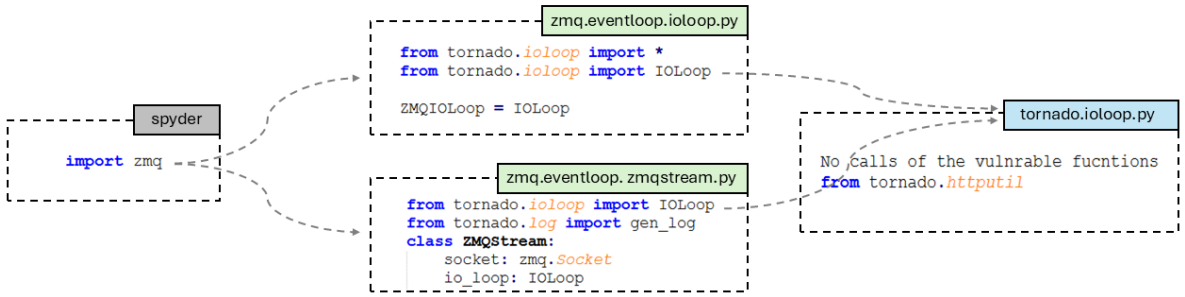

Figure 2: Transitive dependency path from `spyder` to the vulnerable *tornado* package via `zmq`, showing that no vulnerable functions are reached

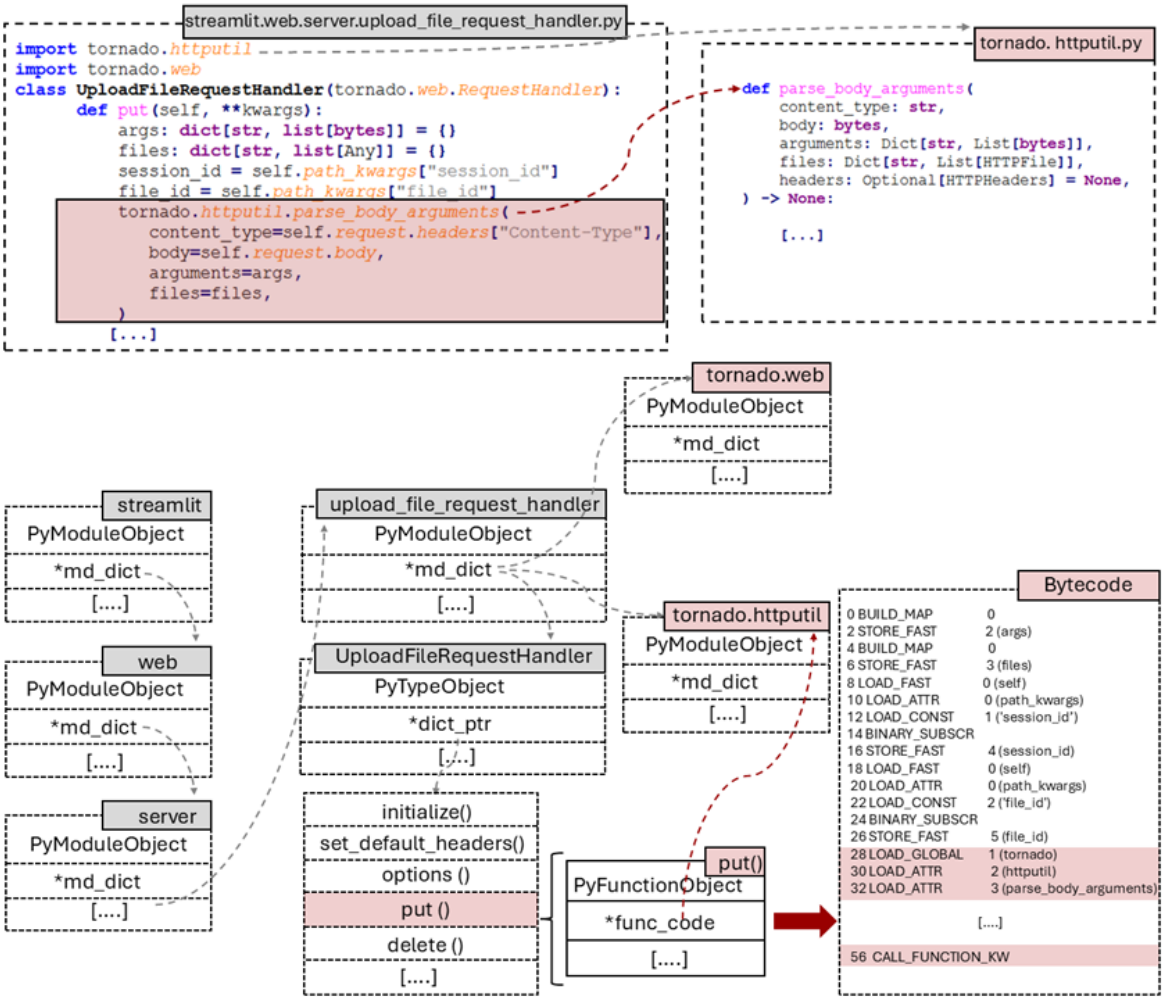

Figure 3: Direct call path from `streamlit` to the vulnerable `tornado.httputil.parse_body_arguments()` function via the `put()` method, showing that the vulnerable function is reached

package, none of the vulnerable `httputil` functions are used. In contrast, Figure 3 shows how `streamlit` reaches the vulnerable `parse_body_arguments()` function through its `UploadFileRequestHandler.put()` method, traversing the `web`, `server`, and `upload_file_request_handler` modules. The bytecode of `put()` confirms a direct call to the vulnerable function.

Table 7 summarizes the function-level reachability results. While all six applications are flagged as vulnerable at the package level, the detailed inspection reveals that five of them never execute the affected routines. For instance, *Flower* imports several *tornado* modules (e.g., `web`, `ioloop`, `httpserver`), yet none originate from `tornado.httputil`. *JupyterLab* imports `httputil` but only uses benign routines

Table 7
Function-level reachability analysis for the *tornado* vulnerabilities. Only `Streamlit` reaches a vulnerable function at runtime, while all other applications import only safe components.

Application	Tornado Usage Summary	Vulnerable Module Imported?	Vulnerable Function Reached?	Result
BentoML	`ioloop, gen, concurrent`	No	No	**Safe**
Flower	`web, ioloop, httpserver, escape, log`	No	No	**Safe**
Jet-Bridge	`ioloop, gen, web, iostream`	No	No	**Safe**
JupyterLab	`httputil, url_concat(), HTTPHeaders.parse_line()`	Yes	No	**Review**
Spyder	`ioloop, httpclient`	No	No	**Safe**
Streamlit	`httputil.parse_body_arguments()`	Yes	**Yes (CVE-2025-47287)**	**Vulnerable**

Summary: Package scanners flag 6/6 (100%), function-level analysis flags 1/6 (16.7%).

such as `url_concat()` and `HTTPHeaders.parse_line()`, neither of which transitively invoke any vulnerable function. Only *Streamlit* demonstrates a verifiable exploitation path, making it the only application where the vulnerability is reachable in memory. Thus, while package-based scanners flag 6/6 applications (100%), function-level validation identifies only one true exposure (16.7%), eliminating 83.3% of false positives. This granularity shows that only *Streamlit*'s upload handler requires remediation rather than upgrading *tornado* across all dependent projects.

Answer to RQ2

MEM-SBOM accurately identifies which applications reach vulnerable routines at runtime, revealing that only *Streamlit* invokes the affected *tornado* functions associated with *CVE-2025-47287*, demonstrating precise vulnerability reachability at runtime.

6.3.3. RQ3: Comparison with SBOM Tools

This section first analyzes the limitations of existing SBOM tools in parsing and supporting heterogeneous metadata formats across the evaluated applications. It then compares the runtime package coverage, dependency-graph accuracy, and vulnerability detection effectiveness of tools capable of analyzing deployed environments, using *MEM-SBOM* as the runtime ground truth.

Metadata-Based Analysis. We collected applications from both *PyPI* and *GitHub* repositories. We observed that 30 *GitHub* projects include a `requirements.txt` file, a format supported by all tools, which explains the higher success rates observed in the *GitHub* dataset. The applications are grouped by metadata configuration into PEP 621-compliant projects (`pyproject.toml`), Poetry-managed projects (`pyproject.toml` + `poetry.lock`), and legacy or hybrid builds relying on `setup.py` and `setup.cfg`. The final group, `requirements.txt` (combined metadata), denotes projects that include a `requirements.txt` file alongside other metadata files. Tables 8 and 9 highlight systematic weaknesses in metadata parsing across all tools. For PEP 621 projects, six tools (*Syft*, *Trivy*, *CDXGen*, *CycloneDX-Python*, *ORT*, and *Jake*) fail completely in both datasets, while only *PIP-SBOM* and *SBOM4Python* succeed (100%). In Poetry-managed projects, reliability depends almost entirely on the presence of a lock file. without `poetry.lock`, all tools except *PIP-SBOM* fail to resolve dependencies; with it, most succeed except *SBOM4Python*, which lacks native support for `[tool.poetry.dependencies]`. *ORT* achieves partial success (66.7%) owing to an outdated Poetry runtime that cannot parse the `priority` field introduced in version 1.2.

For legacy `setup.py`–based projects, success rates vary widely due to the imperative and non-standard nature of dependency declarations. *Trivy*, *CycloneDX-Python*, and *Jake* provide no `setup.py` support (0%). Among regex-based tools, *Syft* captures only pinned dependencies (35.71% PyPI, 20% GitHub), while *CDXGen* performs slightly better (42.86% / 80%) but fails on multiline or variable-defined lists and `extras_require` fields. *SBOM4Python* gives the highest coverage among pattern-matching tools (57.14 % / 60%) yet remains limited to single-line `install_requires`, producing empty SBOMs when absent (e.g., *Glances*). *ORT* achieves comparable rates (42.86% / 80%) by executing `setup.py` and resolving dependencies through PyPI queries, a more flexible but nondeterministic strategy. Only *SBOM4Python* and *PIP-SBOM* correctly interpret auxiliary `setup.cfg` configurations, yielding complete outputs for hybrid declarative-imperative builds.

Although all evaluated tools support the `requirements.txt` file, they differ significantly in version handling and completeness. *Syft* and *Trivy* ignore unpinned dependencies (`==`), omitting over half of declared packages. *CDXGen* retains all constraints, while *SBOM4Python* and *ORT* include unpinned entries but assign versions as `none`, relying respectively on regex parsing and the `pip-requirements-parser` library (nexB, 2025). *Jake* and *CycloneDX-Python* exhibit inconsistent behavior due to API divergence; older parser versions in *Jake* omit unpinned entries, while newer releases used by *CycloneDX-Python* include them unconditionally. None of the tools correctly interpret pip-specific directives such as `-r` (recursive inclusion) or `-e` (editable installations), resulting in systematically incomplete SBOMs. Notably, *MEM-SBOM* is not affected by metadata configuration, producing precise runtime-grounded dependency inventories that reflect actual execution.

These metadata-parsing limitations raise the question of whether SBOM tools can more accurately represent the deployed software state when analyzing installed environments rather than source metadata. To address this, we evaluate their performance in real runtime deployments. Notably, all

Table 8
Success rates (%) of SBOM tools across metadata configurations for 51 Python applications (PyPI dataset).

Metadata Configuration	#Apps	Syft	Trivy	CDXGen	CycloneDX-Python	SBOM4Python	ORT	Jake	PIP-SBOM	MEM-SBOM
pyproject.toml (PEP 621 / setuptools)	8	0/8 (0%)	0/8 (0%)	0/8 (0%)	0/8 (0%)	8/8 (100%)	0/8 (0%)	0/8 (0%)	8/8 (100%)	8/8 (100%)
pyproject.toml (Poetry-managed)	4	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	4/4 (100%)	4/4 (100%)
setup.py ± setup.cfg	14	5/14 (35.71%)	0/14 (0%)	6/14 (42.86%)	0/14 (0%)	8/14 (57.14%)	6/14 (42.86%)	0/14 (0%)	13/14 (92.86%)	14/14 (100%)
pyproject.toml + setup.cfg	4	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	4/4 (100%)	0/4 (0%)	0/4 (0%)	4/4 (100%)	4/4 (100%)
pyproject.toml + setup.py ± setup.cfg	7	1/7 (14.29%)	0/7 (0%)	3/7 (42.86%)	0/7 (0%)	5/7 (71.43%)	2/7 (28.57%)	0/7 (0%)	5/7 (71.43%)	7/7 (100%)
requirements.txt ± (combined metadata)	14	10/14 (71.43%)	6/14 (42.86%)	10/14 (41.43%)	14/14 (100%)	14/14 (100%)	7/14 (50%)	13/14 (92.86%)	14/14 (100%)	14/14 (100%)

Table 9
Success rates (%) of SBOM tools across metadata configurations for 51 Python applications (GitHub dataset).

Metadata Configuration	#Apps	Syft	Trivy	CDXGen	CycloneDX-Python	SBOM4Python	ORT	Jake	PIP-SBOM	MEM-SBOM
pyproject.toml (PEP 621 / setuptools)	4	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	4/4 (100%)	0/4 (0%)	0/4 (0%)	4/4 (100%)	4/4 (100%)
pyproject.toml (Poetry-managed)	3	3/3 (100%)	3/3 (100%)	3/3 (100%)	3/3 (100%)	0/3 (0%)	2/3 (66.67%)	3/3 (100%)	3/3 (100%)	3/3 (100%)
setup.py ± setup.cfg	5	1/5 (20%)	0/5 (0%)	4/5 (80%)	0/5 (0%)	3/5 (60%)	4/5 (80%)	0/5 (0%)	5/5 (100%)	5/5 (100%)
pyproject.toml + setup.cfg	2	0/2 (0%)	0/2 (0%)	0/2 (0%)	0/2 (0%)	1/2 (50%)	0/2 (0%)	0/2 (0%)	2/2 (100%)	2/2 (100%)
pyproject.toml + setup.py ± setup.cfg	7	0/7 (0%)	1/7 (14.29%)	2/7 (28.57%)	0/7 (0%)	5/7 (71.43%)	3/7 (42.86%)	0/7 (0%)	7/7 (100%)	7/7 (100%)
requirements.txt ± (combined metadata)	30	23/30 (76.67%)	17/30 (73.91%)	27/30 (90%)	29/30 (96.67%)	30/30 (100%)	18/30 (60%)	30/30 (100%)	25/30 (83.33%)	30/30 (100%)

subsequent runtime experiments are conducted on the PyPI-deployed versions of the applications.

Deployment-Based Analysis. We evaluated how *Syft*, *CycloneDX-Python*, *CDXGen*, and *PIP-SBOM* tools perform in real deployment environments, where dependencies are installed, version-resolved, and dynamically loaded. This analysis verifies whether these tools can accurately represent the runtime software state and produce accurate dependency graphs, using *MEM-SBOM* as the runtime ground truth. Given that these tools enumerate only direct dependencies, our comparison focuses on first-order dependency relationships, excluding transitive dependencies inferred through indirect imports.

As shown in Table 10, both *Syft* and *CycloneDX-Python* demonstrate stable behavior with high coverage (94.81% and 90.9%) and near-perfect version accuracy (>99.9%) by directly inspecting installed distributions and parsing metadata from the `site-packages` directory of the active virtual environment. Consequently, they construct realistic dependency graphs that closely reflect the deployed state, achieving the highest dependency accuracy, edge accuracy, and overall graph completeness.

In contrast, *CDXGen* and *PIP-SBOM* exhibit predictive behavior and yield the weakest results across all metrics. Although *CDXGen* infers dependencies by creating a temporary virtual environment and installing dependencies using the `-install-deps` flag, it still relies on static metadata analysis to construct its SBOM rather than observing actual runtime imports. *PIP-SBOM* instead resolves constraints in metadata files by predicting what dependencies *would* be installed rather than those actually present. This predictive strategy introduces false positives, omits conditional or optional dependencies, and frequently misreports versions when resolution assumptions diverge from the real deployment, resulting in substantially lower dependency accuracy ($D_{acc} < 50\%$) and incomplete graphs ($G_{complete} < 50\%$), consistent with their weaker runtime coverage.

MEM-SBOM, by directly enumerating the interpreter's loaded modules, eliminates these limitations and achieves complete coverage (100%) with 99.3% version accuracy due to inconsistencies between the declared package version at installation and the version embedded in the loaded code (see Table 5). However, all these tools are inherently static, missing packages that are dynamically loaded or generated at runtime.

Since *Grype* relies on the generated SBOMs, any inaccuracies in these SBOMs directly affect the reported vulnerability set, as illustrated in Table 11. *Syft* achieves full coverage ($C_V = 100\%$) equivalent to *MEM-SBOM*, while *CycloneDX-Python* provides near-complete detection (96.3%), missing only 3.7% of runtime vulnerabilities. In

Table 10
Performance of SBOM tools in deployment environments

Tool (%)	C_P (%)	M_P (%)	R_P (%)	V_{acc} (%)	D_{acc} (%)	E_{acc} (%)	$G_{complete}$ (%)
Syft	94.81	0.90	4.29	99.96	73.81	71.76	62.24
CDXGen	76.55	17.69	5.76	52.11	48.05	54.89	49.63
CycloneDX-Python	90.90	4.45	4.65	99.98	72.23	75.52	71.66
PIP-SBOM	73.42	21.96	4.61	65.22	27.90	34.15	20.09
MEM-SBOM	100.00	0.00	0.00	99.32	100.00	100.00	100.00

Table 11
The impact of SBOM tools on vulnerability detection.

Tool	C_V (%)	M_V (%)	R_V (%)	P_V (%)	F_V (%)
Syft	100.00	0.00	0.00	75.07	80.00
CDXGen	27.48	0.00	72.52	23.58	23.70
CycloneDX-Python	96.30	0.00	3.70	88.49	91.35
PIP-SBOM	37.37	0.00	62.63	33.80	34.67
MEM-SBOM	100.00	0.00	0.00	100.00	100.00

contrast, *CDXGen* and *PIP-SBOM* exhibit coverage below 40% and high omission ratios ($R_V > 60\%$), consistent with their incomplete runtime visibility. Notably, M_V is zero for all tools, indicating that vulnerabilities reported by both `pip audit` and *MEM-SBOM* are consistently detected; the missed vulnerabilities arise entirely from runtime-only packages. Finally, P_V decreases when tools report vulnerabilities associated with packages listed in metadata but never imported at runtime, thereby introducing false positives and reducing the overall F1 score (F_V). By restricting analysis to modules observed in memory, *MEM-SBOM* eliminates such false positives and achieves perfect precision, recall, and F1 ($P_V = R_V = F_V = 100\%$).

Answer to RQ3

Across heterogeneous metadata configurations and deployed environments, existing SBOM tools remain constrained by the metadata they parse, providing only partial visibility into the application runtime state. As a result, they generate incomplete SBOMs and dependency graphs, which in turn impact the accuracy of vulnerability detection.

7. Limitations and Future Work

This work focuses primarily on the Python runtime, excluding native C/C++ extensions that operate outside Python's managed memory. Such extensions introduce additional attack surfaces through language boundary transitions and type conversions (Scholtes et al., 2025). Extending *MEM-SBOM* to support polyglot correlation between Python objects and native components would enhance cross-language visibility. *MEM-SBOM* currently analyzes only components resident in process memory at the time of acquisition. Therefore, attacks that occur solely during installation remain unobserved. Integrating installation-phase monitoring or provenance tracing could capture such transient artifacts produced during setup routines and extend visibility beyond the runtime state. Vulnerability assessment depends on Grype's aggregation of public databases (e.g., NVD, OSV, and GitHub Security Advisories), limiting detection to known CVEs. Future work may incorporate semantic pattern analysis and anomaly detection over memory-derived artifacts to identify previously unseen vulnerabilities. Finally, we assume the integrity of the runtime and in-memory code objects. Prior work (Park et al., 2018) showed that memory-safety flaws in interpreters can corrupt bytecode while preserving valid metadata. Deep inspection of opcode sequences and Python-to-native wrapper transitions could further expose such tampering and injected behaviors.

8. Conclusion

In this paper, we introduced *MEM-SBOM*, the first memory-based framework that generates accurate SBOMs directly from the runtime state of Python applications. By traversing the interpreter registries, thread states, garbage collector, memory arenas, and heap regions, *MEM-SBOM* recovers all resident modules, even under evasive conditions. The extracted modules are grouped by their parent packages, filtered to exclude built-in and standard libraries, and assigned versions through a multi-source resolution process. For each third-party package, the framework analyzes its objects and disassembles the bytecode of its functions to construct execution-aware dependency graphs and perform fine-grained reachability analysis. Evaluation across 51 real-world Python applications demonstrated that *MEM-SBOM* achieves 100% extraction accuracy and detects ten inconsistency cases between the declared package version at installation and the version embedded in the loaded code. In a detailed analysis of six *tornado*-dependent applications, *MEM-SBOM* identifies `Streamlit` as the only application that executes the vulnerable routines at runtime, eliminating 83.3% of package-level false positives. Compared to eight state-of-the-art SBOM tools, *MEM-SBOM* also recovers all runtime-only dependencies, 4–6% of which are consistently missed by existing tools, resulting in more complete dependency graphs and more accurate vulnerability correlation. These results show that memory-based SBOMs provide a practical and forensically sound foundation for improving software supply chain security, particularly in dynamic language ecosystems where metadata and filesystem artifacts fail to reflect the true runtime state.

References

Ali, H., Case, A., Ahmed, I., 2025a. Leveraging memory forensics to investigate and detect illegal 3d printing activities. Forensic Science International: Digital Investigation 53, 301925. doi:10.1016/j.fsidi.2025.301925.

Ali, H., Case, A., Ahmed, I., 2025b. Memory analysis of the python runtime environment. Forensic Science International: Digital Investigation 53, 301920. doi:10.1016/j.fsidi.2025.301920.

Anchore, 2025a. Grype: Vulnerability scanner for container images and filesystems. https://github.com/anchore/grype. GitHub repository. Accessed: 2025-11-15.

Anchore, 2025b. Syft: A cli tool and library for generating sboms. https://github.com/anchore/syft. GitHub repository. Accessed: 2025-03-11.

Aqua Security, 2025. Trivy: A comprehensive and versatile security scanner. https://github.com/aquasecurity/trivy. GitHub repository. Accessed: 2025-03-11.

Benedetti, G., Cofano, S., Brighente, A., Conti, M., 2025. The impact of sbom generators on vulnerability assessment in python: A comparison and a novel approach, in: International Conference on Applied Cryptography and Network Security, Springer. pp. 487–509. doi:10.1007/978-3-031-95764-2_19.

Bi, T., Xia, B., Xing, Z., Lu, Q., Zhu, L., 2024. On the way to sboms: Investigating design issues and solutions in practice. ACM Transactions on Software Engineering and Methodology 33, 1–25. doi:10.1145/3654442.

Bidart, N., 2025. Django security releases issued: 5.2.8, 5.1.14, and 4.2.26. https://www.djangoproject.com/weblog/2025/nov/05/security-releases/. Accessed: 2025-11-11.

Bufalino, J., Di Francesco, M., Blaise, A., Secci, S., 2025. Sbomproof: Beyond alleged sbom compliance for supply chain security of container images. arXiv preprint arXiv:2510.05798 doi:10.48550/arXiv.2510.05798. preprint.

Chen, V., 2025. Awesome python. https://github.com/vinta/awesome-python. GitHub repository. Accessed: 2025-10-02.

Cofano, S., Benedetti, G., Dell'Amico, M., 2024. Sbom generation tools in the python ecosystem: An in-detail analysis, in: 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), IEEE. pp. 427–434. doi:10.1109/TrustCom63139.2024.00077.

Cybersecurity and Infrastructure Security Agency (CISA), 2021. Defending Against Software Supply Chain Attacks. Technical Report. U.S. Department of Homeland Security. TLP:WHITE. Accessed: 2025-02-15.

Cybersecurity and Infrastructure Security Agency (CISA), 2023. Types of software bill of material (sbom) documents. https://www.cisa.gov/sites/default/files/2023-04/sbom-types-document-508c.pdf. Accessed: 2025-03-11.

Cybersecurity and Infrastructure Security Agency (CISA), 2025. 2025 Minimum Elements for a Software Bill of Materials (SBOM): Public Comment Draft. Technical Report. U.S. Department of Homeland Security. Draft for public comment.

CycloneDX Project, 2025a. cdxgen: Universal polyglot sbom generator. https://github.com/CycloneDX/cdxgen. GitHub repository. Accessed: 2025-04-26.

CycloneDX Project, 2025b. Cyclonedx: A standard for software bill of materials (sbom). https://cyclonedx.org/. Accessed: 2025-03-12.

CycloneDX Project, 2025c. cyclonedx-python: Sbom generator for python projects. https://github.com/CycloneDX/cyclonedx-python. GitHub repository. Accessed: 2025-04-26.

Dalia, G., Visaggio, C.A., Di Sorbo, A., Canfora, G., 2024. Sbom ouverture: What we need and what we have, in: Proceedings of the 19th International Conference on Availability, Reliability and Security, Association for Computing Machinery. pp. 1–9. doi:10.1145/3664476.3669975.

Golubin, A., 2019. Memory management in python. https://rushter.com/blog/python-memory-managment/. Last updated July 8, 2019. Accessed: 2025-11-15.

Halbritter, A., Merli, D., 2024. Accuracy evaluation of sbom tools for web applications and system-level software, in: Proceedings of the 19th International Conference on Availability, Reliability and Security, Association for Computing Machinery. pp. 1–9. doi:10.1145/3664476.3670926.

Harrison, A., 2025. Sbom4python: Open-source tool to generate an sbom for installed python modules. https://github.com/anthonyharrison/sbom4python. GitHub repository. Accessed: 2025-04-26.

Kampourakis, V., Kavallieratios, G., Gkioulos, V., Katsikas, S., 2025. Cracks in the chain: A technical analysis of real-life supply chain security incidents. Computers & Security 159, 104673. doi:10.1016/j.cose.2025.104673.

Kawaguchi, N., Hart, C., 2024. On the deployment control and runtime monitoring of containers based on consumer side sboms, in: 2024 IEEE 21st Consumer Communications & Networking Conference (CCNC), IEEE. pp. 1022–1025. doi:10.1109/CCNC51664.2024.10454654.

Lakshmanan, R., 2025. Malicious python packages on pypi downloaded 39,000+ times, steal sensitive data. https://thehackernews.com/2025/04/malicious-python-packages-on-pypi.html. Accessed: 2025-11-10.

Mike Fiedler, 2023. Inbound malware volume report. https://blog.pypi.org/posts/2023-09-18-inbound-malware-reporting/. Accessed: 2025-11-10.

Mirakhorli, M., Garcia, D., Dillon, S., Laporte, K., Morrison, M., Lu, H., Koscinski, V., Enoch, C., 2024. A landscape study of open source and proprietary tools for software bill of materials (sbom). arXiv preprint arXiv:2402.11151 doi:10.48550/arXiv.2402.11151. preprint.

Nahum, M., Grolman, E., Maimon, I., Mimran, D., Brodt, O., Elyashar, A., Elovici, Y., Shabtai, A., 2024. Ossintegrity: Collaborative open-source code integrity verification. Computers & Security 144, 103977. doi:10.1016/j.cose.2024.103977.

National Counterintelligence and Security Center (NCSC), 2021. Software Supply Chain Attacks. Technical Report. Office of the Director of National Intelligence (ODNI). Accessed: 2025-11-15.

Nelson, N., 2023. Novel pypi malware uses compiled python bytecode to evade detection. https://www.darkreading.com/application-security/novel-pypi-malware-compiled-python-bytecode-evade-detection. Accessed: 2025-01-06.

nexB, 2025. pip-requirements-parser. https://github.com/nexB/pip-requirements-parser. GitHub repository. Accessed: 2025-10-05.

Office of the Director of National Intelligence (ODNI), National Security Agency (NSA), Cybersecurity and Infrastructure Security Agency (CISA), 2023. Securing the software supply chain: Recommended practices for software bill of materials consumption. https://www.cisa.gov/sites/default/files/2024-08/SECURING_THE_SOFTWARE_SUPPLY_CHAIN_RECOMMENDED_PRACTICES_FOR_SOFTWARE_BILL_OF_MATERIALS_CONSUMPTION-508.pdf. Accessed: 2025-03-11.

OSS Review Toolkit, 2025. Oss review toolkit (ort): Foss policy automation and sbom generation framework. https://github.com/oss-review-toolkit/ort. GitHub repository. Accessed: 2025-04-27.

Park, T., Lettner, J., Na, Y., Volckaert, S., Franz, M., 2018. Bytecode corruption attacks are real—and how to defend against them, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer. pp. 326–348. doi:10.1007/978-3-319-93411-2_15.

Python Software Foundation, 2024. Python Documentation: site — Site-specific configuration hook. https://docs.python.org/3/library/site.html. Accessed: 2025-01-09.

Samaana, H., Costa, D.E., Shihab, E., Abdellatif, A., 2025. A Machine Learning-Based Approach For Detecting Malicious PyPI Packages. Association for Computing Machinery. p. 1617–1626. doi:10.1145/3672608.3707756.

Scholtes, R., Khodayari, S., Staicu, C.A., Pellegrino, G., 2025. Charon: Polyglot code analysis for detecting vulnerabilities in scripting languages native extensions, in: 2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P), IEEE. pp. 153–168. doi:10.1109/EuroSP63326.2025.00018.

Sharma, A., Wittlinger, M., Baudry, B., Monperrus, M., 2024. Sbom.exe: Countering dynamic code injection based on software bill of materials in java. arXiv preprint arXiv:2407.00246 doi:10.48550/arXiv.2407.00246. preprint.

Smith, E.V., 2012. PEP 420: Implicit Namespace Packages. https://peps.python.org/pep-0420/. Accessed: 2025-10-30.

Snow, E., 2013. PEP 451: A ModuleSpec Type for the Import System. https://peps.python.org/pep-0451/. Accessed: 2025-10-30.

Software Package Data Exchange (SPDX), 2025. Spdx: Open standard for software bill of materials (sbom). https://spdx.dev/. Accessed: 2025-03-12.

Sonatype, 2025. 10th annual state of the software supply chain. https://www.sonatype.com/state-of-the-software-supply-chain/introduction. Accessed: 2025-03-10.

Sonatype Nexus Community, 2025. Jake: Dependency analysis and sbom generation tool. https://github.com/sonatype-nexus-community/jake/tree/803ec4f63e2c117352463065cf69d12a69f38450. GitHub repository. Accessed: 2025-10-02.

Stalnaker, T., Wintersgill, N., Chaparro, O., Di Penta, M., German, D.M., Poshyvanyk, D., 2024. Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, Association for Computing Machinery. pp. 1–13. doi:10.1145/3597503.3623347.

Stufft, D., 2015. PEP 503: Simple Repository API. https://peps.python.org/pep-0503/. Accessed: 2025-11-12.

Tooling and Implementation Working Group, 2024. Framing software component transparency: Establishing a common software bill of materials (sbom), third edition. https://www.cisa.gov/sites/default/files/2024-10/SBOM%20Framing%20Software%20Component%20Transparency%202024.pdf. Third Edition.

Volatility Foundation, 2025. Volatility 3: The volatile memory extraction framework. https://github.com/volatilityfoundation/volatility3. GitHub repository. Accessed: 2024-12-01.

Vu, D.L., Newman, Z., Meyers, J.S., 2023. Bad snakes: Understanding and improving python package index malware scanning, in: Proceedings of the 45th International Conference on Software Engineering (ICSE), IEEE. pp. 499–511. doi:10.1109/ICSE48619.2023.00052.

Wang, J., 2024. Malicious pypi package zlibxjson targets browser data and credentials. https://www.fortinet.com/blog/threat-research/malicious-pypi-package-zlibxjson-steals-browser-data. Accessed: 2025-11-10.

Wu, S.Q., Xu, H., 2016. Take it Easy, and Say Hi to This New Python Ransomware. https://www.fortinet.com/blog/threat-research/take-it-easy-and-say-hi-to-this-new-python-ransomware. Accessed: 2025-11-15.

Xia, B., Bi, T., Xing, Z., Lu, Q., Zhu, L., 2023. An empirical study on software bill of materials: Where we stand and the road ahead, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE. pp. 2630–2642. doi:10.1109/ICSE48619.2023.00219.

Yu, S., Song, W., Hu, X., Yin, H., 2024. On the correctness of metadata-based sbom generation: A differential analysis approach, in: 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE. pp. 29–36. doi:10.1109/DSN58291.2024.00018.

Zahan, N., Lin, E., Tamanna, M., Enck, W., Williams, L., 2023. Software bills of materials are required. are we there yet? IEEE Security & Privacy 21, 82–88. doi:10.1109/MSEC.2023.3237100.

A. Appendix

Table 12
Evaluation dataset of 51 Python applications spanning 23 categories.

Application Category	Application Name (Version)
Web Frameworks	Django (v4.2.23), Flask (v3.0.3), FastAPI (v0.116.1)
Web-based Environments	Jupyter Notebook (v7.4.5), JupyterLab (v4.3.8)
Task Scheduling Systems	Apache Airflow (v3.0.0), Celery (v5.5.3), APScheduler (v4.0.0a5), Prefect (v3.4.17)
Machine Learning Platforms	MLflow (v2.17.2), BentoML (v1.3.5), Seldon Core (v1.18.2), Rasa (v3.6.21), Gradio (v4.44.1)
Data Visualization	Apache Superset (v2.1.3), Orange (v3.38.1)
ASGI Servers	Daphne (v4.1.2), Uvicorn (v0.33.0), Hypercorn (v0.17.3)
Admin Panels	Ajenti-Panel (v2.2.11), Flower (v2.0.1), Jet-Bridge (v1.12.1), Wooey (v0.13.3), Streamlit (v1.40.1)
CLI Tools	iRedis (v1.15.2), LiteCLI (v1.13.2), MyCLI (v1.30.0)
Static Site Generators	Lektor (v3.3.12), MkDocs (v1.6.1), Pelican (v4.10.2), Nikola (v8.3.3)
Email Servers	Modoboa (v2.3.6), Salmon (v3.3.0)
Development Tools	Supervisor (v4.3.0), DevPi-Server (v6.15.0), Spyder (v6.0.8)
Database Tools	SQLite-Web (v0.6.4), Datasette (v0.64.3)
RPC Servers	RPyC (v6.0.2), Zerorpc (v0.6.3)
System Monitoring	Glances (v3.4.0.3)
Penetration Testing	FSociety (v3.2.9)
Network Tools	Mininet (v2.3.0.dev6), Scapy (v2.6.1)
Web Scraping	Scrapy (v2.11.2)
Audio Management	Beets (v2.2.0)
Chatbots	Errbot (v6.2.0)
Business Software	Trytond (v6.6.1)
Document Management	Mayan-EDMS (v4.9.2)
Task Queues	RQ (v2.3.3)
Testing Tools	Locust (v2.25.0)

Table 13
Detected vulnerable packages across the evaluated Python applications.

Project	Version	Affected Package	Package Version	Critical	High	Medium	Low
Ajenti-Panel	2.2.11	urllib3	2.2.3	–	–	2	–
BentoML	1.3.5	bentoml	1.3.5	3	2	1	–
		tornado	6.4.2	–	1	–	–
		starlette	0.44.0	–	–	1	–
DevPi-Server	6.15.0	waitress	3.0.0	1	1	–	–
		urllib3	2.2.3	–	–	2	–
Django	4.2.23	django	4.2.23	–	1	–	–
Errbot	6.2.0	cryptography	41.0.7	–	2	2	–
		jinja2	3.1.2	–	–	5	–
		requests	2.31.0	–	–	2	–
FastAPI	0.116.1	starlette	0.44.0	–	–	1	–
		urllib3	2.2.3	–	–	2	–
Flower	2.0.1	tornado	6.4.2	–	1	–	–
Fsociety	3.2.9	urllib3	2.2.3	–	–	2	–
Gradio	4.44.1	gradio	4.44.1	1	7	7	–
		starlette	0.44.0	–	–	1	–
		urllib3	2.2.3	–	–	2	–
Hypercorn	0.17.3	h2	4.1.0	–	–	1	–
Jet-Bridge	1.12.1	tornado	5.1.1	–	2	4	–
		pymongo	4.1.1	–	–	1	–
		urllib3	2.2.3	–	–	2	–
JupyterLab	4.3.8	tornado	6.4.2	–	1	–	–
		urllib3	2.2.3	–	–	2	–
Jupyter Notebook	7.4.5	jupyterlab	4.4.5	–	–	–	1
Lektor	3.3.12	werkzeug	2.3.8	–	1	2	–
		urllib3	2.2.3	–	–	2	–
Locust	2.25.0	flask_cors	5.0.0	–	–	3	–
		urllib3	2.2.3	–	–	2	–
Mayan-EDMS	4.9.2	django	4.2.23	–	1	–	–
		pypdf	5.1.0	–	–	1	–
		urllib3	1.26.20	–	–	1	–
MLflow	2.17.2	mlflow	2.17.2	–	–	3	–
		urllib3	2.2.3	–	–	2	–
MyCLI	1.30.0	sqlparse	0.4.4	–	1	–	–
		paramiko	2.11.0	–	–	1	–
Nikola	8.3.3	urllib3	2.2.3	–	–	2	–
Rasa	3.6.21	keras	2.12.0	1	1	1	–
		tensorflow	2.12.0	–	1	–	–
		future	1.0.0	–	1	–	–
		aiohttp	3.9.5	–	–	1	–
		pymongo	4.3.3	–	–	1	–
		urllib3	1.26.20	–	–	1	–
Scrapy	2.11.2	urllib3	2.2.3	–	–	2	–
Seldon Core	1.18.2	flask_cors	3.0.10	–	1	4	–
		gunicorn	20.1.0	–	2	–	–
		werkzeug	2.2.3	–	1	3	–
Spyder	6.0.8	tornado	6.4.2	–	1	–	–
		urllib3	2.2.3	–	–	2	–
		aiohttp	3.10.11	–	–	–	1
Streamlit	1.40.1	tornado	6.4.2	–	1	–	–
		urllib3	2.2.3	–	–	2	–
Superset	2.1.3	pyarrow	10.0.1	1	–	–	–
		cryptography	39.0.2	–	2	3	3
		sqlparse	0.4.4	–	1	–	–
		werkzeug	2.3.3	–	1	3	–
		flask_caching	–	–	–	1	–
		urllib3	2.2.3	–	–	2	–
Zerorpc	0.6.3	future	1.0.0	–	1	–	–