Supply Chain Reaction: Enhancing the Precision of Vulnerability Triage using Code Reachability Information

Harshvardhan Patel
Stony Brook University
New York, USA

Alexander Snit
Stony Brook University
New York, USA

Michalis Polychronakis
Stony Brook University
New York, USA

Abstract—**Software bill of materials (SBOMs) have become a crucial component of large-scale vulnerability triage. Existing SBOM approaches operate at the granularity of software packages or binary executables, providing a very coarse and *context-agnostic* view of a program's dependencies. In turn, if a third party component has a vulnerability, existing SBOM tools will report it for remediation irrespectively of whether a given dependent program is actually affected by that vulnerability or not. Although determining the exact conditions under which a vulnerability in a third-party component may affect a dependent program remains a challenging problem, a first robust approximation can be obtained by considering the *reachability* of the vulnerable code from the parent program.**

Based on this observation, the main goal of this work is to assess the improvement that code reachability information can offer in the precision of vulnerability triage for C/C++ programs. To that end, we developed VPChecker, a system that i) captures the dependencies of a program at the granularity of individual *functions*, and ii) localizes CVEs in shared libraries to the corresponding functions that contain the flaw. Using VPChecker, we conducted a large-scale vulnerability assessment of the Debian Linux ecosystem. The results of our cross-examination of over 24,000 C/C++ binaries (shipped by more than 6,600 Debian packages) with 510 CVEs in shared libraries, show that function-level reachability information reduces the number of ELF binaries reported as affected by a particular CVE by 28% on average, while reducing the number of CVEs reported as affecting a particular ELF binary by 30%.

Index Terms—**Vulnerability Triage, Code Reachability, Software Supply Chain**

1. Introduction

Vulnerability triage is the process of evaluating published software vulnerabilities based on their severity and impact in the context of a particular environment, such as an organization's IT infrastructure. This process helps organizations prioritize their remediation efforts according to the most critical vulnerabilities, while at the same time minimizing any potential disruptions due to patching going wrong.

Software bill of materials (SBOMs) have become a crucial mechanism for effective vulnerability triage and software supply chain risk management. Essentially, SBOMs provide a detailed inventory of all components, libraries, and modules in a software product, along with their versions and dependencies. Given the SBOMs of all deployed products and a feed of vulnerabilities, such as the common vulnerabilities and exposures (CVE) list, automated vulnerability assessment tools identify what components must be patched across an IT infrastructure to reduce the risk of exploitation.

Unfortunately, however, this approach has proved ineffective in practice [1]. The extremely large volume of existing CVEs, the continuous discovery of new vulnerabilities, and the sheer complexity of modern software, mean that the probability of a given product being affected by several CVEs at any given time is very high. Consequently, security administrators are faced with daunting lists of vulnerabilities that must be patched. Without any meaningful way to prioritize them and come up with an actionable mitigation strategy, many system administrators end up apprehensively deciding to apply patches after long delays [2], [3], [4].

This problem is severely exacerbated by the coarse granularity and context-agnostic nature of SBOMs, which are oblivious to the particular characteristics of each dependency. Existing SBOM approaches operate at the software package or binary executable level, providing a very coarse view of a program's dependencies. Consequently, if a third party component contains a vulnerability, existing vulnerability triage tools will report it for remediation *irrespectively* of whether a given program that depends on this component is actually affected by this vulnerability.

Determining the exact conditions under which a vulnerability in a third-party component may affect a given program in a certain environment is a challenging problem, and depends on a variety of factors, including whether and how untrusted input can reach the vulnerable component, the hardware and operating system characteristics of the host, and the presence of any exploit mitigations or other defense-in-depth solutions. We observe, however, that a first robust approximation can be obtained by considering solely the *reachability* of the vulnerable (parts of) code from the parent program. For example, a high-risk vulnerability in a shared library may actually be irrelevant (and its remediation could be safely deprioritized) if the vulnerable part of its code is not reachable under any conditions by a dependent program—in which case the vulnerability cannot be triggered in the first place. As prior research on attack surface reduction has shown [5], [6], [7], [8], [9], programs typically import and use only a *fraction* of a shared library's functions. Consequently, a vulnerability in a shared library may not pose a risk to a dependent program if there is no control flow path that reaches the vulnerable code.

Based on this observation, our main contribution is a large-scale assessment of the improvement that code reachability information can offer in the precision of vulnerability triage for C/C++ programs. To that end, we developed *VPChecker* (Vulnerable Path Checker), a system that i) captures the (direct and transitive) module dependencies of a program at the granularity of *individual functions*, by extracting and aggregating function call graphs; and ii) localizes CVEs in shared libraries to the corresponding functions that contain the flaw by automatically identifying the relevant parts of code from public software patches. Function call graphs and localized CVEs are then unified in a *supply chain knowledge graph*, which enables rapid and precise vulnerability triage at the function level.

We opted to implement VPChecker at the binary level rather than the source code level for several reasons. Although source code analysis offers more precise function call graphs due to the availability of rich compiler-level semantics, it is impractical at the scale of modern IT infrastructures. The diversity of build systems, the multitude of source code dependencies, and the need for recompilation for accurate global call graph extraction, make source-level analysis cumbersome. In fact, even open-source software is mostly deployed in the form of binaries through each OS distribution's package manager for convenience and scalability. Binary analysis facilitates the rapid generation of the call graphs required for conducting a large-scale study as ours (recompiling tens of thousands of binaries from scratch is infeasible), and for performing effective vulnerability triage across an entire organization's IT infrastructure.

Using VPChecker, we conducted a large-scale vulnerability assessment of the Debian Linux ecosystem, by analyzing more than 24,000 C/C++ ELF64 binaries shipped by more than 6,600 Debian packages. Our results show that function-level reachability information reduces significantly the number of false positives, and provides a more precise CVE impact assessment. Specifically, it reduces the average number of ELF binaries (directly or transitively) affected by a given CVE by 28%, and reduces the average number of CVEs affecting a given ELF binary by 30%, i.e., *roughly one third of the CVEs reported for a given application are false positives*. Given the overapproximated—but sound—nature of binary-level function call graph extraction, these numbers represent a lower bound, and could be further improved using more precise function call graph extraction techniques.

In summary, we make the following main contributions:

- We developed VPChecker, a system for rapid large-scale vulnerability triage at the function level for 64-bit C/C++ binaries.
- We highlight the absence of function-level details in current vulnerability feeds, and develop methods to automatically augment CVE reports with affected function names using automated patch analysis.
- We conducted a large-scale vulnerability assessment of Debian Linux using 24,076 dynamically linked C/C++ ELF64 binaries shipped by 6,632 Debian packages, and 510 CVEs in shared libraries.

Our implementation and data set are publicly available through https://github.com/harshvp1621/VPChecker.

2. Background

Vulnerability Triage. Vulnerability triage prioritizes remediation efforts for the most critical vulnerabilities. This assessment is based on exploitability, the presence of mitigations, the sensitivity of the affected assets, and the difficulty of implementing a fix. In practice, the process of automated vulnerability triage begins with just scanning dependencies for vulnerabilities. Most vulnerability scanners take an application's list of dependencies as input, and cross-reference their properties (e.g., version strings and checksums) against a database of known vulnerabilities [10], [11], [12], [13]. Upon a successful match, the dependency is flagged as vulnerable without conducting any further analysis [14]. However, this overlooks the fact that not all code present in a dependency will be executed by the program. As a result, some vulnerabilities may never be exploitable because they are simply *unreachable* under all possible program executions.

Software Bills of Materials. The first step in vulnerability triage is to identify all dependencies of the target software. To facilitate comprehensive, machine-readable and interoperable tracking of software components, the adoption of Software Bills of Materials (SBOMs) has gained traction [15]. The SBOM of a project can be defined as a structured way of representing its inventory, which captures the hierarchical relationships between all software *dependencies* used [16].

One of the recommended use cases for SBOMs is to aid in vulnerability triage [17]. It is therefore desirable for SBOMs to contain sufficient information not only to aid in the automated identification of vulnerable software components [18] but also to enable maintainers to analyze the reachability—and subsequently the exploitability—of vulnerabilities in their supply chain. Several vulnerability scanning tools [10], [11], [19], [12] parse dependency metadata from SBOMs and cross-reference them with various vulnerability databases to flag at-risk components. Often, one or more dependencies may have an associated Vulnerability Exploitability eXchange (VEX) [20] document that indicates the status of a vulnerability—*affected, not affected, fixed,* or *under investigation*—with respect to the component. However, the vulnerability status alone does not provide insight into how the vulnerable code interacts with the dependent program. This information can only be obtained through code analysis—a feature that SBOM scanning tools in the wild currently lack, as they track dependencies only at the package or binary level [13], [14].

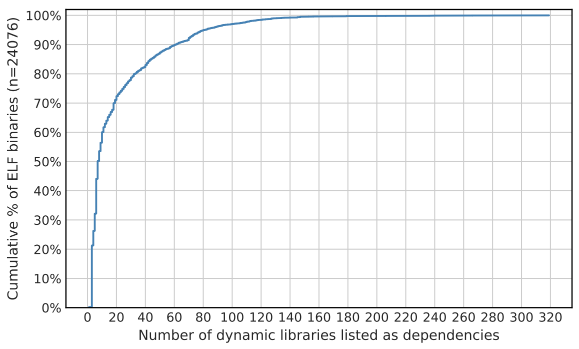

Figure 1: CDF of the total number of (direct and transitive) shared-library dependencies per binary across 24,076 Debian ELF binaries. About half of the binaries depend on 15 or more libraries.

3. Motivation

Dependency Management. Languages such as Java, Node.js, Python, and Rust have native package management infrastructures (Maven, npm, Pip, and Cargo, respectively) that serve as central repositories for hosting, distributing, and managing dependencies. In contrast, C/C++ projects *lack* a unified package management infrastructure. Instead, they rely on a diverse array of build systems to manage dependencies, which often are not interoperable [21]. Previous works on determining dependencies for C/C++ projects [21], [22] have noted that the lack of a unified package manager is one of the primary hindrances for performing large-scale empirical studies on the supply chain of this ecosystem.

In case of Linux distributions, C/C++ software is typically distributed in the form of compiled binaries bundled within packages. Most of these binaries follow the ELF file format and are dynamically linked to other ELF binaries (referred to as shared libraries) [23]. It is important to note that each package often contains multiple binaries. For example, the libssl3 Debian package ships six binaries [24]. However, binaries in other packages that declare a dependency on libssl3 are unlikely to rely on all six binaries it provides. More importantly, a vulnerability reported in the libssl3 *package* may not impact all of its included binaries. Unfortunately, some popular SBOM generation tools [25], [26] perform their dependency analysis just at the package level, thereby missing critical interdependencies at the binary level.

This observation underscores the necessity of tracking C/C++ dependencies at a granularity finer than the package level. Figure 1 shows the number of dependencies across the 24,076 64-bit ELF binaries in our dataset (15,827 main executables and 8,249 shared libraries). As shown in Figure 1, approximately half of the binaries depend on at least 15 dynamically linked libraries.

Vulnerability Scanning. Vulnerability scanners typically rely on querying a database to check for known vulnerabilities in a project's software dependencies [13], [14]. Using the lens of code reachability, we can assert that a project will only be at risk from vulnerable code that is reachable [27]—rendering any alerts reported due to unreachable vulnerabilities as *false positives*. The number of false positives generated by vulnerability scanners is exacerbated due to the dependency explosion illustrated in Figure 1, which often leads to vulnerability alert fatigue [28], [29].

Prior research in the Java ecosystem has demonstrated that a significant number of vulnerability alerts are false positives and can be filtered out by computing the reachability of vulnerable function(s) from the program's entry points. This is typically done either through manual inspection of API usage, or by constructing function call graphs using static or dynamic analysis on Java source code and binaries [30], [31], [27], [32].

To the best of our knowledge, a similar study has not been conducted for C/C++ software, and is the main motivation for our work. Conducting large-scale reachability analysis in this ecosystem presents several challenges. First, as previously discussed, there is no unified package management infrastructure for C/C++ code. Even within the sub-ecosystems of specific Linux distributions, dependency metadata is maintained at a very coarse granularity, i.e., at the package level, which is insufficient for precisely triaging vulnerabilities.

Second, most C/C++ software is distributed in the form of compiled (and often *stripped*) standalone binaries that lack high-level semantic information such as function names, type annotations, and class hierarchies. In general, precise disassembly of C/C++ programs is undecidable [33], [34]. The ubiquitous use of function pointers, manual memory management, and compiler optimizations makes recovering control-flow information from C/C++ binaries even more complex. In contrast, the Java ecosystem offers significant advantages for static analysis: Java bytecode retains rich type information, method signatures, and class hierarchies, enabling the construction of call graphs with higher precision. Furthermore, some studies move from bytecode to source code to further improve call graph accuracy [35], [31].

Given these challenges, in this work, we conduct the first large-scale function-level code reachability study on CVEs reported for C/C++ binaries shipped by a popular Linux distribution—Debian Sid [36]. Aware of the difficulties in extracting precise call graphs from binaries, we adopt a conservative method for call graph construction by prioritizing *soundness*. Our results show that, despite the inherent overapproximation of the extracted call graphs, function-level analysis significantly reduces the number of false positives.

4. Function-level CVE Triage

Challenge 1: Lack of Reachability Information in Existing SBOMs. On Linux systems, popular SBOM-generation tools query the OS package manager to retrieve a list of installed packages and their dependencies [25], [26]. Since this approach does not capture any dependencies on shared libraries, some tools parse the dynamic section of ELF binaries to identify dynamically linked libraries and generate an ELF-level SBOM [37]. While more fine-grained than

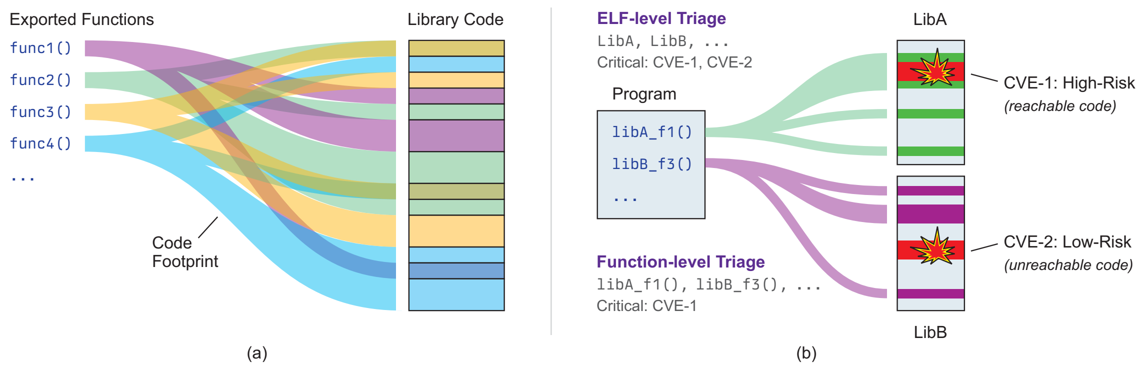

Figure 2: An exported function of a library has a "code footprint" that comprises all reachable code from its entry point (a). VPChecker performs vulnerability triage at the function level, considering the footprints of each imported function from external libraries, and thus it can deprioritize the remediation of vulnerabilities associated with unreachable code (b).

package-level analysis, ELF-level analysis still ignores what parts of code from each shared library are actually reachable.

Shared libraries expose their functionality mainly through a set of *exported functions*. In turn, programs explicitly *import* each function they want to use from external libraries. Extensive research on software debloating and attack surface reduction [38], [5], [6], [39], [40], [41], [42], [43], [44], [7], [8], [9], [45], [46], [47] has shown that in the vast majority of cases, programs use only a *fraction* of a library's exported functions. Each exported function is a potential entry point into the library and is associated with only a *subset* of the library's code, to which we refer as the function's *code footprint*, as illustrated in Figure 2(a). For a given vulnerability in a library, if the footprints of the library's functions imported by a program do not overlap with the vulnerable code, then the program cannot be affected by the vulnerability, simply because the flaw is unreachable.

As shown in Figure 2(b), tracing the actual code footprint of imported functions reveals vulnerabilities in third-party dependencies that are unreachable, thereby allowing these to be de-prioritized in favor of reachable vulnerabilities that pose a more concrete risk. In this example, relying on coarse-grained package-level or ELF-level reachability information would report both vulnerabilities as equally critical, while function-level reachability reports only CVE-1 as critical, because CVE-2 is not actually reachable.

Challenge 2: Lack of Vulnerable Code Location Information in CVE Records. Information about CVEs is publicly available in the form of CVE records [48]. Various public databases maintain these records, with those relevant to the Debian supply chain obtainable from the CVE Program [49] and the Debian Security Tracker [50].

The first challenge in CVE triaging for Debian binaries is specific to its package management infrastructure. In the Debian ecosystem, software is bundled into `deb` packages, which are of two types: *source* and *binary* [51]. As indicated by their names, source `deb` packages contain source code files that can be compiled into one or more binary `deb` packages. These binary packages include compiled executables and shared libraries. For instance, the binary packages `ffmpeg`, `libavfilter7`, and `libpostproc55` are all compiled from the source package `ffmpeg` [52].

Vulnerabilities in Debian, however, are reported *at the level of source packages*. Given that a source package can result in several binary packages, it is likely that a CVE reported in a source `deb` may only affect some—but not all—of the binary `deb` packages that are *compiled* from the problematic source code file(s). This discrepancy between source and binary packages results in dubious vulnerability reporting [22], as all binary packages originating from a source package are flagged as vulnerable—even those that do not contain the flaw. This is a limitation of many Linux vulnerability scanning tools, as documented in the open-source tool `debsecan` [53], which reports CVEs disclosed for `deb` packages installed on Debian hosts.

The next challenge in CVE localization is the lack of an explicit field in CVE records for specifying the names of the vulnerable functions or the affected code locations. The CVE Program mandates a minimum set of information to be published in each record [54], including the affected product, affected or fixed version(s), CVE ID, references (e.g., blog posts or patches), the vulnerability type or root cause, and a plain-English description. However, the mention of vulnerable function names in this description is often incomplete or entirely missing. In 2024, the CVE Program released a new schema that introduces the `programRoutines` field to address this gap [55], [56], but this field remains largely unused, with only 105 of 65,000 CVEs reported between 2022–2024 populating it [57]. Other vulnerability reporting standardization efforts, such as the OSV format [58], also attempt to address this issue by including explicit fields for vulnerable symbol names. However, this information is still largely absent [59], especially for C/C++ vulnerabilities.

As a result, existing SBOM-based vulnerability triage tools rely solely on component names and version strings provided in CVE records [10], [11], [60]. The lack of function-level details in C/C++ CVE records is thus a key roadblock in determining whether imported code actually overlaps with vulnerable code.

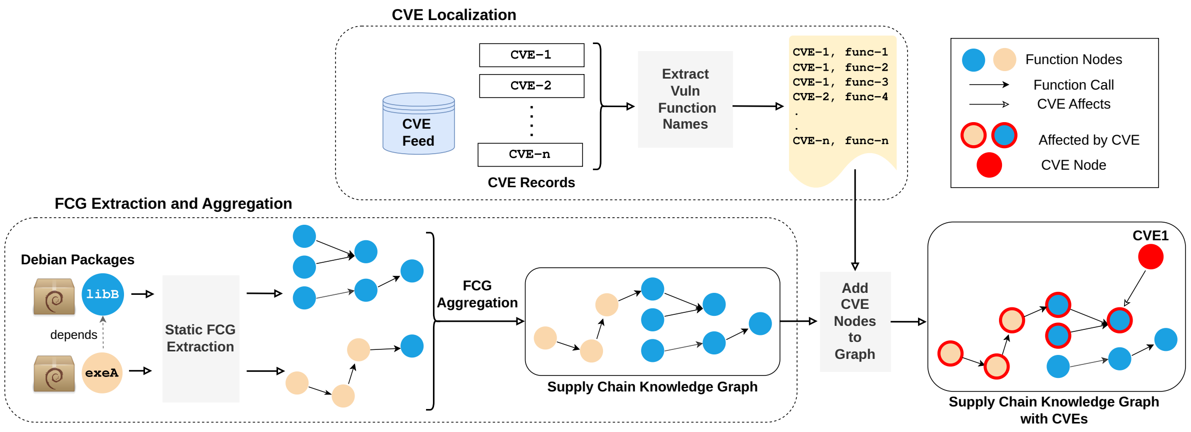

Figure 3: High-level overview of VPChecker's architecture. The FCG Extraction and Aggregation pipeline illustrates how the FCGs extracted from an executable and its library dependencies are aggregated to create a unified graph. A list of function names obtained from the CVE localization pipeline is then used to annotate vulnerable functions in the unified graph.

5. VPChecker Architecture

We introduce *Vulnerable Path Checker (VPChecker)*, a tool that analyzes dynamically linked C/C++ binaries to track their dependencies at the function level, and to identify function-call paths that reach vulnerable functions associated with CVEs in those dependencies. Using VPChecker, we studied vulnerable code reachability on a large scale across binaries shipped by `deb` packages in Debian Sid.

Figure 3 shows an overview of VPChecker, which consists of two main parts: i) the extraction and aggregation of function call graphs (FCG) from binaries shipped by `deb` packages, and ii) the localization of CVEs to the corresponding vulnerable function(s). The utility of combining FCGs from all component binaries lies not only in identifying the precise code footprint of all functions imported from shared libraries used by a given binary, but also in providing a bird's-eye view from a library's perspective—revealing the impact of its exported functions and, by extension, its internal code on the entire supply chain.

The CVE Localization pipeline then enables us to *localize* CVE information to the actual vulnerable functions. This is done by systematically scraping all publicly available patch links available in CVE records, and heuristically (and conservatively) extracting a list of functions that were modified as part of the fix. For each CVE, we introduce a CVE node into the unified graph and connect it via edges to all functions modified across its patches. This allows us to *localize* each modified function to the final compiled binary that will ultimately run the vulnerable code—a crucial mapping unavailable in any existing open-source vulnerability database. Borrowing terminology from traditional supply chain literature [61], we refer to our unified function-level graph, annotated with vulnerability information, as Debian Sid's *supply chain knowledge graph*. As new CVE nodes are added to the graph, only those programs with a direct code path to a given CVE are considered at risk, regardless of any higher-level dependencies.

5.1. Function-level Dependency Graphs

5.1.1. Source Code vs. Binary Analysis. Program analysis can be performed either at the source or at the binary level. Source-level analysis benefits from rich semantic information, enabling higher precision. In contrast, binary-level analysis faces significant challenges in recovering high-level semantics, resulting in over-approximations for control flow recovery, type inference, and related tasks [34].

For our purposes, a C/C++ program's FCG can be extracted from either its source code (e.g., using a compilation pass) or its binary executable (e.g., using binary analysis tools). Although our focus is on the Linux ecosystem—where all packages in major distributions are open-source—we chose to not rely on source code-based FCG extraction for the following reasons.

First, source-level analysis does not scale to the size and complexity of modern IT infrastructures. Extracting FCGs typically requires compiler-level tools (e.g., additional compiler passes) [62]. Requiring administrators to retrieve the code of all deployed software, configure the necessary build tools and dependencies, and recompile all programs, is unrealistic in terms of time, scale, effort, and required expertise. The Debian Package Rebuild project [63], which attempts to recompile the entire Debian archive using `clang`, underscores the high computational cost of such efforts, and the associated challenges, as many packages still fail to build. Additionally, the diversity of build systems across C/C++ packages (e.g., `make`, `cmake`, `ninja`) [64] further complicates compiler-based call graph extraction, particularly when linking source files to their correct binary targets [62].

Second, the vast majority of C/C++ open-source software is deployed in the form of binaries distributed via package management tools such as `apt`. This distribution model offers unparalleled convenience and scalability compared to having to compile each new or updated version of a program from source. At the same time, vulnerability triage and prioritization must be operational at scale, to offer meaningful response to the ever-increasing number of vulnerabilities being discovered and exploited in the wild.

For the above reasons, we focus on the generation of FCGs directly from binaries, to enable the practical and rapid construction of a function-level view of large software supply chains. Another advantage of this approach is that it is directly applicable to proprietary software, which may still have dependencies on open-source libraries, although its source code is not available.

5.1.2. Supply Chain Knowledge Graph. Given a program, VPChecker extracts the FCG of each direct and transitive ELF binary dependency and aggregates them into a unified graph capturing all function-level interdependencies. Each node represents a function, and directed edges denote caller–callee relationships within and across different binaries.

The FCGs of all binaries in our dataset are aggregated into a unified supply chain knowledge graph, stored in a graph database. This knowledge graph includes two types of nodes: i) *function* nodes, derived directly from the *FCG*; and ii) *bridge* nodes, added for each binary that has at least one indirect function call site. Each bridge node receives an incoming edge from every function that includes at least one indirect call site. Correspondingly, an outgoing edge is added from the bridge node to every address-taken function in the binary, conservatively connecting all indirect function call sites to all potential indirect call targets.

For our experimental evaluation, we construct a separate graph characterizing the Debian supply chain at the ELF binary level. Each node in the ELF-level graph corresponds to a binary in our dataset, and each directed edge represents a dynamically-linked dependency between binaries. Hence, for each node in the ELF-level graph, a simple breadth-first search will return a list of all dynamically-linked dependencies—effectively an ELF-level SBOM. In essence, the nodes of the dependency tree derived from the ELF-level graph of a binary correspond to the list of shared libraries generated by the `ldd` utility [65] on Linux-based systems.

5.2. CVE Localization

Due to the lack of explicit code location information in CVE records, prior research in the area of large-scale analysis of software vulnerabilities resorted to scraping patches from the web pages pointed to by URLs typically found in the *references* field of various CVE record formats [66], [67], [68]. The links crucial for extracting the names of vulnerable functions typically point to Git commits.

Figure 4 provides an overview of our CVE localization pipeline. Stage (i) gathers relevant CVEs for each Debian source package that ships at least one dynamic shared library. In Stage (ii), we scrape the references listed in the NVD and Debian Security Feed entries for each CVE, searching for Git URLs that link directly to a commit. For some CVEs, references may not directly include a Git commit link, but may link to a bug hosting service such as Bugzilla [69]. For such cases, we attempt to extract Git commit links from the comments of the corresponding Bugzilla pages.

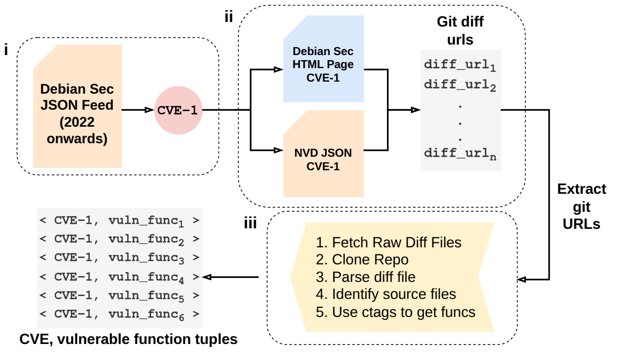

Figure 4: CVE localization process for pinpointing the functions that contain a given flaw.

Stage (iii) clones the repositories of the URLs extracted from the identified Git commit links. We then parse the patch files to identify the range of source code lines modified by the patch (known as *diff hunks*) [70]. This is done using the `ctags` [71] utility to record each function's start and end line numbers. Based on the line numbers indicated in the patch files, we determine which function's body encompasses the change. If such a function is identified, we extract its name and generate a tuple. The final output of this stage is a list of tuples, each representing a CVE ID and the corresponding affected function. Since a patch may modify multiple functions across different files, we conservatively consider the names of *all* altered functions as vulnerable, and include all of them in the final list of tuples.

6. Implementation

We implemented our framework using Debian Sid Docker [72] containers, to allow for clean installation (and rollback) of packages and all their dependencies. Installation of each `deb` package is carried out using `apt`, the default package manager for Debian, which installs all direct and transitive dependencies of the target `deb` package. Additionally, for each `deb` and its dependencies, we explicitly install the associated *debug* packages, if available. Debug packages are usually hosted on a separate repository from the one that hosts the `release` packages, and are typically named `<package_name>-dbg` or `<package_name-dbgsym>` [73].

For packages without a corresponding debug package (2,831 out of 6,632 in our dataset), the analysis proceeds without debug symbols. The absence of symbolic information implies that the actual names of the *internal* functions of a binary are not available, in which case a unique identifier is generated based on the function's position in the binary.

6.1. Function Call Graph Extraction

6.1.1. Handling Indirect Function Calls. Edges in a call graph represent either *direct* or *indirect* function calls. Extracting the targets of direct calls is straightforward, but precisely determining the possible targets of indirect calls is challenging, because the high-level type information and control-flow context needed to resolve these calls is lost during compilation. Consequently, binary code analysis tools, especially those in the area of retrofitting control flow integrity at the binary level [74], [75] typically resort to heuristics and approximations to infer a conservative set of potential targets for a given indirect call site [76].

For our use case, which is determining the reachability of vulnerable functions, the extracted call graph must be *sound*. We prioritize soundness over precision, because a vulnerability reachability analysis performed on a precise but unsound call graph would miss some paths to vulnerable functions—a costly oversight if severe vulnerabilities are incorrectly reported as unreachable.

6.1.2. Binary Analysis Tool Selection. Given our soundness requirements, we resorted to using the binary analysis tool Sysfilter [41]. Starting with the main executable of a program, Sysfilter analyzes the whole program along with its direct and transitive dependencies, and resolves the origin of all function nodes to their respective binaries, including any dynamically-linked shared libraries. This differentiates functions native to the binary from those imported from external modules.

Soundness for indirect call target resolution. We considered and evaluated several popular binary analysis tools before settling on Sysfilter. Ghidra [77] does not support indirect call target resolution and would therefore report many address-taken functions as "uncalled," an issue also documented by Pang et al. [34]. Angr faces similar issues with indirect call target resolution [34], [78], and its documentation states that its indirect jump resolvers cannot be used for indirect call target resolution [79]. Research prototypes such as BPA [78] support only 32-bit binaries; TypeArmor [74] requires recompilation of all binaries using the LLVM compiler; and BinPointer [80] scales poorly in terms of runtime and memory overhead even on the SPEC 2016 binaries.

Sysfilter, on the other hand, identifies all *address-taken* functions in a binary and, for each indirect call site, considers *all* address-taken functions as potential targets. This approach is *sound* but conservative, leading to overestimation of a function's code footprint. For vulnerability reachability analysis, soundness is preferred over precision: missing indirect call edges that are actually feasible could cause critical vulnerabilities in dependencies to be overlooked. In practice, our analysis in Section 7.4 suggests that this overapproximation does not significantly affect the precision of vulnerability triage. In any case, our results represent a lower bound, and can only be improved if more precise call graph extraction becomes available.

6.1.3. FCG Extraction. After installing a `deb` package and all its dependencies, we run Sysfilter on all installed binaries to obtain their FCG. For main executables, Sysfilter identifies all entry points automatically. For shared libraries, we first use `readelf` [81] to extract the list of exported functions, which is then passed to Sysfilter as entry points for FCG generation. To analyze the 6,632 packages of our dataset, we split the list of `deb` files into 48 batches, and execute this stage of the pipeline in parallel.

6.2. Creating The Knowledge Graph

The extracted FCGs are aggregated into an ArangoDB graph database [82]. This aggregation captures function-level dependencies across the supply chain, resulting in a *knowledge graph* that enables graph traversal queries for assessing the reachability of library functions across thousands of binaries that depend on them. Each function node uses a primary key in the format `name@elf_name@deb_name` to distinguish between functions with the same name in different binaries and `deb` packages.

The next step is to annotate the vulnerable function nodes based on the <*CVE ID*, *vulnerable function name*> tuples generated by the CVE localization pipeline (Figure 3). Each function name is looked up in the knowledge graph, and upon a successful match, we add a node for the CVE and create a directed edge from the CVE to each vulnerable function. If no match is found for a function, we skip it and proceed to the next tuple.

6.3. CVE Reachability Analysis

In the function-level knowledge graph, all nodes adjacent to a CVE node correspond to vulnerable functions. To determine the reachability of a given vulnerability, we use common graph traversal techniques to identify paths leading to vulnerable functions.

Specifically, to find all the binaries that are affected by a given CVE, we designate each vulnerable function as a *start node*, and run an INBOUND breadth-first search (a built-in operation provided by ArangoDB [83]), which recursively follows all incoming edges to a start node. Conversely, to find all CVEs affecting a particular binary, we perform an OUTBOUND breadth-first search operation by designating each function in the binary as a *start node*. We then calculate the total number of unique CVEs corresponding to the vulnerable functions that were reached across all traversals. These two operations compute the *CVE Impact* and *CVE Exposure* metrics which we introduce in Section 7.1 as part of our experimental evaluation.

7. Experimental Evaluation

CVE Dataset. The Debian Security Tracker [50] tracks over 36,000 CVEs across 3,581 `deb` sources. For our study, we focus only on the CVEs reported from 2022 onwards for `deb` sources that ship *at least one library package*. This is because the goal of our reachability analysis is to identify vulnerable functions in shared libraries that are actually used as dependencies, rather than focusing on main executables. While it is possible for a vulnerability in a main executable to affect another executable that depends on its output or execution behavior, such interactions cannot be captured by a function call graph and are therefore beyond the scope of our analysis. With this constraint, we started with an initial pool of 2,180 CVEs reported for 338 `deb` sources.

Unfortunately, the lack of valid links to Git commits in CVE records leaves out many CVEs. Going back to stage (ii) in Figure 4, out of the 2,180 CVEs in our initial set, we had to disregard 1,116 (51.2%) for which we could not obtain any link to a patch, leaving us with 1,064 CVEs having at least one publicly available patch. Without a direct link to the patches that fix a given CVE, we cannot localize the CVE to the corresponding vulnerable function(s).

After CVE localization is complete, we obtain a list of vulnerable functions extracted from the patches of each CVE in a given `deb` source package. However, as discussed in Section 3, each source package often compiles into multiple binary (`.deb`) packages. Consequently, the data present in Debian CVE records is insufficient to determine exactly which ELF64 binaries (shipped by a given `deb` package) contain the vulnerable code. We bridge this gap by querying the list of vulnerable function names in VPChecker's knowledge graph. This step further eliminates 554 CVEs: 63 CVEs where the vulnerable functions affected main executables instead of libraries, and 491 CVEs where the patched functions were not present in our knowledge graph.

Manual inspection of patches for which we could not find a corresponding function in our graph revealed that many of these patches affected the testing infrastructure of a package, and some were not security issues at all. Ultimately, our final pool of vulnerabilities in shared libraries consists of 510 CVEs, affecting 1,541 functions across 150 shared libraries. About half of the CVEs require patching of more than one function. Detailed statistics are provided in Appendix A.

ELF64 Binaries. We collected dynamically linked ELF64 main executables and shared libraries from the Debian Sid repositories. Sid is the official Debian development distribution, and hence all its packages are built from the latest upstream version of their respective source code [36]. Sid's repositories provide debug symbols [73] for *most* packages which aid static binary analysis tools in extracting symbolic information, producing call graphs with accurate function names. These factors led us to choose Debian Sid for our study.

The Sid repository contains over 70,000 Debian packages. Since we aim to measure the impact of vulnerable third-party shared libraries on the Debian supply chain, we focused on binaries from packages that depend on at least one vulnerable library. We found 6,632 such dependent `deb` packages, and successfully extracted the FCGs from a total of 24,076 ELF64 binaries (8,249 shared libraries and 15,827 main executables). Note that we restrict our study to only *dynamically-linked ELF64 executables* compiled from C/C++ code, ignoring all other binary file formats.

7.1. Metrics

Baseline: For our baseline, we consider SBOM tools that identify dynamically linked dependencies at the ELF binary level. We construct an ELF-level knowledge graph (Section 5.1.2) that captures the dynamically linked dependencies of all 24,076 binaries, allowing us to model the core functionality of ELF-level SBOM generation tools [84] for our comparisons.

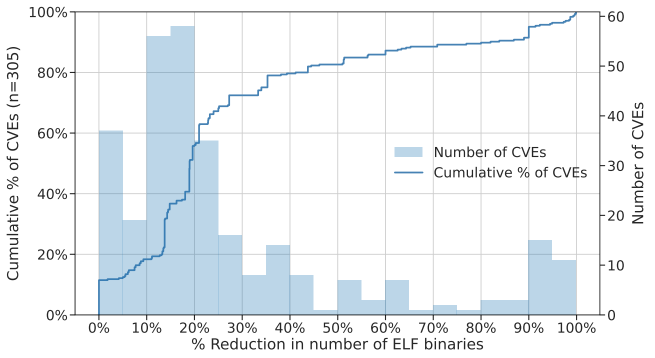

Figure 5: CDF of the percentage reduction in the number of ELF binaries reported as affected by a given CVE when using function-level information, as a percentage of all CVEs in our dataset. For 62 CVEs (20%), *40% or more of the binaries are falsely deemed vulnerable* by ELF-level SBOMs.

CVE Impact: *Given a CVE, the number of ELF binaries that have at least one path reaching any of the vulnerable function nodes affected by the CVE.*

This metric is calculated per CVE over both knowledge graphs as follows. At the ELF level, if an ELF binary includes in its dependency chain a library that is affected by the given CVE, then it is counted. At the function level, an ELF binary is counted if it contains at least one function that has a path to one of the vulnerable functions affected by the CVE.

CVE Exposure: *Given an ELF binary, the number of CVEs that have at least one vulnerable function node lying on a path starting from the target binary.*

This metric is calculated per ELF binary over both knowledge graphs as follows. At the ELF level, every CVE associated with a direct or transitive dependency of the target binary is counted. At the function level, every CVE associated with at least one vulnerable function that is *reachable* from the target binary is counted. To quantitatively measure the difference in these two metrics between the ELF and function levels, we calculate the *percentage reduction* in CVE Impact and CVE Exposure obtained by function-level reachability information when compared to ELF-level information.

7.2. CVE Impact Analysis

We measure the number of ELF binaries reported as affected by a given CVE using both ELF-level and function-level reachability information, and report the reduction in the number of false positives, i.e., the number of binaries unnecessarily deemed vulnerable by ELF-level analysis. To identify actual third-party shared libraries (and distinguish them from application-specific shared libraries that are not used by other packages), we measure CVE Impact for only those libraries on which at least 10 different ELF binaries

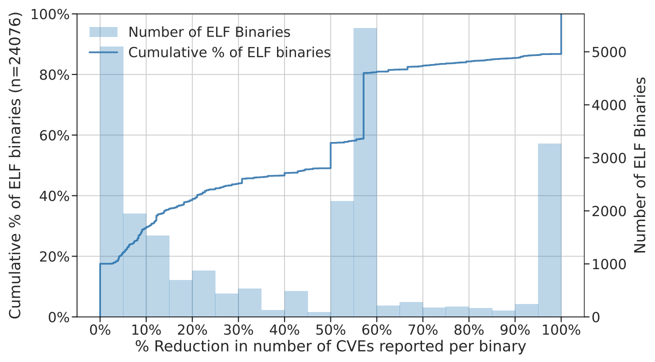

Figure 6: CDF of the percentage reduction in the number of CVEs reported as affecting a given ELF binary when using function-level information, as a percentage of all ELF binaries in our dataset. For about half of the binaries, legacy ELF-level SBOMs falsely report *twice or more* CVEs as relevant, while in reality they are harmless.

depend (we chose this threshold based on our observation that most application-specific libraries are required by fewer than 10 binaries). Figure 5 shows the percentage reduction in the CVE Impact score at the function level, as a cumulative percentage of all CVEs reported for third-party shared libraries.

For about 50% of the CVEs, the reduction in the number of vulnerable binaries is 20% or higher, i.e., 20% or more of the binaries are unnecessarily deemed vulnerable by ELF-level SBOM tools. For 35 CVEs (11%), there is no reduction. Overall, we observe an average of 28% reduction in CVE Impact scores for the CVEs in our dataset.

In practice, the CVE Impact metric is a useful indicator for identifying and prioritizing the remediation of critical vulnerabilities that affect the largest number of software packages deployed in an organization. Note that percentage reduction will never reach 100% because a shared library affected by a CVE will always be reported as vulnerable by both function-level and ELF-level analysis. The actual number of affected binaries varies significantly depending on the location of the vulnerable code and the popularity of the vulnerable shared library. We provide detailed data on the top-50 CVEs with the highest ELF-level CVE Impact in Table 3 in the appendix.

7.3. CVE Exposure Analysis

For each ELF binary in our dataset, we measure how many CVEs are reachable according to the ELF-level and function-level knowledge graphs, and plot the percentage reduction in Figure 6. Out of 24,076 binaries, 12,265 (50.9%) have a reduction of *50% or higher* in the number of CVEs that are actually reachable, i.e., ELF-level reachability analysis falsely reports *twice or more* CVEs as relevant compared to function-level analysis.

For 4,222 binaries (17.5%), function-level information offers no reduction in the number of reachable vulnerabilities. For 2,012 binaries (8.3%), there is exactly a 50 percent reduction (vertical line in the middle of the plot). For 5,221 binaries, the only vulnerable library in their dependency tree is `libc.so.6`. VPChecker reports that out of seven CVEs in `libc`, only three are reachable from these binaries at a function level, bringing their percentage reduction to 57%. For 3,189 binaries (13%), there is a 100% reduction in the number of reachable CVEs.

To summarize, the average reduction in CVE Exposure for 24,076 binaries is 30%, and for 82% of the binaries, we observe that at least one CVE is unreachable. These results suggest that VPChecker offers quite a significant improvement in the accuracy of vulnerability triage for individual programs, reducing considerably the number of CVEs that should be prioritized for remediation. Despite i) the extensive overapproximation of our indirect call target resolution analysis, and ii) the limited set of CVEs for which we could localize their vulnerable functions, these results suggest that VPChecker still significantly reduces the number of false positives reported by ELF-level analysis, which is particularly important for large organizations that need to triage thousands of different deployed binaries and CVEs.

7.4. VPChecker Accuracy

As discussed in Section 6.1.1, extracting a call graph with precise indirect call target resolution from compiled C/C++ binaries is a challenging task. VPChecker uses Sysfilter [41] for static binary analysis, which overapproximates the set of address-taken functions reachable from an indirect call site. This overapproximation affects the vulnerability assertions made by VPChecker, i.e., VPChecker may report a CVE as reachable while it is actually not. To evaluate the extent of this discrepancy, we performed some additional experiments with a subset of binaries for which we extract a more precise call graph at the compiler level.

Compiler-level call graph. To extract more precise call graphs, several works first recompile programs to obtain their intermediate representation (IR), and then match indirect call sites with potential targets by extracting function signatures at both ends using various LLVM passes [75], [85], [86], [87]. We use the LLVM pass proposed by Lin et al. [75], who evaluate their technique on over 1,300 C/C++ binaries. The resulting *compiler-level call graph* serves as our ground truth. The standard Debian compilation process uses a GCC-based toolchain [88]. However, constructing our compiler-level call graph requires recompiling binaries using the LLVM toolchain. Recompiling thousands of packages from the Debian archive is not only computationally intensive [63], but also requires extensive manual intervention to resolve build issues introduced by the different tools—making this effort infeasible at scale. To keep the effort manageable, we perform our compiler-level analysis on the nine libraries with the highest number of reported CVEs in our dataset.

7.4.1. False Positives.

We consider it a false positive when VPChecker reports a vulnerable function as reachable, but the same function is actually *unreachable* in a more precise

TABLE 1: Impact of call-graph soundness and precision on CVE reachability within the nine most vulnerable shared libraries. Sysfilter produces a sound but over-approximated call graph. LLVM IR generates a more precise graph through indirect callsite–callee signature matching. Angr's graph is unsound due to its inability to resolve indirect calls. Replacing Sysfilter with LLVM IR does not reduce reachable CVEs but does lower the number of vulnerable exported functions. In contrast, Angr's unsound graph misses both vulnerable exported functions and reachable CVEs due to missing indirect-call edges.

			Sysfilter			LLVM			Angr	
Library	Total Functions	Exported	Vuln. Exported	CVEs Reached	AICT	Vuln. Exported	CVEs Reached	AICT	Vuln. Exported	CVEs Reached
libexpat.so.1	294	69	19	14	2.12	17	14	0.48	7	11
libXpm.so.4	97	34	16	6	0.82	16	6	0.36	16	6
libtiff.so.6	699	193	107	21	2.32	80	21	1.0	44	14
libcurl.so.4	2811	88	69	28	0.95	68	28	0.44	16	19
libr_core.so.5.9.2	2424	759	295	7	4.06	191	7	3.89	136	4
libgnutls.so.30	2820	1312	931	6	0.54	910	6	0.19	71	4
libxml2.so.2	2171	1398	915	9	0.47	907	9	0.15	517	9
libnetsnmpmibs.so.40	2050	1717	135	6	4.49	132	6	2.59	7	5
libcrypto.so.3	9422	4003	2675	25	5.17	2661	25	2.55	2367	21

version of the call graph (e.g., extracted at the compiler level). It is important to note that the use of a more precise call graph does not affect the reachability of all vulnerable functions. For instance, out of 1,541 vulnerable functions identified by our CVE Localization pipeline (Section 7 CVE Dataset), 556 are exported and hence *directly called* by dependents, leaving no ambiguity about their reachability from direct dependents. The remaining 985 are internal functions of the 150 shared libraries, out of which 815 internal functions are reported as reachable from any indirect call site in their respective shared library. Consequently, these are the only functions (52.9%) that would benefit from a more precise call graph, i.e., for the remaining 47.1% of the vulnerable functions, the conservative analysis of VPChecker cannot result in false positives.

Impact of call graph precision on false positives. Intuitively, the compiler-level call graph may have fewer entry points leading to a vulnerable function when compared to its overapproximated binary-level counterpart. To assess the impact of this overapproximation, we measured the number of reachable CVEs when using the compiler-level and binary-level call graphs. We expect that the more accurate compiler-level call graph will result in fewer CVEs being reachable from a library's entry points (i.e., all its exported functions). The results of our analysis are presented in Table 1.

A common metric for estimating the overapproximation of indirect call target resolution is the *Average Indirect Call Targets (AICT)* score [80], [78], [76]. For a given binary, the AICT score is calculated as the ratio of the total number of potential indirect call targets to the total number of indirect call sites. As shown in the "AICT" columns for Sysfilter and LLVM in Table 1, the call graph extracted at the compiler level is clearly more accurate, as it has a lower AICT score for all nine binaries. Strikingly, this increased precision *does not lead* to any reduction in the number of reachable CVEs, as evident by the "CVEs reached" columns. However, we do observe a slight reduction in the number of exported functions that can reach a vulnerable function within the same library. This reduction stems from replacing Sysfilter's over-approximation of indirect-call edges, where instead of assuming every indirect call site can target every address-taken function, targets are restricted to address-taken functions whose signatures match the call site.

Considering the example of `libcrypto.so.3`, from the results of Table 1 we observe that switching to the compiler-level call graph reduces the number of vulnerable exported functions (the subset of exported functions that have at least one path to a function containing a CVE) by just 14 (from 2,675 to 2,661). Further inspection of these 14 functions reveals that 13 of them are *never called* by any of the 24,075 binaries in our dataset, and only one of them is called by three binaries. This implies that for `libcrypto.so.3`, switching to the more precise compiler-level call graph would reduce the number of reported CVEs for just three out of the 24,076 programs. We observe similar results for the other eight binaries, and report our detailed findings in Appendix B. Consequently, for the 116 CVEs across these 9 shared libraries, switching to a more precise call graph will not offer any significant advantage for vulnerability triage, which means that VPChecker's less precise binary-level operation results in a negligible number of falsely reported CVEs—always much lower compared to existing ELF-level tools, as discussed in Sections 7.2 and 7.3.

7.4.2. False Negatives. We consider as a false negative any case in which VPChecker reports a vulnerable function as not reachable, if the same function is actually reachable in the compiler-level call graph. We did not observe any such case in our experiments, because the call graph generated by Sysfilter is *complete* [41], i.e., it never excludes any function that may be executed by the program, under any possible input. This includes asynchronous programming, signal handlers, and other callback functions, all of which are address-taken and will thus be included in the (overapproximated) call graph. We thus claim that the output of VPChecker will not result in any false negatives for a given binary and all its dynamically linked dependencies.

The only possibility for false negatives in our current implementation would stem from programs that load dynamic shared objects using dlopen, which is typically used for loading plugins or external modules at runtime. Given a binary, statically determining the particular shared libraries it may load using dlopen remains a challenge. To avoid such false negatives, VPChecker falls back to package-level analysis for binaries that contain a call to dlopen.

The corresponding Debian package of a given program explicitly declares all its library dependencies, including those loaded using dlopen. If the program uses dlopen to load a shared library that is not dynamically linked, it will still be included in the package (e.g., Apache's loadable modules). When VPChecker statically identifies the presence of dlopen in a binary, it reports all CVEs present in the *non-dynamically linked* shared libraries included in the binary's Debian package, thereby performing no worse than existing package-level vulnerability scanners. In our dataset, we found only 1,153 (4% of all) binaries making a call to dlopen.

7.4.3. Using Unsound Call Graph. To evaluate the impact of using an unsound call graph for VPChecker, we replaced Sysfilter—the underlying call-graph tool—with Angr [89], a popular binary analysis framework. We used Angr's static call graph extraction algorithm CFGFast and set the option resolve_indirect_jumps as *true* [90]. Note that although Angr supports indirect jump resolution, the resolver does not support indirect call target resolution [79], hence the graph it produces is unsound due to missing edges corresponding to potential indirect function calls. We run Angr on the same nine most vulnerable libraries and compare the results with the ground truth call graph obtained from LLVM bitcode.

We show our results in Table 1 (two right-most columns). Note that we leave out AICT for Angr since CFGFast does not give us any approximation of indirect call targets. For seven out of nine libraries, Angr falsely reports many CVEs as unreachable. For example, in libcrypto, Angr fails to report four CVEs as reachable. One of the missed CVEs is CVE-2024-4603 [91], which required patching the function ossl_dsa_check_priv_key, which in turn is directly called by dsa_validate [92]. However, dsa_validate is an address-taken function that appears in the function pointer array ossl_dsa_keymgmt_functions [92]. Because Angr does not resolve indirect calls, it misses the edge to dsa_validate, and CVE-2024-4603 is falsely reported as unreachable.

8. Discussions and Limitations

Call Graphs for C/C++ Binary SBOMs. Our results demonstrate that even a highly conservative function call graph generator for C/C++ binaries can serve as an effective starting point for identifying unreachable CVEs (*false positives*) in the dependencies of many binaries. We recommend that package maintainers generate SBOMs for each binary they distribute and embed the corresponding FCG as SBOM metadata, since FCGs can be generated more precisely at the source or IR level. At the same time, we show that in practice—particularly for widely used and complex libraries—a conservative call graph generated at the binary level using an off-the-shelf static analysis tool can be nearly as effective for vulnerability triage (Section 7.4). Therefore, system administrators can immediately begin leveraging FCG-based analysis to monitor their applications' code reachability to vulnerable functions in third-party dependencies.

Call Graph Precision. The primary challenge in increasing the accuracy of our code reachability analysis lies in improving the precision of binary-level indirect call target resolution. While tools such as TypeArmor [74], TypeSqueezer [76], BPA [78], and BinPointer [80] incorporate advanced heuristics for improving AICT numbers in COTS binaries, their evaluations have primarily been limited to the SPEC binaries. In our future work, we plan to assess the suitability of these techniques for large-scale, ecosystem-wide binary analysis—which is the focus of our study.

A limitation stemming from the difficulty of resolving indirect call targets is that our knowledge graph does not account for cross-module (cross-binary) indirect function calls. The FCG generated by Sysfilter enumerates all address-taken functions within the scope of the program analysis. However, it is possible that some address-taken functions may be potential targets of cross-module indirect calls. While we conservatively assume that address-taken functions may only be targets of indirect calls originating from call sites within the same binary, it remains difficult to determine which external binary in the analysis scope (if any) may have taken a function's address. Nevertheless, such cases are rare in our dataset—functions that could potentially serve as targets of cross-module indirect calls comprise only 0.3% of all functions in our graph. In future iterations, a reasonable first approximation could be to propagate reachability only to those binaries within the scope of the program analysis that directly import symbols from the binary containing the address-taken function under consideration.

CVE Localization. Our CVE localization pipeline treats the list of patches linked to a CVE record as the ground truth for identifying the precise set of vulnerable functions. However, in practice, not all functions modified in a patch are actually vulnerable, which can cause the pipeline to overestimate vulnerable functions in some cases. For instance, GitHub/GitLab allow multiple commits in a single pull/merge request. Maintainers sometimes fix several CVEs in one request, which is linked into the record of each patched CVE. Consider five libexpat CVEs (CVE-2022-22822 through CVE-2022-22827), caused by flaws in five different functions, all fixed in a single pull request [93]. When our pipeline processes these five CVE records, we encounter the same patch that fixes five functions for each CVE, creating an impression that each CVE was caused by all five functions. This yields false positives for dependents that call only one vulnerable function and realistically are at risk from only one CVE, but instead get flagged for four additional CVEs.

This depends on maintainer practices, and is uncommon in our dataset: across all Git references, we observe only 47 pull requests linked as fixes to 61 CVEs, out of more than 500 CVEs and thousands of commits. Additionally, patches modifying inline functions or structure definitions introduce identifiers that do not appear as symbols in the final binary because they are inlined during compilation.

9. Related Work

Function-level Reachability. The Java and JavaScript ecosystems have witnessed extensive work showcasing the effectiveness of using function-level reachability information to reduce false positives originating from unreachable vulnerabilities. Zapata et al. [30] *manually inspected* over 60 `npm` projects and found that 73% of them were safe from vulnerable dependencies because the vulnerable function was *never called*. Nielsen et al. [94] used function call graphs for `npm` applications and found that security warnings could be reduced by up to 81% when reporting vulnerabilities using a function-level dependency tree.

Plate et al. [32] and Ponta et al. [95] used dynamic and static analysis to determine the reachability of vulnerable code in Java-based projects. Wu et al. [96] published an empirical study measuring the reachability of CVEs in the Maven ecosystem. Building on this work, Zhang et al. introduced VAScanner [35] to improve the process of extracting vulnerable function names from CVE patches, and conducted another reachability study that demonstrated the effectiveness of function-level vulnerability reachability analysis in the Maven ecosystem.

Our work is the first to apply large-scale reachability analysis to the C/C++ ecosystem. Despite the inherent challenges in constructing precise FCGs, we demonstrate that conservative overapproximations yield results that are precise enough to motivate a shift toward function-level vulnerability triaging for C/C++ binaries. The closest related effort in a compiled language ecosystem was conducted in Rust. Präzi [97], is a framework that built function-level knowledge graph for the Rust supply chain, relying on the recompilation of all Rust packages and extracting FCGs from their LLVM IR. However, Präzi was not used to perform any CVE reachability analysis.

SBOMs for Compiled Binaries. Since compiled binaries lack much of the information that is crucial for dependency identification, a few studies have attempted to embed additional metadata in a separate section of the file [98], [99], [100]. Automatic Bill of Materials (ABOM) [100] rethinks binary-level SBOM construction by computing the hashes of source code files and embedding them into a compressed data structure that is included as a separate section in the binary. However, the mapping from the binary files to source files remains too coarse for a function-level study.

Vulnerability Scanning Tools for C/C++ Binaries. There are several open-source [101], [102] and proprietary [103], [104] vulnerability scanning tools for C/C++ binaries. These tools employ static binary analysis techniques to identify patterns and code signatures in target binaries. For example, `cve-bin-tool` [101] by Intel extracts `ASCII` strings from the target binary, which include the program name and release version. These are then used to look up the corresponding entries in the NVD feed. The `cwe_checker` tool uses Ghidra [77] to search for vulnerable code patterns. However, these tools do not uncover dynamic library dependencies and operate solely on the target binary. Syft [84] reads the dynamic section in ELF header, which contains only the direct dependencies of a binary. Observing a gap in accurately identifying all dependencies of a binary and integrating them with vulnerability information, we attempt to plug this by constructing an ecosystem-wide, function-level knowledge graph annotated with vulnerability data.

CVE Mining and Parsing Patches. Several studies have examined vulnerabilities and their patches on a large scale [68], [67], [66], [105]. Li et al. [66] and V0Finder [68] employ a heuristic approach using the `ctags` utility to extract the names of modified functions from patch files. We adopt a similar approach to identify the functions impacted by a CVE. Fan et al. [105] mined code and vulnerability data associated with 348 C/C++ repositories from Github, and created Big-Vul, a dataset curated by extracting vulnerability-related code changes from repository-specific CVEs that were reported until 2019. Our CVE dataset is more recent (2022–2024) and is specifically curated for the Debian ecosystem - which inherently lacks explicit mention of vulnerable function information and mapping of CVEs to compiled binaries.

10. Conclusion

We show that the *precision* and *actionable usefulness* of vulnerability triage can be significantly improved by taking into account the context in which a vulnerable dependency is used in terms of code reachability. Our results show that code reachability information reduces the number of binaries reported as affected by a given CVE by 28%, while reducing the number of different CVEs reported as affecting a given ELF binary by 30%. These findings underscore the value of fine-grained dependency analysis for improving the precision of vulnerability triage, and provide strong motivation for enhancing existing SBOM formats to support dependencies modeled at the level of function call graphs.

As part of our future work, we plan to incorporate more precise indirect call target identification approaches, and improve our CVE localization process by exploring more comprehensive textual analysis techniques for the description fields of existing CVE records to identify and extract the relevant patches.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-24-1-2054.

References

[1] F. Li, L. Rogers, A. Mathur, N. Malkin, and M. Chetty, "Keepers of the machines: Examining how system administrators manage software updates for multiple machines," in *Proceedings of the USENIX Symposium on Usable Privacy & Security (SOUPS)*, 2019.

[2] K. Vaniea, E. J. Rader, and R. Wash, "Betrayed by updates: How negative experiences affect future security," in *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, 2014.

[3] O. Bergman and S. Whittaker, "The cognitive costs of upgrades," *Interactive Computing*, vol. 30, pp. 46–52, 2018.

[4] E. M. Redmiles, M. L. Mazurek, and J. P. Dickerson, "Dancing pigs or externalities? measuring the rationality of security decisions," in *Proceedings of the ACM Conference on Economics and Computation (EC)*, 2018.

[5] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proceedings of the USENIX Security Symposium*, 2018.

[6] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.

[7] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *Proceedings of the USENIX Security Symposium*, 2019.

[8] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[9] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[10] Anchore, "Grype - Vulnerability Scanner," 2024. [Online]. Available: https://github.com/anchore/grype

[11] "Bomber," https://github.com/devops-kung-fu/bomber, 2024.

[12] "Dependency Track," OWASP, 2025. [Online]. Available: https://github.com/DependencyTrack/dependency-track

[13] E. J. O'Donoghue, "Using software bill of materials for software supply chain security and its generation impact on vulnerability detection," https://www.cs.montana.edu/izurieta/thesis/ODonoghue.pdf, 2024.

[14] S. Berkovich, J. Kam, and G. Wurster, "UBCIS: Ultimate benchmark for container image scanning," in *Proceedings of the 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/cset20/presentation/berkovich

[15] "Executive Order on Improving the Nation's Cybersecurity," 2024. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[16] "SBOM at a Glance," National Telecommunications and Information Administration, 2021. [Online]. Available: https://www.ntia.gov/sites/default/files/publications/sbom_at_a_glance_apr2021_0.pdf

[17] "SBOM Use Cases and Benefits," National Telecommunications and Information Administration, 2021. [Online]. Available: https://www.ntia.gov/sites/default/files/publications/ntia_sbom_use_cases_roles_benefits-nov2019_0.pdf

[18] R. A. Martin, "Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things," in *Proceedings of the IEEE Systems Security Symposium (SSS)*, 2020.

[19] "Snyk CLI," Snyk, 2025. [Online]. Available: https://github.com/snyk/cli

[20] "Vulnerability Exploitability eXchange (VEX) - Minumum Requirements," CISA, 2023. [Online]. Available: https://www.cisa.gov/sites/default/files/2023-04/minimum-requirements-for-vex-508c.pdf

[21] Y. Na, S. Woo, J. Lee, and H. Lee, "CNEPS: a precise approach for examining dependencies among third-party C/C++ open-source components," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1020–1020.

[22] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in C/C++ ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[23] "Executable and Linkable Format," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[24] "File list of package libssl3 in sid of architecture amd64," 2024. [Online]. Available: https://packages.debian.org/sid/amd64/libssl3/filelist

[25] "OSV Scalibr," Google, 2025. [Online]. Available: https://github.com/google/osv-scalibr/tree/main

[26] "Trivy," Aqua Security, 2025. [Online]. Available: https://github.com/aquasecurity/trivy

[27] A. A. Younis, Y. K. Malaiya, and I. Ray, "Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability," in *Proceedings of the IEEE 15th International Symposium on High-Assurance Systems Engineering*, 2014.

[28] S. Tariq, M. Baruwal Chhetri, S. Nepal, and C. Paris, "Alert fatigue in security operations centres: Research challenges and opportunities," *ACM Comput. Surv.*, Mar. 2025. [Online]. Available: https://doi.org/10.1145/3723158

[29] B. A. Alahmadi, L. Axon, and I. Martinovic, "99% false positives: A qualitative study of SOC analysts' perspectives on security alarms," in *Proceedings of the 31st USENIX Security Symposium*, 2022, pp. 2783–2800.

[30] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 559–563.

[31] Y. Zhao, Y. Zhang, D. Chacko, and J. Cappos, "CovSBOM: Enhancing software bill of materials with integrated code coverage analysis," in *Proceedings of the IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, 2024.

[32] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 411–420.

[33] D. Engel, F. Verbeek, and B. Ravindran, "On the decidability of disassembling binaries," in *Proceedings of the 18th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2024.

[34] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021, pp. 833–851.

[35] F. Zhang, L. Fan, S. Chen, M. Cai, S. Xu, and L. Zhao, "Does the vulnerability threaten our projects? Automated vulnerable API detection for third-party libraries," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2906–2920, 2024.

[36] "Debian Unstable Distribution," 2024. [Online]. Available: https://wiki.debian.org/DebianUnstable

[37] "Syft ELF Cataloger," Anchore, 2025. [Online]. Available: https://github.com/anchore/syft/blob/da62a82413e86017215d1612a70a23183d4469d3/syft/file/cataloger/executable/elf.go

[38] C. Mulliner and M. Neugschwandtner, “Breaking Payloads with Runtime Code Stripping and Image Freezing,” 2015, black Hat USA.

[39] C. Porter, G. Mururu, P. Barua, and S. Pande, “BlankIt library debloating: Getting what you want instead of cutting what you don’t,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[40] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, “Temporal system call specialization for attack surface reduction,” in *Proceedings of the USENIX Security Symposium*, 2020.

[41] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, “Sysfilter: Automated system call filtering for commodity software,” in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[42] C. Canella, M. Werner, D. Gruss, and M. Schwarz, “Automating Seccomp filter generation for Linux applications,” in *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2021, pp. 139–151.

[43] S. Mishra and M. Polychronakis, “Shredder: Breaking exploits through API specialization,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[44] ——, “Saffire: Context-sensitive function specialization against code reuse attacks,” in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[45] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, “Slimium: Debloating the Chromium browser with feature subsetting,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[46] H. Koo, S. Ghavamnia, and M. Polychronakis, “Configuration-driven software debloating,” in *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2019.

[47] S. Ghavamnia, T. Palit, and M. Polychronakis, “C2C: Fine-grained configuration-driven system call filtering,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[48] “CVE Records,” 2024. [Online]. Available: https://cve.mitre.org/cve/identifiers/index

[49] “CVE Program,” 2024. [Online]. Available: https://www.cve.org/

[50] “Debian Security Tracker,” 2024. [Online]. Available: https://security-tracker.debian.org/tracker/data/json

[51] “Basics of the Debian package management system,” 2024. [Online]. Available: https://www.debian.org/doc/manuals/debian-faq/pkg-basics.en.html

[52] “Debian Package: ffmpeg (7:6.1.1-4 and others),” 2024. [Online]. Available: https://packages.debian.org/sid/ffmpeg

[53] “debsecan - Debian Security Analyzer,” 2024. [Online]. Available: https://manpages.debian.org/testing/debsecan/debsecan.1.en.html

[54] “CVE Record Information Requirements,” 2020. [Online]. Available: https://www.cve.org/ResourcesSupport/AllResources/CNARules#section_8_cve_record_requirements

[55] “CVE JSON Record Format is 5.1.0,” 2024. [Online]. Available: https://github.com/CVEProject/cve-schema/tree/main

[56] “MindMap: CVE JSON Record Format is 5.1.0,” 2024. [Online]. Available: https://cveproject.github.io/cve-schema/schema/docs/mindmap.html

[57] “CVE List V5,” 2024. [Online]. Available: https://github.com/CVEProject/cvelistV5/tree/cve_2024-06-23_0200Z

[58] “Open Source Vulnerability format,” https://ossf.github.io/osv-schema/, 2024.

[59] M. Malone, Y. Wang, K. Z. Snow, and F. Monrose, “Applicable micropatches and where to find them: Finding and applying new security hot fixes to old software,” in *Proceedings of the IEEE Conference on Software Testing, Verification, and Validation (ICST)*, 2021.

[60] “SBOM in Action: finding vulnerabilities with a Software Bill of Materials,” https://security.googleblog.com/2022/06/sbom-in-action-finding-vulnerabilities.html, 2022.

[61] B. Rolf, N. Mebarki, S. Lang, T. Reggelin, O. Cardin, H. Mouchère, and A. Dolgui, “Using knowledge graphs and human-centric artificial intelligence for reconfigurable supply chains: A research framework,” in *Proceedings of the 10th IFAC Conference on Manufacturing Modelling, Management and Control (MIM)*, 2022, pp. 1693–1698.

[62] H. Hoogendorp, “Extraction and visual exploration of call graphs for large software systems,” Ph.D. dissertation, University of Groningen, 2010.

[63] “Debian Package Rebuild,” 2025. [Online]. Available: https://clang.debian.net/

[64] “An overview of build systems (mostly for C++ projects),” 2018. [Online]. Available: https://web.archive.org/web/20231019163807/https://julienjorge.medium.com/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444

[65] “ldd,” 2023. [Online]. Available: https://man7.org/linux/man-pages/man1/ldd.1.html

[66] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, p. 2201–2215.

[67] S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: a scalable approach for vulnerable code clone discovery,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 595–614.

[68] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, “V0Finder: Discovering the correct origin of publicly reported software vulnerabilities,” in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 3041–3058.

[69] “Bugzilla Project,” 2024. [Online]. Available: https://www.bugzilla.org/

[70] “Diff hunks,” 2024. [Online]. Available: https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html

[71] “Universal ctags,” 2023. [Online]. Available: https://github.com/universal-ctags/ctags

[72] “Debian Sid,” 2024. [Online]. Available: https://hub.docker.com/layers/library/debian/sid/images/sha256-6fa6d6dfe6ad51b53f5241965474da3fdf4cd5927cb27f29b599c66eeecc2256

[73] “Debug Symbols,” 2022. [Online]. Available: https://wiki.debian.org/UsingSymbolsFiles

[74] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[75] Y. Lin and D. Gao, “When function signature recovery meets compiler optimization,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[76] Z. Lin, J. Li, B. Li, H. Ma, D. Gao, and J. Ma, “Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[77] “Ghidra.” [Online]. Available: https://ghidra-sre.org/

[78] S. H. Kim, C. Sun, D. Zeng, and G. Tan, “Refining indirect call targets at the binary level,” *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[79] “Param jumptable_resolver_resolves_calls in Angr CFGFast API,” 2025. [Online]. Available: https://docs.angr.io/en/latest/api.html

[80] S. H. Kim, D. Zeng, C. Sun, and G. Tan, “BinPointer: towards precise, sound, and scalable binary-level pointer analysis,” in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC)*, 2022.

[81] "readelf," 2024. [Online]. Available: https://man7.org/linux/man-pages/man1/readelf.1.html

[82] "ArangoDB." [Online]. Available: https://arangodb.com/community-server/

[83] "ArangoDB Graph Traversals." [Online]. Available: https://docs.arangodb.com/3.12/aql/graphs/traversals/

[84] "Syft," Anchore, 2025. [Online]. Available: https://github.com/anchore/syft

[85] D. Liu, S. Ji, K. Lu, and Q. He, "Improving indirect-call analysis in LLVM with type and data-flow co-analysis," in *Proceedings of the 33rd USENIX Security Symposium*, 2024, pp. 5895–5912.

[86] T. Xia, H. Hu, and D. Wu, "DEEPTYPE: Refining indirect call targets with strong multi-layer type analysis," in *Proceedings of the 33rd USENIX Security Symposium*, 2024, pp. 5877–5894.

[87] K. Lu and H. Hu, "Where does it go? Refining indirect-call targets with multi-layer type analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, p. 1867–1881.

[88] "Debian Toolchain," 2025. [Online]. Available: https://wiki.debian.org/ToolChain

[89] "angr," 2025. [Online]. Available: https://angr.io/

[90] "angr CFGFast," 2025. [Online]. Available: https://docs.angr.io/en/latest/analyses/cfg.html

[91] "CVE-2024-4603 Debian Security Tracker," 2024. [Online]. Available: https://security-tracker.debian.org/tracker/CVE-2024-4603

[92] "DSA Validate Key," 2023. [Online]. Available: https://salsa.debian.org/debian/openssl/-/blob/debian/openssl-3.2.2-1/providers/implementations/keymgmt/dsa_kmgmt.c

[93] "Fix for CVE-2022-22822 to CVE-2022-22827," 2022. [Online]. Available: https://github.com/libexpat/libexpat/pull/539

[94] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of Node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[95] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Softw. Engg.*, vol. 25, no. 5, pp. 3175–3215, Sep. 2020.

[96] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 1046–1058.

[97] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empirical Softw. Engg.*, vol. 27, no. 5, 2022.

[98] "Enabling universal artifact traceability in software supply chains," OmniBor, 2022. [Online]. Available: https://omnibor.io/resources/whitepaper/

[99] "Package information on ELF objects (System-Wide Change proposal)," 2021. [Online]. Available: https://lwn.net/ml/fedora-devel/CA+voJeU--6Wk8j=D=i3+Eu2RrhWJACUiirX2UepMhp0krBM2jg@mail.gmail.com/

[100] N. Boucher and R. Anderson, "Automatic bill of materials," 2023.

[101] cve-bin tool, "CVE Binary Tool," 2024. [Online]. Available: https://github.com/intel/cve-bin-tool

[102] cwe checker, "CWE Checker Tool," 2024. [Online]. Available: https://github.com/fkie-cad/cwe_checker

[103] "Synopsys Application Security," 2024. [Online]. Available: https://www.synopsys.com/software-integrity/software-composition-analysis-tools/binary-analysis.html

[104] "Binary code scanners." [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/binary-code-scanners

[105] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proceedings of the 17th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2020, pp. 508–512.

Appendix

1. Functions modified by CVE Patches

As discussed in Section 7, our final pool of vulnerabilities in shared libraries consists of 510 CVEs, affecting 1,541 functions across 150 shared libraries. As shown in the CDF of Figure 7, about half of the CVEs require patching of more than one function.

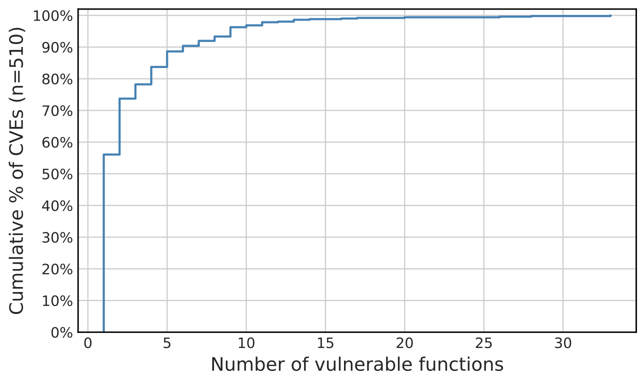

Figure 7: Number of affected (vulnerable) functions per CVE, as identified from the corresponding patches. About half of the CVEs involve flaws in more than one function.

2. Impact of call graph overapproximation

In Table 2, we report the reduction in the number of vulnerable entry points for the nine most vulnerable (containing the highest number of CVEs) shared libraries in our dataset. Eight out of the nine libraries have a reduction in the number of vulnerable entry points (exported functions having a path to a vulnerable function within the same library) when using the more precise LLVM call graph. However, the last column shows that almost all these extra functions are *not imported at all* by any of the 24,076 binaries in our dataset. This indicates that a more precise call graph would not result in any meaningful reduction in the number of reported CVEs for most of the binaries that import at least one function from these nine shared libraries.

3. CVE Impact Scores

Table 3 presents the raw CVE Impact scores measured at the ELF level and at the function level. For the sake of brevity, we list only the top-50 CVEs that have the maximum impact at the ELF level, and we include only one representative vulnerable function per CVE.

TABLE 2: Reduction in the number of vulnerable entry points for the nine libraries with the most CVEs in our dataset when using a compiler-level call graph (G), compared to the overapproximated binary-level call graph (O). As shown in the rightmost column, for 8 out of the 9 libraries, the vast majority of the extra functions included in the binary-level call graph (i.e., the set $O - G$) are not imported by any binary in our dataset.

Library Name	**Vuln. Exported Binary (O)**	**Vuln. Exported LLVM (G)**	**Extra Functions ($O - G$)**	**# of Uncalled Functions in ($O - G$)**
libexpat.so.1	19	17	2	2
libXpm.so.4	16	16	0	-
libtiff.so.6	107	80	27	14
libcurl.so.4	69	68	1	1
libr_core.so.5.9.2	295	191	104	104
libgnutls.so.30	931	910	21	15
libxml2.so.2	915	907	8	8
libnetsnmpmibs.so.40	135	132	3	3
libcrypto.so.3	2675	2661	14	13

TABLE 3: Detailed CVE impact results for the top-50 CVEs with the highest impact at the ELF level.

CVE-ID	**Function**	**ELF Binary**	**Deb Package**	**ELF-level Reach**	**Function-level Reach**
CVE-2022-39046	`__vsyslog_internal`	`libc.so.6`	`libc6_2.38-13`	24076	12135
CVE-2022-37434	`inflate`	`libz.so.1`	`zlib1g_1:1.3.dfsg%2Breally1.3.1-1`	11461	10199
CVE-2022-1587	`jit_compile`	`libpcre2-8.so.0`	`libpcre2-8-0_10.42-4%2Bb1`	5866	2567
CVE-2023-32611	`g_variant_serialise`	`libglib-2.0.so.0`	`libglib2.0-0t64_2.80.3-1`	5049	4672
CVE-2023-2603	`_libcap_strdup`	`libcap.so.2`	`libcap2_1:2.66-5`	4483	32
CVE-2023-43787	`XCreateImage`	`libX11.so.6`	`libx11-6_2:1.8.7-1%2Bb1`	4302	2839
CVE-2022-27404	`sfnt_init_face`	`libfreetype.so.6`	`libfreetype6_2.13.2+dfsg-1+b4`	3768	3260
CVE-2023-0215	`BIO_new_NDEF`	`libcrypto.so.3`	`libssl3t64_3.2.2-1`	3481	4
CVE-2023-2804	`start_pass_dpost`	`libjpeg.so.62`	`libjpeg62-turbo_1:2.1.5-3`	3447	3447
CVE-2022-40674	`internalEntityProcessor`	`libexpat.so.1`	`libexpat1_2.6.2-1`	3379	2716
CVE-2022-42898	`krb5_pac_parse`	`libkrb5.so.3`	`libkrb5-3_1.21.2-1`	2486	11
CVE-2024-26458	`gss_krb5int_make_seal_token_v3`	`libgssapi_krb5.so.2`	`libgssapi-krb5-2_1.21.2-1`	2406	2201
CVE-2024-0567	`_gnutls_sort_clist`	`libgnutls.so.30`	`libgnutls30t64_3.8.5-4`	2146	1828
CVE-2024-2511	`ssl_update_cache`	`libssl.so.3`	`libssl3t64_3.2.2-1`	1853	1387
CVE-2022-45873	`source_disconnect`	`libudev.so.1`	`libudev1_256.1-2`	1842	1345
CVE-2022-44638	`pixman_sample_floor_y`	`libpixman-1.so.0`	`libpixman-1-0_0.42.2-1+b1`	1579	1360
CVE-2024-25062	`xmlTextReaderRead`	`libxml2.so.2`	`libxml2_2.12.7%2Bdfsg-3`	1496	242
CVE-2022-25310	`fribidi_remove_bidi_marks`	`libfribidi.so.0`	`libfribidi0_1.0.13-3%2Bb1`	1438	5
CVE-2024-28182	`nghttp2_session_mem_recv2`	`libnghttp2.so.14`	`libnghttp2-14_1.62.1-1`	1324	1208
CVE-2023-48795	`_libssh2_transport_send`	`libssh2.so.1`	`libssh2-1t64_1.11.0-5`	1236	1057
CVE-2023-38039	`Curl_pp_readresp`	`libcurl-gnutls.so.4`	`libcurl3t64-gnutls_8.8.0-2`	872	752
CVE-2023-1999	`EncodeAlphaInternal`	`libwebp.so.7`	`libwebp7_1.4.0-0.1`	755	602
CVE-2023-29491	`_nc_read_termtype`	`libtinfo.so.6`	`libtinfo6_6.5-2`	729	729
CVE-2023-52356	`TIFFReadRGBAStripExt`	`libtiff.so.6`	`libtiff6_4.5.1%2Bgit230720-4`	663	73
CVE-2022-33065	`mat4_read_header`	`libsndfile.so.1`	`libsndfile1_1.2.2-1+b2`	486	182
CVE-2024-2236	`rsa_decrypt`	`libgcrypt.so.20`	`libgcrypt20_1.10.3-3`	366	282
CVE-2022-47022	`hwloc_linux_set_tid_cpubind`	`libhwloc.so.15`	`libhwloc15_2.11.0-2`	330	330
CVE-2024-25260	`arm_machine_flag_name`	`libdw.so.1`	`libdw1t64_0.191-1+b1`	255	91
CVE-2023-38473	`avahi_alternative_host_name`	`libavahi-common.so.3`	`libavahi-common3_0.8-13%2Bb2`	255	4
CVE-2023-43789	`xpmNextWord`	`libXpm.so.4`	`libxpm4_1:3.5.17-1+b1`	211	162
CVE-2023-22745	`Tss2_RC_Decode`	`libtss2-rc.so.0`	`libtss2-rc0t64_4.1.3-1`	144	144
CVE-2022-36227	`__archive_write_allocate_filter`	`libarchive.so.13`	`libarchive13t64_3.7.2-2.1`	132	124
CVE-2023-41105	`_Py_normpath`	`libpython3.12.so.1.0`	`libpython3.12t64_3.12.4-1`	129	129
CVE-2024-35235	`httpAddrListen`	`libcups.so.2`	`libcups2t64_2.4.10-1`	111	3
CVE-2022-33099	`luaV_concat`	`liblua5.1.so.0`	`liblua5.1-0_5.1.5-9%2Bb2`	93	88
CVE-2024-1013	`SQLStatistics`	`libodbc.so.2`	`libodbc2_2.3.12-1%2Bb2`	81	70
CVE-2023-5217	`vp8_change_config`	`libvpx.so.9`	`libvpx9_1.14.1-1`	80	67
CVE-2023-31147	`ares_destroy`	`libcares.so.2`	`libcares2_1.31.0-1`	79	61
CVE-2023-50471	`cJSON_SetValuestring`	`libcjson.so.1`	`libcjson1_1.7.18-3`	76	1
CVE-2024-31578	`av_hwframe_ctx_init`	`libavutil.so.58`	`libavutil58_7:6.1.1-4%2Bb4`	76	60
CVE-2022-1475	`g729_parse`	`libavcodec.so.60`	`libavcodec60_7:6.1.1-4%2Bb4`	72	59
CVE-2024-28836	`mbedtls_ssl_session_reset_int`	`libmbedtls.so.14`	`libmbedtls14t64_2.28.8-1`	70	54
CVE-2022-1215	`evdev_device_get_sysname`	`libinput.so.10`	`libinput10_1.26.0-1`	63	39
CVE-2022-48468	`parse_required_member`	`libprotobuf-c.so.1`	`libprotobuf-c1_1.4.1-1+b2`	61	37
CVE-2022-28805	`singlevar`	`liblua5.3.so.0`	`liblua5.3-0_5.3.6-2+b2`	61	53
CVE-2023-2283	`pki_verify_data_signature`	`libssh-gcrypt.so.4`	`libssh-gcrypt-4_0.10.6-3`	60	40
CVE-2023-39976	`_blackbox_vlogger`	`libqb.so.100`	`libqb100_2.0.8-2`	55	54
CVE-2022-48554	`file_copystr`	`libmagic.so.1`	`libmagic1t64_1:5.45-3`	55	48
CVE-2022-3341	`nut_read_header`	`libavformat.so.60`	`libavformat60_7:6.1.1-4%2Bb4`	50	38
CVE-2024-24806	`uv__idna_toascii`	`libuv.so.1`	`libuv1t64_1.48.0-5`	50	9