École Polytechnique Fédérale de Lausanne

Deptyque: eBPF-Based Build Provenance for Dependencies

by Guochao Xie

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Christopher Ferreira, Oracle
External Expert

Dr. Hugo Guiroux, Oracle
Adam Zurada, Oracle
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

March 1, 2024

It is only with the heart that one can see rightly;
what is essential is invisible to the eye.
— Antoine de Saint-Exupéry, The Little Prince

Dedicated to my precious friends Tong, Jin, and Yuening.

Acknowledgments

This thesis acknowledgment stands as a tribute to the numerous individuals who have made my academic journey not only possible but also enriching and rewarding. It is with heartfelt gratitude that I extend my thanks to those who have supported me throughout my master's studies and the completion of this thesis.

First and foremost, I wish to express my deepest appreciation to my esteemed supervisors at Oracle Labs and EPFL. Christopher Ferreira and Dr. Hugo Guiroux played pivotal roles in guiding my thesis, enriching it with practical industrial insights and direction. Their unwavering support, coupled with their invaluable feedback, significantly advanced this work forward. I also owe a great deal of gratitude to Prof. Dr. sc. Mathias Payer for his indispensable comments and continuous encouragement throughout this endeavor.

I am also immensely grateful to my manager, Adam Zurada, and my esteemed colleagues within the Oracle Autonomous Middleware team at the Prime Tower in Zurich. Yuening Yang, Bence Czipo, Jonathan Dietrich, and Milos Malesevic fostered a conducive and collaborative work environment, which significantly contributed to the success of this thesis.

To my dear friends, your unwavering support has been a constant source of strength and inspiration. I extend my sincere thanks to Tong Liu for being a pillar of support, sharing joyful moments, and being a pillar of support during both my academic and professional pursuits. Special gratitude is also extended to Jin Zhang, Chun-Hung Tseng, and Yifei Li for their steadfast help and support in both my personal and professional life in Zurich.

Lastly, but certainly not least, I extend my deepest appreciation to my family for their firm financial support, which enabled me to pursue my studies at EPFL and reside in Switzerland. Without their sacrifice and encouragement, none of this would have been possible.

I am also grateful to HexHive Lab and Mathias Payer for providing the EPFL thesis report LaTeX template, which formed the basis of this report.

Lausanne, March 1, 2024 Guochao Xie

Abstract

Software dependency analysis summarizes the usage of packages and artifacts in the target software, crucial for vulnerability tracking to enhance security and license compliance validation to avoid legal issues. However, existing approaches are often tailored to specific build tools, resulting in a complex and fragile ecosystem, which demands expert knowledge of the interface or implementation of every specific build tool and is costly to develop and maintain. This becomes particularly evident with generic build tools like Make, where dynamic and potentially indeterministic build-time behavior hampers effective dependency tracking. To address these **complexities**, we present Deptyque, an innovative eBPF-based build provenance framework designed for software dependency analysis. Deptyque **simplifies** dependency analysis by tracking the build provenance graph, exposing **generic** read-write dependencies between files and processes. Its Online Tracing component is the first eBPF-based build provenance generation tool to trace OS events on VFS and memory mapping functions with low build-time overhead. Meanwhile, it incorporates a **specializable** Post-Processing framework providing rule-based customization for high **correctness**. Currently, Deptyque supports various build tools, including Maven, Gradle, Go, Cargo, Conan, and generic Make builds. Validated through small-scale benchmarks, a medium-sized Micronaut application, and large-scale Firefox builds, Deptyque achieves an impressive precision exceeding 92.5% and a recall of 99.2%. The incurred latency on build process is modest, not surpassing 12.8%, and specialization efforts involve straightforward rules such as file extension matching, call stack matching, and process command identification.

Contents

Chapter 1

Introduction

Software dependency analysis tracks the usage of packages and artifacts, which plays a crucial role in security analysis for several reasons. Firstly, identifying these dependencies during the build phase is instrumental in mitigating the risk of supply chain attacks, as exemplified by incidents such as SolarWinds (2021) [1], the Event-Stream Incident (2022) [4], and Dependency Confusion Attacks (2023) [32]. Secondly, distributing software dependencies to end-users can significantly reduce the effort required to detect and track vulnerabilities after the software release, as observed in cases like Heartbleed (2014) [24] and Log4j (2021) [27]. Furthermore, updating dependencies due to bugs is a necessity. The expense associated with dependency analysis constitutes a one-time effort during software build, yet it streamlines subsequent dependency updates and substantially diminishes operational costs and vulnerability to attacks when the software might be exposed to new bugs or vulnerabilities. For example, by matching the package name and version against Common Vulnerabilities and Exposures (CVE) database, software developers can mitigate the risks quickly by patching or upgrading third-party packages. The significance of dependency analysis is further underscored by governmental entities. In 2021, the US government mandated software providers to conduct thorough analyzes of software dependencies and generate a Software Bill of Materials (SBOM) [35] as a receipt [8, 28]. This thesis centers on CycloneDX, one of the most widely recognized SBOM formats.

Dependency analyzes can be categorized into two main types: build-tool-specific analyzes, which involve components or plugins integrated with the build tool, and generic analyzes, which necessitate less knowledge of specific build tools and accommodate diverse build environments. The former usually offers precise and granular identification of software dependencies, leveraging expert knowledge of the build tools. For instance, plugins like CycloneDX-Maven-Plugin [20] and CycloneDX-Gradle-Plugin [19] consume build-tool internal information through build tool plugin handlers. Additionally, tools like Cargo-CycloneDX [21] and CycloneDX-Go [18] utilize internal package information through dependency declaration or lock files exported by the build tool. In contrast to build-tool specific dependency analyzes, the potential exists to generalize and simplify

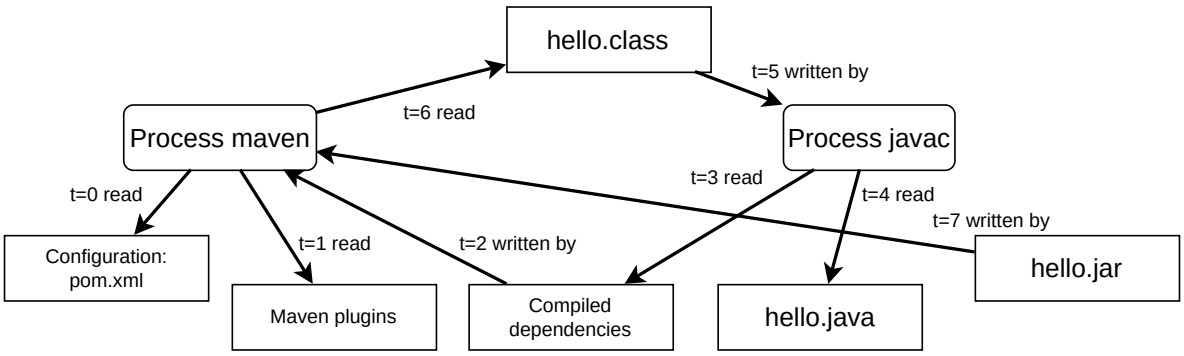

Figure 1.1: Simplified Build provenance: Maven builds "hello.jar".
The Maven process initially reads the configuration and loads plugins. It then downloads compiled dependencies and forks a Javac process. The Javac process reads both the compiled dependencies and source code to generate class files. Finally, the Maven process reads the class files and packages them into the final artifact "hello.jar".

dependency tracking across multiple build tools at the operating system level, although there is currently no existing implementation.

Build provenance is a suitable medium in generic software dependency tracking. According to the World Wide Web Consortium (W3C), provenance refers to information about entities, activities, and individuals involved in the creation of a data element or artifact [7, 34]. This information can be instrumental in forming assessments related to the quality, reliability, or trustworthiness of the artifact. In the context of this thesis, we define build provenance as involving two key elements: entities, which comprise both processes and files, and relationships, which are represented by interactions between them. Figure 1.1 illustrates an exemplar of build provenance, providing insight into the Maven compilation process for a "hello" project. In this context, we assume a **generic** read-write dependencies that a file being written by a process has the potential to depend on all files read by the same process before the writing occurs, with the consideration of transitive dependencies.

Build provenance is essential for software dependency analyzes following our read-write dependency assumption. Genericity mandates minimizing the reliance on build-tool-specific knowledge, favoring common event patterns across various build environments. As a standardized data record of operating system (OS) events during the build process, build provenance provides sufficient information to unveil not only potential direct dependencies but also transitive relationships between processes and files.

While tools like CamFlow [38] can generate build provenance, tracking software dependencies with these tools poses challenges. Firstly, lacking specialization for dependency analysis, they generate excessive information, rendering dependency analysis impractical at scale. For instance, instrumenting syscalls unrelated to process and file systems introduces overhead and noise in post-processing. Secondly, their instrumentation is coarse-grained at the kernel level without exposing

user-space information such as user-space stack trace of events, making it challenging to minimize false positives and negatives. Thirdly, designing a provenance graph processing solution to perform dependency analysis requires considering trade-offs between the **simplicity** of **specialization** and the analysis **correctness**.

In this thesis, we design and implement **Deptyque**, an eBPF-based build provenance framework designed for software dependency analysis. Our primary focuses are exploring the possibility of leveraging Linux eBPF for build provenance generation and developing a simple-to-customize provenance processing framework for dependency analysis. Our solution demonstrates support for multiple compiled languages and their build tools, including Maven, Gradle, Cargo, Go, and Conan. Additionally, developers can declaratively add support for more build tools in JSON or imperatively through Python scripts. Supplementarily, Deptyque incorporates an Offline Discovery phase for finding essential functions to guide the eBPF probe instrumentation. We assess our framework's correctness and measure the additional overhead of eBPF loading, online instrumentation, and Post Processing by comparing outputs with state-of-the-art build-tool-specific analyzers, showcasing its usability in real workloads.

The primary challenge lies in the balance between **genericity** and **specialization**. While a straightforward build provenance that reveals the read-write dependencies between files and processes in the build process can reveal software dependencies, it compromises the analysis granularity, indicating that Deptyque should also support specialization to increase correctness. For example, distinguishing between the build tool and the artifact's dependencies is crucial to reduce false positives and improve the usefulness of dependency analysis. Furthermore, such specialization should minimize efforts to configure.

We will demonstrate that Deptyque achieves 100% precision and recall for standard usage of Maven, Gradle, Cargo, Go, and Conan, while over 92.5% precision and 99.2% recall for the complex builds of Firefox. Simultaneously, eBPF instrumentation incurs only a 12.8% overhead on build time and statically 10 seconds on eBPF compilation and loading. The configurable eBPF memory reserved in our current setting is around 3.71GiB out of 64GiB. The support for these build tools illustrates the simplicity and flexibility inherent in extending build-tool support.

Our core contributions consist in the following parts:

1. Design and development of an eBPF-based build provenance framework;
2. Development of a specializable provenance processing framework for dependency analysis;
3. Implementation of a polynomial algorithm for provenance graph processing;
4. Introduction of a supplementary tool for exploring useful probes;
5. Provision of specialization examples supporting the following build tools: Maven, Gradle, Cargo, Go, Conan, and Make.

Chapter 2

Background

2.1 Software Dependency Analysis

Software dependency analysis is a crucial component of software security, which provides a comprehensive view of the software supply chain [37, 41]. It assists developers and users in identifying all packages utilized in the build tool, compiler, runtime, and testing processes. This section will elucidate the receipt (Software Bill of Materials), the scopes, build-tool-specific generation, and the practical applications of software dependency analysis.

2.1.1 Software Bill of Materials

The Software Bill of Materials (SBOM) functions as a documentation tool for software dependency analysis, providing a structured inventory that classifies software components, including libraries, packages, and modules, along with their transitive dependencies. Recognized SBOM formats, such as CycloneDX, SPDX, and SWID, present summaries of software dependencies with varying levels of metadata granularity. In this thesis, CycloneDX is chosen as the representative SBOM format.

Within CycloneDX, the two primary fields are the components and their inter-dependencies. The *components* field lists all software packages, categorized by dependency scopes (refer to subsection 2.1.2). Meanwhile, the *interdependencies* field records the hierarchical relationships among components to trace the usage of dependencies. Since Deptyque lacks the capability to retrieve inter-dependencies, our focus is on extracting the components.

2.1.2 Dependency Scopes

Dependency scope dictates how a dependency influences the artifact, with varying granularity and names across build tools and standards [5]. For simplicity, we categorize them into four scopes: build-time, compiled, runtime, and test (as presented in Table 2.1).

Build Tool	Build-Time	Compiled	Runtime Only	Test
CycloneDX	-	required	optional	excluded
Maven	plugins	compile, provided, system	runtime	test
Gradle	plugins	api, compiledOnly	implementation,runtimeOnly	test*
Go	-	included	-	included
Rust	build, proc-macro	default	-	dev
Conan	-	default	shared	test

Table 2.1: Summary of dependency scopes.
- Gradle: test* includes testImplementation, testCompileOnly, and testRuntimeOnly.

First, **build-time dependencies** are components involved in build tools. These dependencies may vary with different build environment configurations, impacting the produced artifact. For example, a vulnerable component within Maven could permit remote code execution when processing network requests, as demonstrated by CVE-2020-13956 [16]. Similarly, an impacted version of Gradle may enable remote attackers to execute arbitrary code, as illustrated in CVE-2016-0785 [15]. Although essential for identifying build tool vulnerabilities, they are often overlooked in static dependency analysis and SBOM. Deptyque can identify build-time dependencies for JVM frameworks.

Compiled dependencies are usually the most important ones, which include packages used during compilation for type checking, reflection and runtime information, and bytecode verification. For example, Javac reads JAR files and gcc reads source files. Most dependency analyzes include this scope and Deptyque considers it as the key metric for dependency coverage.

We define **runtime dependencies** as those required solely at runtime, which provide the flexibility to load a specific implementation separated from the interfaces. By design, these dependencies do not need to be determined during software build. Deptyque currently does not support runtime dependency analysis but focuses on other scopes.

The last dependency scope is **test dependencies**, which are utilized in software testing. Usually these dependencies do not have effect on the software security. Some languages like Java compile tests into classes during the build process. Deptyque can analyze their test dependencies using the same methodology as compiled dependencies without extra specialization.

2.1.3 Build-Tool-Specific SBOM Generation

Most existing dependency analyzes are tailored to build tools through information extracted by the build tool, falling into two main categories: those provided directly by the build tool and those implemented as plugins.

Build tools like Go and Cargo **seamlessly integrate** the dependency analysis and SBOM generation. Executing commands such as *go mod graph* and *cargo tree* yield software dependencies, which can then be used to generate SBOM. Maintenance of this type of dependency analysis is typically handled by the build tool developers and requires in-depth understanding of the build tool implementation.

Plugin-based dependency analysis adopted by Maven and Gradle is a different approach. By providing hook to export dependency information during the build process, external community developers can implement various dependency analysis plugins for different use cases such as *CycloneDX-Maven-Plugin* and *SPDX-Maven-Plugin*. This approach requires knowledge of the plugin interfaces provided by the build tools.

2.1.4 Applications of Software Dependency Analysis

Software dependency analysis plays a pivotal role in detecting vulnerabilities and tracking licenses. Firstly, dependency analysis is crucial for identifying new **vulnerabilities** like Heartbleed[24], SolarWinds [1, 46], Log4j [27] and Event-Stream Incident [4]. By comparing package dependencies with vulnerable versions, developers receive alerts and can promptly upgrade affected dependencies to mitigate potential security risks.

Furthermore, through dependency analysis, developers can query the **licenses** of each dependency, ensuring compliance and verifying the compatibility of licenses. Adhering to software licenses is vital when utilizing open source packages [40]. For example, inadequate compliance, closed source distribution, omitting credits, unlawful changes, and unauthorized commercialization might lead to legal disputes, reputational damage, and a breakdown of trust within the open source community.

2.2 Data Provenance and Its Use Cases

Data provenance encompasses the origins of data and the processes through which it has evolved to its current state. It delves into the intricate connections between data entities, the transformations applied to these entities (activities), and the individuals or organizations linked to the data and transformations (agents) [10]. The primary objective is to construct a thorough and dependable

record of data lineage, yielding applications that are particularly pertinent to the fields of scientific computation and database systems.

In **scientific workflows**, data provenance elucidates the processes involved in generating intermediate data, diagrams, and visualizations. This tracing capability is indispensable for ensuring the trustworthiness, reproducibility, and educational value of the data, facilitating robust data exploration [22, 42].

Data provenance within **database systems** involves the meticulous tracking of data sources and intermediate computations pertaining to database queries [9, 12]. This approach finds application in diverse areas such as data annotations [45] and data integration [30], enhancing the overall transparency and reliability of data management.

2.3 Operating System Event Tracing

Operating system (OS) event tracing stands out as a highly effective method for generating OS-level provenance. An event trace encapsulates essential details such as timestamps, involved processes and files as entities, activity specifics, and supplementary information at various granularity, forming the foundation for deriving comprehensive data provenance.

2.3.1 eBPF

eBPF, a bytecode format providing strict typing and checking, is one of the best code instrumentation candidates: Linux kernel supports executing eBPF codes without necessitating changes to the kernel or the loading of a kernel module. Figure 2.1 demonstrates an overview of eBPF for tracing. During development, BCC compiles eBPF source code into bytecode, which is loaded dynamically to kernel and user probes. The data storage includes eBPF maps and ring buffers. Multiple eBPF programs can share the same eBPF map for communication and state storage, while user space controllers can also access the maps through eBPF virtual file system. Moreover, eBPF programs can efficiently submit data to ring buffers where user space controllers can consume them asynchronously.

eBPF has been popularily adopted for OS observability and visibility. The bpftrace [26], BCC [29] and eBPF-go [13] are toolkits to compile and generate eBPF bytecode using high-level languages. Based on these frameworks, Cilium [33] and Tetragon [14] cooperate to enhance container networking and security. Falco [25] is another example, using eBPF for real-time security threat detection by collecting OS event logs. The success of these products proves eBPF as a compelling choice for instrumentation and provenance generation.

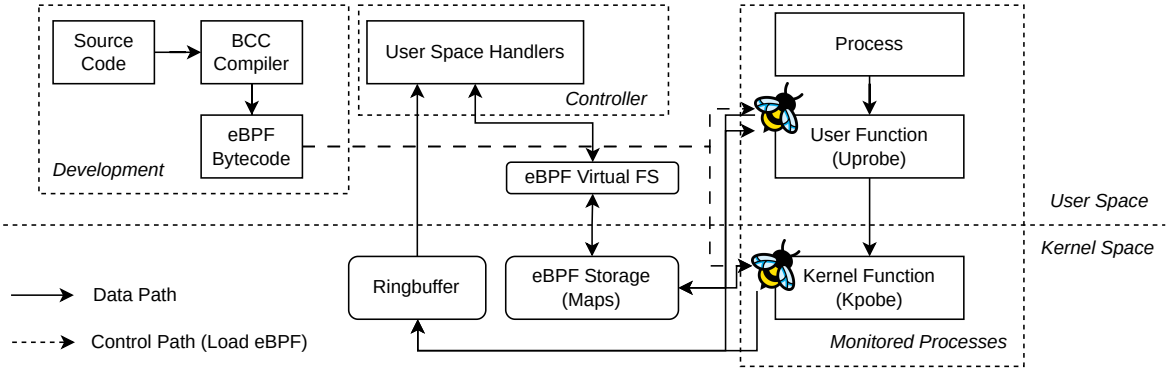

Figure 2.1: Overview of eBPF for tracing.

Chapter 3

Design

We will delve into the design goals and system architecture of Deptyque from the viewpoints of end users and build tool developers. Subsequently, we will elucidate the primary challenges faced by Deptyque and the solutions designed to address these issues.

3.1 Goals

Our objective is to develop a **generic** dependency analysis framework that supports multiple build tools with minimized **specialization** efforts. Specifically, Deptyque is designed to accomplish the following goals:

1. **Correctness**: Deptyque should maximize the precision and recall of dependency analysis as well as minimize the false positives and false negatives.
2. **Simplicity**: Deptyque should require minimal specialization for various build tools, ensuring ease of use and adaptability.
3. **Low Overhead**: Deptyque should minimize the impact on the software build latency and resource consumption, ensuring efficiency and practicality,

Furthermore, our work has two fundamental assumptions:

1. **Build Environment**: We assume the build process runs on a Linux container and Deptyque has the permission to load eBPF programs and instrument the build container. The usage of containers aligns with the trend in modern software development (i.e. CI/CD pipelines).

2. **File-Based Data Exchange**: We assume that processes only exchange data through Linux Virtual File System (VFS) and there is no implicit memory sharing between processes through fork. Instead, the prevalent "fork and exec" pattern is employed in the multi-processing of build tools. Processes exchange data through the Linux virtual file system, including files such as packages, download caches, temporary compilation caches, and virtual files like TCP sockets. This assumption is crucial to minimize false data dependencies across process forking.

While we focus on software builds within a Linux container, Deptyque remains adaptable and can seamlessly transition to a virtual machine environment.

3.2 System Overview

Deptyque's primary concern lies in figuring out a delicate balance between **genericity** and **specialization**. While a generic approach simplifies dependency analysis, it may compromise its accuracy due to a lack of specialization and the coarse granularity of information collected. In the design of Deptyque, we demonstrate our careful consideration of this trade-off.

To ensure **simplicity**, Deptyque primarily employs probes within the Linux Virtual Filesystem (VFS), a strategy detailed in subsection 3.3.1. These probes suffice to establish a rudimentary build provenance, outlining the read-write dependencies between files and processes. However, while this generic build provenance serves its purpose, it fails to provide the granular insights necessary to distinguish the specific usage patterns of read and write operations. Furthermore, the distinction between build-time and compiled dependencies is blurred, resulting in useless dependency analysis.

To address this limitation, Deptyque employs a straightforward **specialization** solution: the collection of user-space library function calls, detailed in subsection 3.3.3. This is facilitated by Deptyque's eBPF-based lightweight call stack collection mechanism, discussed in subsection 3.3.2, which imposes **low overhead** on the build process. Deptyque also includes a specializable Post Processing phase (in section 3.4) that users define rule-based specialization to consume build provenance and output analysis result.

3.2.1 Essential Components

Deptyque comprises two main phases: Online Tracing and post-processing. Optionally, it includes an Offline Discovery phase to facilitate the identification of build-tool-specific probes, supporting the integration of new build tools. Build containers, eBPF instrumentation, probes, and events data are four key concepts to understand the design of Deptyque.

As shown in Figure 3.1 and Figure 3.2, the basic runtime environment is a **build container**, a container with the build environment installed running the software build. Users mount the source code directory and optionally package caches to the container. The container provides a clean environment with Overlay Filesystem and Mount Namespace to reduce the instrumentation noise and simplify data cleaning before Post Processing.

Both Offline Discovery and Online Tracing instrument a build container using **eBPF**. The eBPF programs collect and export parameters and return values of function calls, as well as related data reachable with pointer traversal. Different consumers are used for Offline Discovery and Online Tracing to analyze potentially useful probes and produce OS events data respectively.

Probes are the kernel and user functions to attach eBPF programs. Most kernel probes are pre-defined to instrument syscalls at the VFS level because of their genericity. For some languages like Java, developers can use Offline Discovery to summarize interesting user-space probes at the shared-library level to provide granular information. The number of enabled probes affect the overhead of the build container, which can be controlled by the Configurations.

Events data, as an intermediate product between Online Tracing and Post Processing, is a sequential log of OS events captured by eBPF programs. Each event data entry includes a timestamp, the related process IDs and dentry addresses, a lightweight call stack, and optionally some other metadata for specific probes. Post Processing utilizes these data to generate build provenance and analyze software dependencies.

We will present Deptyque from the perspectives of both end users (as in Figure 3.1) and build tool developers (as in Figure 3.2.)

3.2.2 Overview for End Users

End users leverage Deptyque for the collection of build provenance and software dependency analysis. Illustrated in Figure 3.1, they input their build containers for Online Tracing and provide configurations, including the artifact's path and enabled specializations, for post-processing. Deptyque produces build provenance as an intermediate output and an SBOM for dependency analysis as a final output. We pre-define essential probes for our currently supported build tools; if desired, end users can supplement these with build-tool-specific probes and configurations. In subsection 3.2.3, we will show how build tool developers can generate these additional probes and configurations using Offline Discovery to achieve more granular analysis.

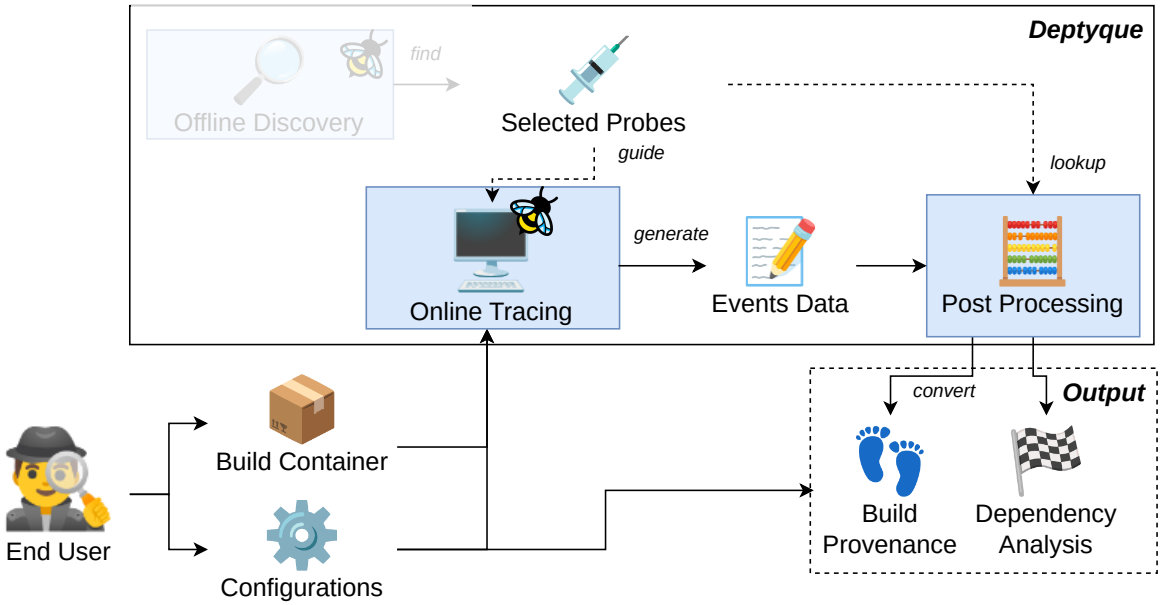

Figure 3.1: Deptyque Overview of end users.
Goal: A simple dependency analysis.

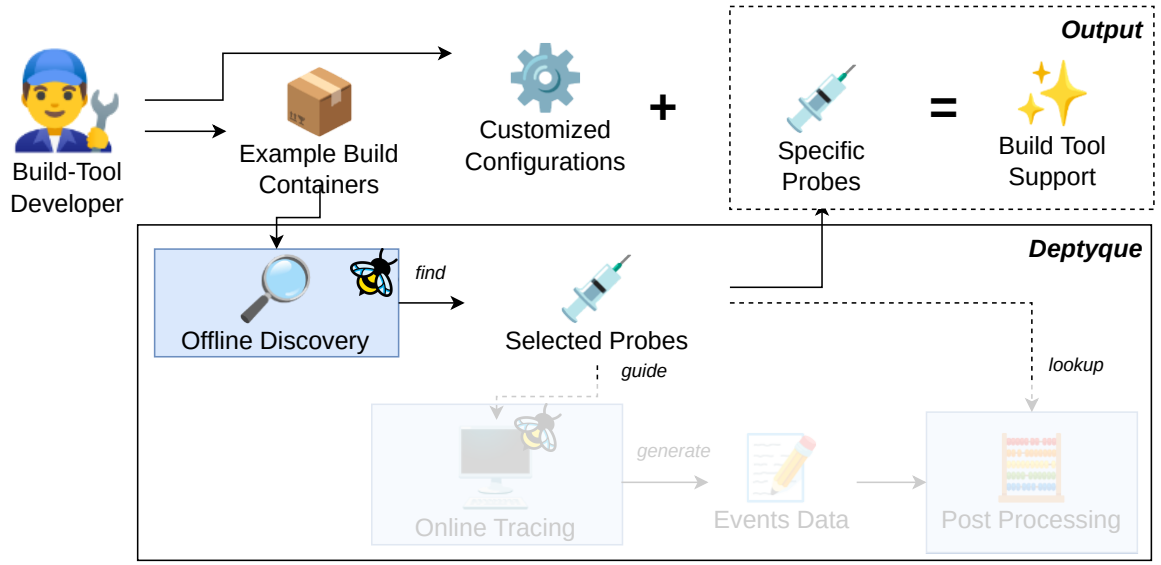

Figure 3.2: Deptyque Overview of build tool developers.
Goal: Specialization of a specific build tool.

3.2.3 Overview for Build Tool Developers

For detailed dependency analysis, build tool developers supply sample build containers to the Offline Discovery, which aggregates common user-space probes during file-system syscalls. Developers can then formulate Post Processing rules, leveraging this extra user-space information for each syscall to enhance the analysis accuracy of a build tool. As shown in Figure 3.2 the build-tool-specific probes, along with customized configurations, collectively constitute the support for a new build tool, which end users can reuse to enable specialization of the specific build tool.

3.3 Online Tracing

The Online Tracing phase is designed to gather event data using eBPF, which will subsequently serve as input for the Post Processing phase. In this section, we will elaborate on how we collect OS event data at the VFS and dentry levels, alongside lightweight call stacks, to offer fine-grained user-space information.

3.3.1 Generic Probes for Simplicity

Probes are eBPF programs that are dynamically executed on the entry or exit of a kernel or user-space function call. Operating within the kernel space, eBPF programs have privileged access to both kernel and user-space data. The strategic selection of probes is crucial for achieving **simplicity** and **genericity** in instrumentation.

While syscall probes, often employed by tools like strace [44], are a prevalent option, challenges arise in dealing with file representations using relative path strings and file descriptors. The primary hurdle involves obtaining absolute paths from either a relative path with its current working directory (cwd) or a file descriptor. The kernel refrains from utilizing or resolving absolute paths, opting instead to parse relative paths and traverse corresponding dentry and inode structures for operations. Additionally, the complexity and size constraints of eBPF pose difficulties in implementing resolution of absolute paths. In user space, resolving absolute paths is also difficult because it needs to maintain states of directories, files and the current working directory.

We observe that the dentry data structure is widely used in VFS and file-backed memory mapping functions. Each dentry structure includes both the name and its parent dentry pointer, which can be used to recursively resolved the absolute full path. While reclaiming and reusing dentries occur frequently, which indicates that the same dentry address might point to different files at different timestamps, a file or directory can be uniquely identified by the combination of a dentry memory address and a timestamp. Consequently, Deptyque records *dentry* addresses instead of path strings to represent files for each event.

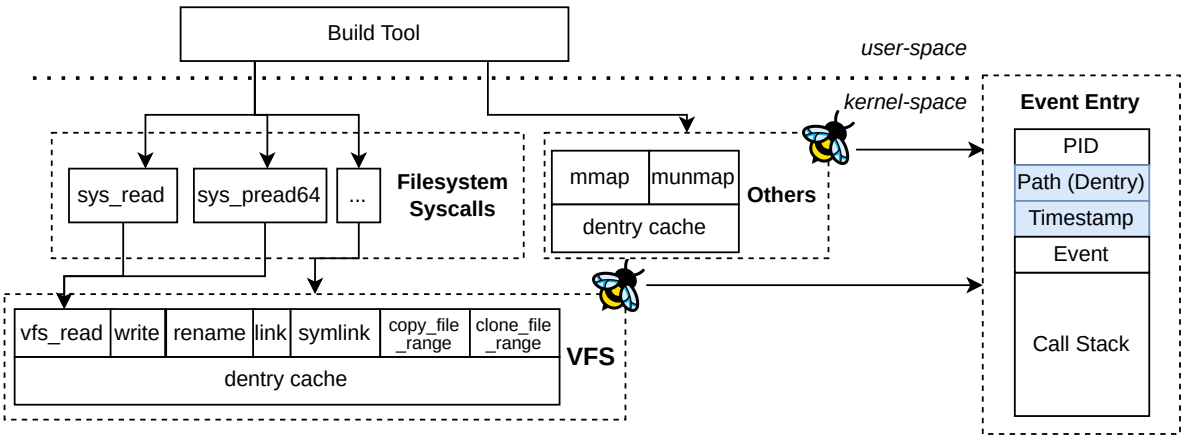

Figure 3.3: The eBPF instrumentation detail.

In our design of generic probes, we choose to instrument filesystem kernel functions at the Virtual File System (VFS) level, as depicted in Figure 3.3, for simplicity. By targeting fewer VFS functions, we can effectively cover filesystem syscalls, considering that multiple syscalls may share the same underlying VFS functions. Furthermore, eBPF functions gain direct access to function parameters, typically "dentry" or "file" structures. The "file" structure conveniently encodes the corresponding "dentry". Consequently, we unify the file system provenance collection at the VFS level using "dentry" as the tracing object.

Furthermore, file-based memory **mmap** and **munmap** functions are equivalently essential, representing another form of interactions between files and processes bypassing the vfs_read and vfs_write. If the dentry address is available for these two cases, Deptyque records the event for build provenance.

Deptyque decouples dentry memory address recording and full path resolution for simplicity and eBPF-restriction reasons. Because eBPF programs have limited program size and strict safety requirements, the eBPF verifier will decline the invocation of kernel's path resolution functions. Instead, Deptyque uses the combination of dentry address and timestamp as a unique identifier of files during Online Tracing and resolves their full paths during Post Processing, which we will present in section 4.2.

3.3.2 Lightweight Call Stack Collection for Specialization

Event call stacks offer detailed information about probe callers to support build-tool **specialization**, which compensate the **inaccuracy** of using generic functions. For instance, if a vfs_read is issued by a JVM class loader related function, Deptyque can identify it as a build-time dependency instead of a compiled dependency. This distinction is often overlooked by existing tracing and provenance tools that do not delve into stack traces, resulting in a loss of valuable information.

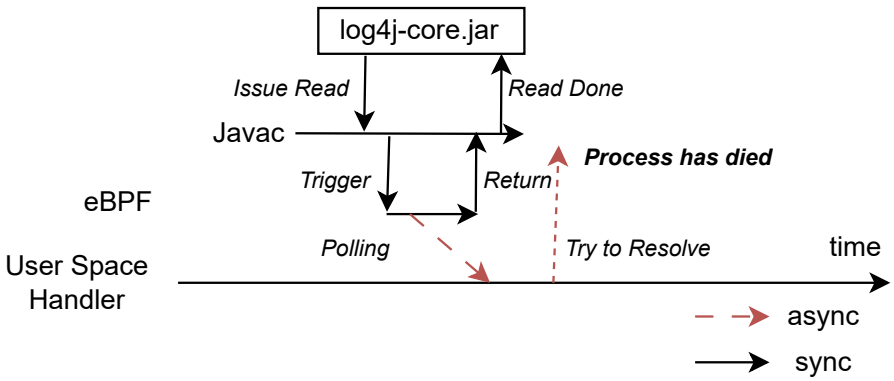

Figure 3.4: Existing eBPF functionality cannot resolve symbols after a process dies.

Obtaining a call stack per event poses a challenge with existing eBPF capabilities. While for long-lived applications like web servers, eBPF supports capturing a stack trace of callers' addresses, which user space handlers can translate into symbols using the process ID (PID), this approach is ineffective for short-lived build processes as shown in Figure 3.4. The main reason is that the instrumented process might have died before resolving the addresses' symbols because of the latency between event emission and handling can be high. The latency includes the queuing latency and the relatively slow user-level event handlers in Python, Go or other high-level languages. In such case, the symbol table of the dead build process will be unavailable. Furthermore, while in principle resolving address symbols without a live process is possible, implementing it with eBPF would incur significant difficulties especially considering the dynamic loading of libraries during build time.

To address these challenges, we introduce a eBPF-based lightweight call stack collection mechanism, as illustrated in Figure 3.5. This involves implementing a map-of-array structure for each ongoing thread, where the call stack is a fixed-size array of integers, with each element corresponding to a pre-defined caller table of function names. During an entry probe, eBPF pushes the probe ID to the thread's call stack, while an exit probe pops the probe ID. Event data submitted to the user space handler includes the call stack, and caller IDs are translated into names during post-processing.

Through co-designed Online Tracing and post-processing, we selectively choose functions essential for software analysis to instrument, reducing eBPF program size and execution overhead, while also minimizing irrelevant information for Post Processing.

An important consideration is the coverage of essential functions, influencing the quality of software analysis. Despite covering most in-kernel syscall-related and some user-space runtime library (i. e. JVM) functions with tests of example builds, there remains a possibility of missing some for specific build tools. To address this, we introduce an Offline Discovery phase to enhance function coverage, automatically collecting and summarizing hot functions, as discussed in section 3.5.

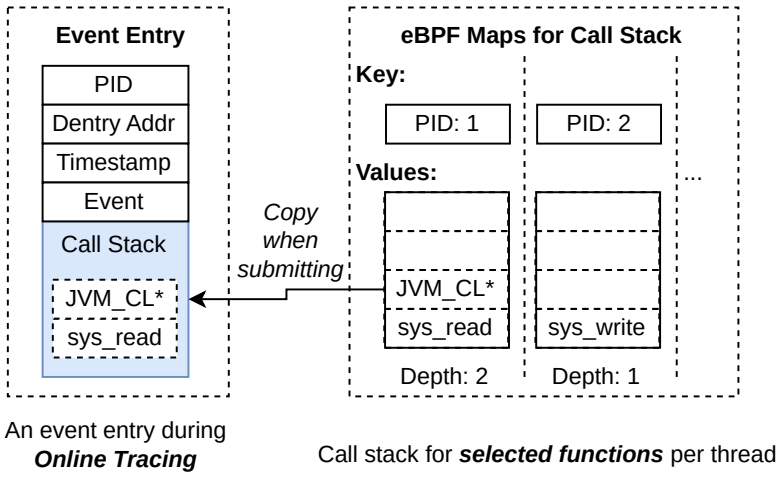

Figure 3.5: Overview of eBPF-based lightweight call stack collection.
JVM_CL*: JVM class loading related functions.

3.3.3 User-Space Library Instrumentation For High Correctness

The absence of user-space information stands out as a prominent limitation in current provenance frameworks, particularly affecting interpreter languages and build tools employing dynamic loading of plugins and codes. The lack of user-space information leads to an inability to distinguish between build-time and compiled dependencies, significantly increasing false positives in software analysis. For instance, JVM build tools like Maven and Gradle read both build-time dependencies, such as Maven and its plugins' JAR files, and compiled dependencies, representing libraries used by the artifact. Despite their distinct nature, both types are treated as read syscalls by the kernel, resulting in a provenance with **high false positives** and limited utility.

To address this issue, we employ eBPF to instrument user-space libraries, represented by the .so files within the build container. Our eBPF programs, acting as entry and exit handlers (see subsection 3.3.2), capture crucial user-space caller information, particularly valuable for JVM and analogous languages and build tools. In Figure 3.6, recording function calls from the JVM class loader, such as "JVM_FindClassFromCaller" and "SystemDictionary::load_instance_class_impl", enables differentiation between reads to JAR files for JVM class loading (indicating build-time dependency) and other purposes. We have verified the utility of fine-grained syscall information on JVM build tools Maven and Gradle, with the expectation that other JVM build tools will also benefit.

Additionally, this design applies to other user-space libraries including *libc* and *ld*. For example, when instrumenting *dl_open* related functions, it would be on our eBPF-based call stack which is usable for the Post Processing rules to identify build-time and runtime dependencies. To find these relevant functions for instrumentation, build-tool developers can utilize the Offline Discovery phase without requiring in-depth knowledge of the build tool's implementation. Further details on the

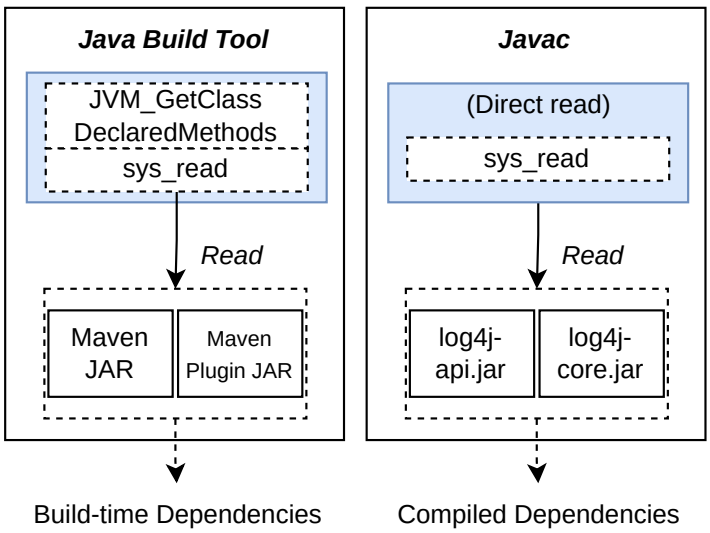

Figure 3.6: Example of differentiating JVM Build-Time and Compiled dependencies by instrumenting JVM libraries.

Offline Discovery phase will be discussed in section 3.5.

3.4 Post Processing

The Post Processing of Deptyque incorporates three steps as illustrated in Figure 3.7: build provenance generation, annotation, and output. In this section, we will explain the design of these three steps, the design of Post Processing rules, and one greedy algorithm we design to process temporal provenance.

3.4.1 Key Steps in Post Processing

In Post Processing, Deptyque consumes the *events* data from the Online Tracing and produces the build provenance and dependency analysis. Firstly, Deptyque preprocesses the raw event data to obtain the file path and filters some irrelevant events selectively. The resolved event data includes five components. Deptyque creates process vertices using the PID and file vertices using the path. To determine the edge direction, Deptyque uses the event type. Furthermore, each edge also includes the timestamp (interval) and the call stack as metadata.

Secondly, Deptyque applies annotation rules to the provenance graph. The annotation rules are simple to define and use. For example, we define annotations based on the file extension to filter JAR files. In section 4.3, we will show the specialized rules we define for our supporting build tools.

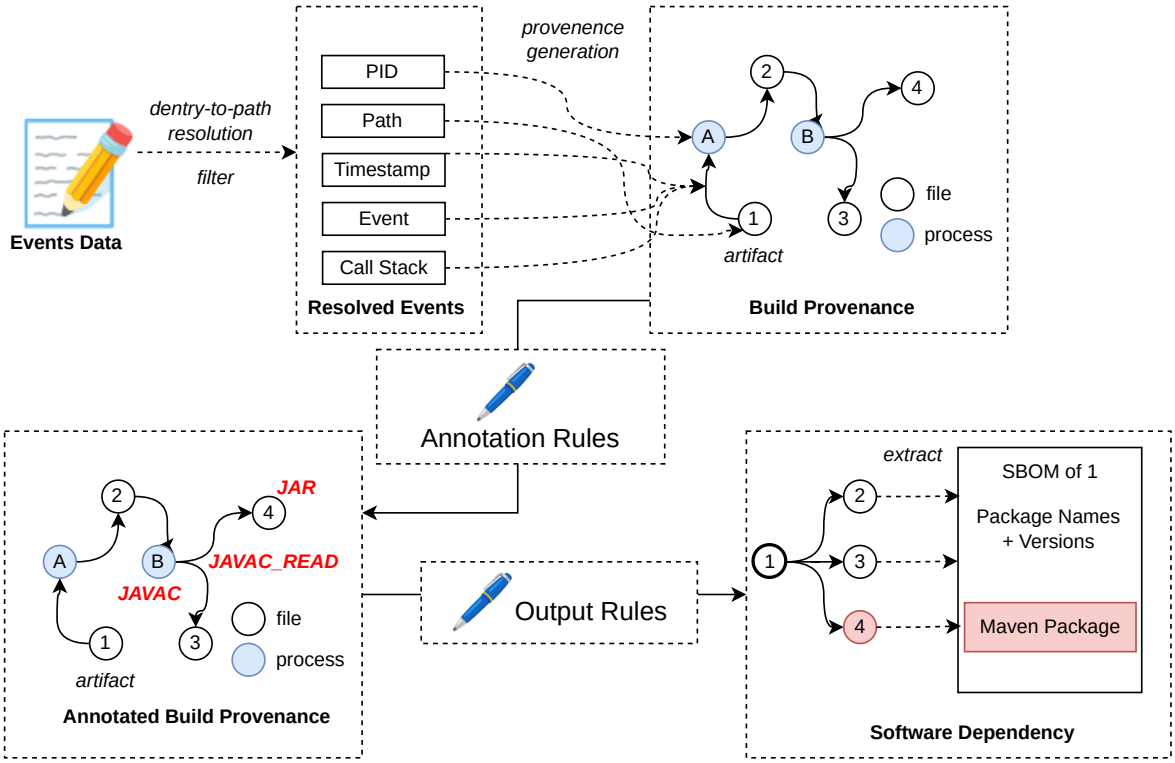

Figure 3.7: Overview of Post Processing.

Lastly, Deptyque outputs the analysis result using output rules which filter reachable vertices by annotations. Users can further extract package information using the path (as our current solution) or the file checksum (when a pacakge database is available.)

3.4.2 Design of Annotation and Output Rules

Deptyque supports both Edge and Vertex rules of annotation and output for fine-grained processing. In this section, we will briefly explain both types of rules with an illustrative example.

Since there might be multiple edges between two vertices on the provenance graph with different event call stacks or disconnected timestamps, Edge rules are necessary to understand individual events, which can access data of the edge as well as the two end vertices. In contrast, Vertex rules only look into the detail of vertices. Users can configure the order of annotation rules. After annotation phase, the edge annotations will be moved to one end of vertex depending on its configuration. Eventually, Deptyque scans all vertices and exports analysis results using Output rules.

Figure 3.8 shows some example rules of a Java build process and Listing 3.1 shows their pseudo-code implementation. The Vertex rules *JAR* and *JAVAC* identify JAR files by file extension and Javac process by process command. Then, the Edge rule *JAVAC_READ* utilizes the both end vertices' annotations *JAR* and *JAVAC_READ*; additionally, it checks JVM class loading functions are not in the call stack. The *JAVAC_READ* annotation will be propagated to the vertex *log4j-api.jar*. Eventually, the *JAVA_DEPENDENCY* Output rule looks for vertices with both *JAVAC_READ* and *JAR*.

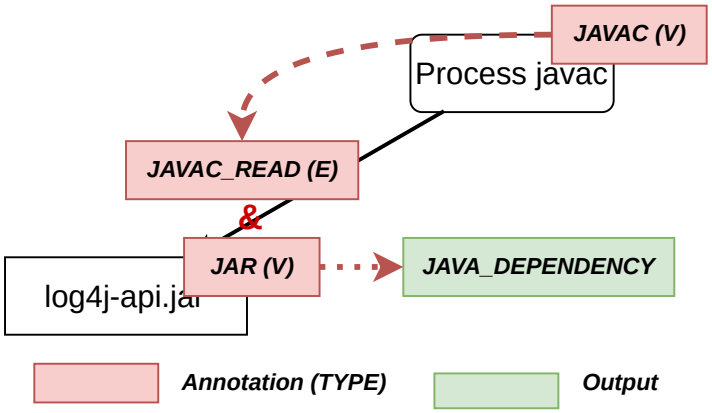

Figure 3.8: Example of Annotation and Output rules.
- V: Vertex rules.
- E: Edge rules.

Listing 3.1: Pseudo Code of Example Annotation and Output Rules

```
# Annotation Rules
```

```
IF V.extension=".jar"
    THEN annotate JAR.

IF V.command="javac"
    THEN annotate JAVAC.

IF (
    E.v_from.annotations include "JAVAC"
    and E.v_to.annotations include "JAR"
    and E.call_stack.intersect(JVM_CLASS_LOADING_FUNCTIONS) = none
) THEN annotate JAVAC_READ.

# Output Rule

IF (
    V.annotations include "JAVAC_READ"
    and V.annotations include "JAR"
) THEN output JAVA_DEPENDENCIY.
```

3.4.3 Efficient Temporal Provenance Processing: A Greedy Algorithm

The primary characteristic of temporal provenance is the inclusion of a timestamp or interval associated with each relationship, represented as an edge in the graph. The rationale behind this is to link writes exclusively to reads issued beforehand, aiming to minimize false causality. A proposed solution advocates duplicating nodes with different timestamps to convert the temporal provenance graph into a time-insensitive one [6] and adopt previous works on provenance processing [2, 31, 36] and storage [11, 23, 47].

However, this solution presents challenges in terms of both space and time complexity. In a build provenance scenario, a single process may issue interleavin reads and writes, resulting in a significantly larger graph.

To address this, we propose a greedy algorithm to process temporal provenance in place. The approach involves examining a directed graph where each edge has a time interval and a source node. The goal is to identify all nodes that the source node can reach. A path is deemed valid if every edge on the path concludes after all its descendants commence.

Algorithm 1: Processing Temporal Build Provenance

Data: **V**: set of vertices, **sourceV**: the source vertice, and **E**: set of edges, while each **e** in **E** has an interval

Result: **RE** and **RV**: Set of reachable edges and vertices from **sourceV**

Initialize: Priority queue **Q.init(sorted by interval end descendingly)**; **RE.init()**; Set of visited edges **VE.init()**; Set of visited vertices **VV.init()**; End threshold **ET** $\leftarrow +\infty$.

for $e \in$ *sourceV.outgoing_edges:* **do**
    VE.add(**e**) **Q**.enqueue(**e**)

while *Q.len > 0* **do**
    e $\leftarrow$ **Q**.dequeue()
    RE.add(**e**)
    ET $\leftarrow$ min(**ET**, **e**.end)
    if *e.node_to* $\notin$ *VV* **then**
        VV.add(**e**.node_to)
        for $g \in$ *e.node_to.outgoing_edges:* **do**
            if $g \notin$ *VE* **then**
                VE.add(**g**)
                if *g.start* $\leq$ *ET* **then**
                    Q.enqueue(**g**)

RV $\leftarrow$ Set(**RE**[*].node_to)

return *RE, RV*;

We extend the Dijkstra algorithm to incorporate temporal constraints, as demonstrated in algorithm 1. The primary invariant is the non-increasing nature of the end threshold (**ET**) and the singular processing of each vertex and edge. This guarantees the visitation of edges and vertices on all feasible paths exactly once. The overall time complexity is $O(|E| \times \log|E| + |V|)$.

We will briefly demonstrate the correctness of the algorithm. Initially, we employ **VE** and **VV** to ensure that each edge and vertex is visited at most once, guaranteeing the algorithm's termination in $|E|$ iterations.

Furthermore, **ET** maintains the minimum end timestamp among all reachable and processed edges. If **e.start** $\leq$ **ET**, the edge starts before all reachable edges thus far, making it reachable as well. Conversely, if **e.start** > **ET** and the edge is reachable, it has a higher timestamp and should have been processed **before** the edge with the end timestamp **ET**. Consequently, our algorithm is assured to terminate without overlooking any potential edge or vertex.

The overall time complexity is $O(|E| * log|E| + |V|)$. Initially, the enqueue and dequeue operation can be executed at most once for each edge. Subsequently, each edge and vertex is visited at most once, resulting in a complexity of $O(|E| + |V|)$. Therefore, the combined time complexity is calculated

as $O(|E| * log|E|) + O(|E| + |V|) = O(|E| * log|E| + |V|)$.

3.5 Offline Discovery

As outlined in subsection 3.3.2 and subsection 3.3.3, we designate a set of function probes for generic build provenance, with build tool developers having the flexibility to incorporate additional probes for supplementary user-space insights. However, the task of selecting a minimal set of functions and instrumenting them for dependency classification is non-trivial, often demanding an extensive understanding of the build tool's implementation.

Deptyque introduces an optional Offline Discovery phase to facilitate the identification and summarization of potential probes, aiding in the recognition of build-time dependencies in JVM build tools. The key challenge is the high latency of stack trace symbol resolution as discussed in subsection 3.3.2. In section 4.4, we will describe our solution to resolve symbols for short-live processes as well as how we summarize these call stack symbols to help discover the key probes of JVM Class Loader.

3.6 Design Limitation: Dependency Hierarchy

Deptyque has a fundamental limitation of analyzing dependency hierarchy, the transitive relationships between dependencies because of the lack of genericity across multiple build tools in this task. Most build tools include a package manager and a compiler. Package managers like Maven and Cargo resolve and download the dependent packages. Then, these package file paths are passed as arguments for the corresponding compilers like Javac and Rustc. Since Deptyque adopts a generic approach which analyzes files read by compilers, it can only capture a flat list of all dependencies. In contrast, while instrumenting package managers' internal logics might capture build-tool-specific information like dependency hierarchy, this approach loses the benefits of simplicity. Notably, dependency hierarchy is unnecessary for the use cases of vulnerability scanning and license tracking and a flat list of dependencies is sufficient in practice.

Chapter 4

Implementation

In this section, we will present an overview of the implementation and describe the specializations we employ to support multiple build tools: Maven, Gradle, Go, Cargo, Conan, and Make.

4.1 Implementation Overview of Online Tracing and Post Processing

Figure 4.1 provides an overview of Deptyque's implementation. During Online Tracing, eBPF programs submit data to the Ringbuf ring buffer and eBPF Maps, while Python controllers actively poll data and flush it to disk. Given that eBPF programs generate data at a higher rate than Python consumers can process, we employ multiple eBPF maps and utilize multiple Python processes, with each dedicated to consuming one map. This approach prevents data overflow, allowing Online Tracing to collect raw events and dentry states effectively.

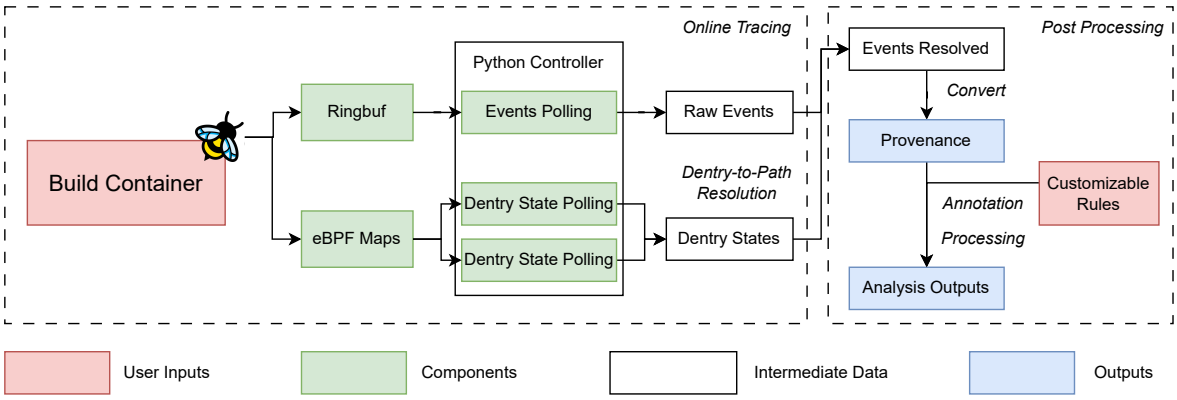

Figure 4.1: Implementation overview of Online Tracing and Post Processing.

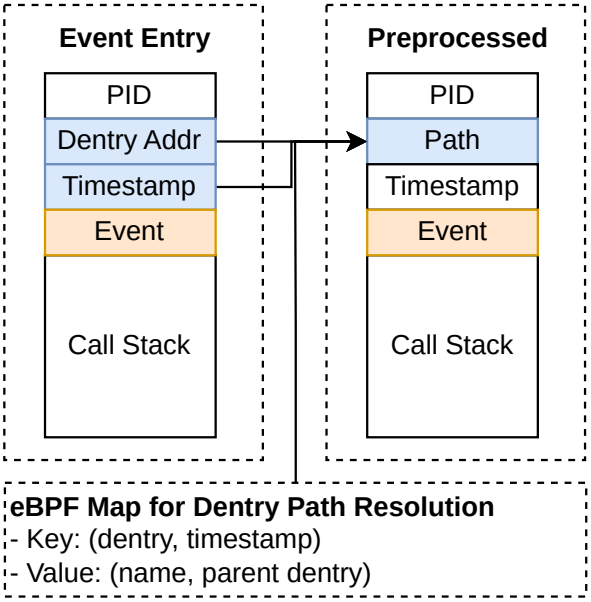

Figure 4.2: Implementation of Dentry-to-Path Resolution.

In the Post Processing phase, Deptyque translates event dentries into full paths. The dentry states encompass the dentry name, its parent dentry, and filesystem metadata at the time of each event. The resolution of dentry-to-path done in user space involves recursive lookups of states to derive complete paths. As an optimization, if the dentry remains unchanged, we store the last timestamp of its modification to eliminate duplicated names and metadata.

Subsequently, Deptyque transforms events into provenance using the syscall type, which can be mapped to one of three relationship types: file-process, process-file, and file-file. File-process relationships involve write and munmap syscalls. Process-file relationships encompass read and mmap syscalls. Moreover, file-file relationships include file renaming and copying.

Lastly, Deptyque incorporates customizable rules for annotation and processing. Annotation rules leverage the paths of vertices, the event type and call stack of edges to annotate each edge and vertex. The detailed discussion of supporting multiple build tools with customized annotation rules will be presented in section 4.3. Following annotation, output rules guide Deptyque to export and summarize data into intriguing paths and visualizations, which undergo further processing as SBOM components.

4.2 Dentry-to-Path Resolution

Deptyque separates the dentry address recording in the Online Tracing and the dentry-to-path resolution in the Post Processing for simplicity and eBPF-restriction reasons. We design the data

structure as shown in Figure 4.2 to support the decoupling.

During Online Tracing, the event-submission probes will record the dentry address of each event entry. They will also maintain dentry states of each timestamp, so that during the Post Processing Deptyque can recursively resolve full paths of each dentry address.

To minimize the memory usage, Deptyque only inserts dentry mapping when an update occurs on the name or parent dentry address. This update process will apply to not only the file's dentry but also its all ancestor directories'. If the dentry is unchanged, Deptyque records the timestamp of its last update instead of the name string to perform fast queries.

4.3 Support For Multiple Build Tools

In this section, we will demonstrate the specialization efforts needed for different build tools. Table 4.1 shows a summary of the specializations.

Build Tool	**File Extension Rule**	**Call Stack Rule**	**Process Command Rule**
Maven & Gradle (Java)	Match .jar	JVM-class-loading probes	Match javac
Cargo (Rust)	Match .rmeta	-	-
Conan (C++)	Match .a	-	-
Go	Match .go	-	-
Make	-	-	-
Generic	-	-	-

Table 4.1: Specialized Rules for the supported build tools.
-: No specific rule is needed

4.3.1 Maven and Gradle (Java)

Maven and Gradle stand out as two prominent build tools utilizing the JVM and Javac for software builds, representing notable challenges among the supported build tools. The process of building a JAR artifact with Maven and Gradle involves several steps. Initially, these build tools resolve packages and versions, download the necessary dependency JAR files, and then pass the paths of these files as arguments to Javac. This encompasses both direct and transitive dependencies. Subsequently, Javac reads the source code and the downloaded JAR files to generate the corresponding class files. Finally, Maven and Gradle package these class files into the final artifact JAR. For simplicity, we consider only the original JAR artifact without runtime dependencies, as these are typically included in a Shaded JAR.

Following the methodology, we employ three annotation rules. The first rule identifies JAR files

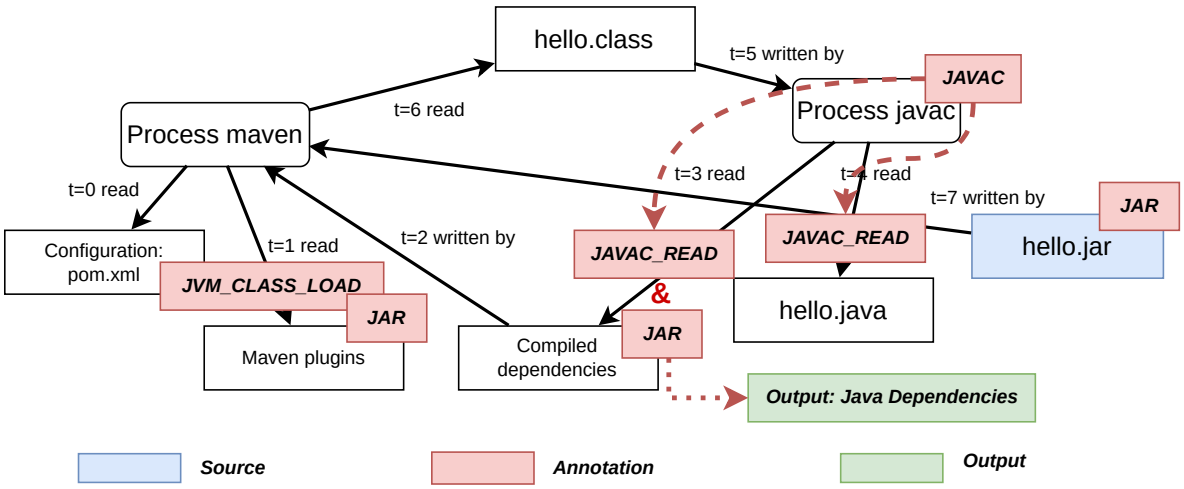

Figure 4.3: An annotation and output example of the simplified Maven build provenance.

based on their file extension, which is commonly denoted as "Java ARchive." The second rule marks any call stack that includes selected probes for JVM class loading. The third rule identifies the Javac process either by the command name or through the loading of Gradle's Javac Compiler Plugin classes.

By applying the three annotation rules, we gain the ability to identify and categorize various dependencies by traversing from the target artifact file. Build-time dependencies refer to JAR files that are reachable and read by the JVM's class loaders. On the other hand, compiled dependencies are those read during the execution of Javac, excluding the probes of JVM's class loaders. The use of negation in this context arises because libjvm is the finest granularity achievable with eBPF. If probing Java code becomes feasible, employing more specific probes within the call stack could replace the need for negation.

Figure 4.3 illustrates an example run of the annotation and output rules. While all vertices shown are reachable from the source vertex "hello.jar" because we have trimmed the graph, only those annotated with "JAVAC_READ" and "JAR" are Java Dependency outputs.

Additionally, initiating provenance processing from any Test Class of the target artifact allows Deptyque to analyze test dependencies following the same principles applied to compiled dependencies. This cohesive approach ensures consistency in the identification and classification of dependencies throughout the Javac-based build processes.

End users need to configure the forking of a Javac process. Forking the Javac process separates the execution before and during the Javac compilation, thereby excluding false positives related to files read before the compilation. Configuring this is straightforward for both Maven and Gradle users. Maven users need to set *maven-compiler-plugin.fork = true*, while Gradle users are required to set *JavaCompile.options.fork = true*.

4.3.2 Cargo (Rust)

In resolving Cargo's dependencies, Deptyque leverages the "rmeta" files—Rust metadata files containing package interfaces. Instead of reading source code, the Rust compiler, Rustc, relies on loading the rmeta files of dependency packages for type checking and compilation. As a result, marking and exporting these rmeta files prove to be adequate for identifying all compiled dependencies.

However, a notable limitation arises from the fact that the names of rmeta files typically include the package name and a hash without the package version. For example, *libsyn-afca3c8c7c39ef62.rmeta* is the rmeta file of *syn*. While Deptyque can extract the name *syn* by matching the prefix *lib*, it fails to extract package information like the package version, which we will discuss in section 4.5. Consequently, Deptyque presently can only resolve package names for Cargo, lacking specific version information.

4.3.3 Conan (C++)

Conan downloads packages with source codes, compiles them into static libraries with ".a" extension, and invokes ld to link them with the project's objective files, ultimately creating the artifact. Consequently, we need one rule to identify the ".a" extension. Similar to Cargo, the static library's name does not contain package version.

4.3.4 Go

To support Go, Deptyque annotates files with ".go" extension. Their full paths embed the package names and versions as directories systematically. Therefore, Deptyque supports both package names and versions for Go dependencies.

4.3.5 Make and Generic Build Tools

While there are no specific rules tailored for Make and other generic build tools, it's important to note that these systems often generate multiple intermediate files. Deptyque, naturally, has the capability to trace back to the origins of these intermediate files to identify transitive dependencies.

Some build tools utilize Make to separate build phases and invoke other build tools for components. For example, MozillaBuild for Firefox uses CMake and Make to define component dependencies and invokes Clang and Cargo for the compilation of C++ and Rust components respectively. When those components are consumed or packaged, Deptyque can trace the relationships between related files and processes transitively, so that it conducts a comprehensive dependency analysis. We will present more detail of the Firefox builds in section 5.7.

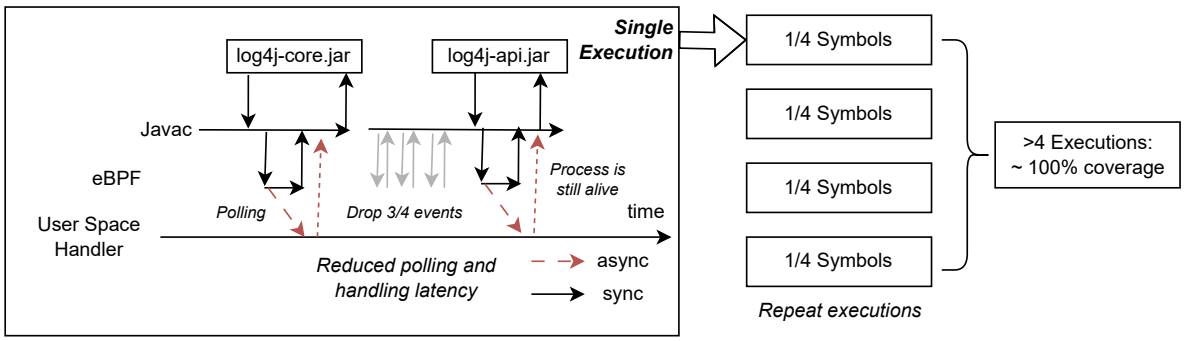

Figure 4.4: Example of Symbol Resolution.
- Set N=4. Repeat 4 executions while every execution resolves 1/4 symbols.
- Increase N to minimize the resolution latency until all symbols are resolved.

4.4 Offline Discovery: Symbol Resolution of Short-Lived Processes

In the Offline Discovery phase, our focus is on obtaining comprehensive call stacks for each event and extracting insights from these stacks. In the absence of prior knowledge, we leverage eBPF functionality to capture full call stacks and attempt to resolve symbols as thoroughly as possible. As depicted in Figure 3.4, achieving this in Online Tracing is unfeasible due to the latency of symbol resolution surpassing the lifecycle of short-lived build-system processes.

Our solution is illustrated in Figure 4.4. First, we run an example project multiple times, each time resolving a portion of symbols. This simple design significantly enhances the success rate of symbol resolution, theoretically covering all probe symbols as the number of repetitions increases.

The second task involves providing insights from the call stacks. We establish rules to categorize events into groups based on our knowledge of example projects; for instance, distinguishing read syscalls to JAR files for build-time dependencies from those for compiled dependencies. The challenge lies in discovering generic rules, essential probes, to cluster events based on call stacks. We sort probes by their occurrence and average distance to the event syscall.

Subsequently, through manual examination of the source code for each probe and assessing coverage when included, we identify key probes for JVM and libc. Build tool developers can employ similar approaches to support new build tools.

4.5 Implementation Limitation: Package Information Extraction

The main implementation limitation is the package information extraction. Deptyque can reliably produce a list of dependencies' file paths while the remaining task is to extract package information from the path. Our current solution is by extracting package names and versions from the path,

which works for some build tools like Maven, Gradle and Go embedding the information on the path as shown in Table 5.2. Still, there is no reliable solution to extract package metadata from Cargo's rmeta file paths and Conan's static library paths.

Alternatively, computing file checksums by accessing container files after build and querying a dependency database with these checksums is a generic solution for package information extraction. Vulnerability scanning and license tracking will also benefit from this type of dependency database.

4.6 Implementation Limitation: Network Provenance

Another limitation is the network provenance, which is the source (i.e. the URL) of each downloaded files. However, Deptyque cannot retrieve HTTP-layer information because of the SSL encryption. While the kernel uses the TCP-layer information including the IP address and port for packet routing, the HTTP-layer information is in the payload, which is encrypted and decrypted in user space for HTTPS connection, designed as the purpose of SSL.

One solution is instrumenting libraries like OpenSSL to retrieve the decrypted payload; nevertheless, build tools of different languages like Java and Go might have various implementations of SSL and this would end up with high complexity. Another solution used by the Kubernetes community is using self-issued certificates for decryption, while it breaks the certificate verification process, requires users' trust, and may limit Deptyque's potential use case as a public cloud service.

Chapter 5

Evaluation

This section aims to substantiate the system's efficacy by addressing the following research questions:

- RQ1: To what extent does Deptyque **correctly** identify all compiled and test dependencies within the supported build tools?
- RQ2: How much additional **latency** and **resource consumption** are introduced by Deptyque?
- RQ3: Does Deptyque demonstrate scalability when applied to **large projects**?

To evaluate Deptyque's performance, we conduct a thorough analysis on example projects from the supported build tools. Additionally, we present a case study involving a Micronaut application as a multi-module Maven project, and Firefox build, a representative example of a large-scale software system with a complex build structure.

5.1 Setup

We conducted the experiment on an OCI virtual machine equipped with a 16-core AMD EPYC 7J13 processor and 64GB of memory, running Oracle Linux Server 8.9 with Linux kernel version 5.15.0. It's worth noting that one can reproduce these experiments on any Linux server with eBPF installed and enabled.

Our study comprises three sets of experiments: small-scale, medium-scale and large-scale case studies. For the small-scale benchmarks, we utilized simple examples from Maven, Gradle, Go, Cargo, and Conan to compare results with state-of-the-art dependency analysis tools. Moving to the medium-scale experiment, we employed a Micronaut example to illustrate the impact of package

downloading on provenance complexity. Additionally, we designed a large-scale example of the Firefox build. In these instances, we not only compared dependency coverage where available but also showcased the visualization and reasoning of their results to highlight the potential capabilities of Deptyque.

Table 5.1 presents the complete set of experiments:

Name	**IDs**	**Type**	**Build Tool**	**Context**
maven-helloworld	1-2	small	Maven (Java)	Maven "helloworld" with Log4j.
gradle-helloworld	3-4	small	Gradle (Java)	Gradle "helloworld" with Log4j.
cargo-helloworld (*)	5-6	small	Cargo (Rust)	Cargo "helloworld" with clap.
conan-helloworld (*)	7-8	small	Conan (C++)	Conan "helloworld" with zlib.
go-helloworld	9-10	small	Go	Go "helloworld" with go-cmp.
go-gin	11-12	small	Go	Go Gin web application.
micronaut	13-16	medium	Maven (Java)	Micronaut web application.
firefox-1 (-)	17-18	large	Mixed	Firefox build with artifact mode.
firefox-2 (*)	19-20	large	Mixed	Firefox build with source mode.

Table 5.1: Table of example projects.
(*): Projects whose SBOM is compared without package versions.
(-): Projects which we do not measure dependency coverage.
Experiment 1-6 and 9-14: The cache directories are mounted.

To address RQ1, we generated SBOM for each experiment and compared it against results from state-of-the-art tools. It's important to note that, unless specified otherwise, we define a match as agreement on both package name and version. Coverage is assessed through precision and recall, without considering the dependency hierarchy in this context.

For RQ2, we measured the time and resource consumption of different phases for each experiment. We provided a breakdown of time spent on each phase. More specifically, we record phase changes to a log file and measure the end-to-end latency of different phases with the granularity of one second. Additionally, we also monitor the memory and CPU consumption per second using vmstat and aggregate them by phases. Due to eBPF storage's nature as kernel space memory shared by both the kernel and user space processes, we monitored the overall memory consumption of the server.

In response to RQ3, we conducted three experiments involving Micronaut, and Firefox artifact build and Firefox source build. The Micronaut application served as an example of a multi-module Maven application, while the Firefox project encompassed multiple build tools, including C++, Rust, and JavaScript source codes. Leveraging the support for C++ and Rust, we specified multiple targets and explored dependencies, highlighting the lack of comprehensive understanding provided by existing tools in this context.

5.2 Software Dependency Coverage

Name	Provenance	Build	Compiled	Test	Pkg Name	Pkg Version
Maven	Yes	Yes	Yes (100%, 2)	Yes (100%, 2)	Yes	Yes
Gradle	Yes	Yes	Yes (100%, 8)	Yes (100%, 2)	Yes	Yes
Cargo	Yes	-	Yes (100%, 16)	-	Yes	No
Go	Yes	-	Yes (100%, 17)	-	Yes	Yes
Conan	Yes	-	Yes (100%, 1)	-	Yes	No
Make	Yes	-	-	-	-	-
Micronaut	Yes	Yes	Yes (100%, 30)	Yes (100%, 19)	Yes	Yes
Firefox	Yes	-	Partial (>92.5%, 62 and 130)	-	Yes	No

Table 5.2: Table of dependency coverage.
- Yes (100%): precision and recall.
- No: Lack of support.
- Partial: Support with lower than 100% precision or recall.
- The number after percentage is the count of dependencies in our benchmarks.

Table 5.2 details the dependency coverage provided by Deptyque using example projects. In general, Deptyque generates comprehensive build provenance for all supported build tools, ensuring coverage of all compiled dependencies. Additionally, we offer partial support for the Firefox build with 92.5% precision and 99.2% recall, a topic that will be further elaborated upon in section 5.7.

As explained in chapter 4, Deptyque can extract package names and versions from the paths of Maven, Gradle, Go, and the Micronaut example, thereby producing complete identifiers for dependency packages. However, it's noteworthy that for Cargo and Conan, we have not figured out an effective extraction of version from the file path yet.

In summary, the dependency coverage of Deptyque for small and medium scale examples has reached 100% precision and recall, while we need further specialization for the Firefox builds.

5.3 Build Provenance Examples of Supported Build Tools

In this section, we illustrate the build provenances directly generated by Deptyque or simplified manually to demonstrate the prowess of Deptyque in understanding the build process comprehensively. Furthermore, in section 5.7, we will present a case study on Firefox builds.

Deptyque produces two types of build provenance: a trimmed provenance and a full provenance. The trimmed provenance only includes edges of all possible paths from the source vertex to the vertices of compiled dependencies, which focuses on understanding the usage of compiled

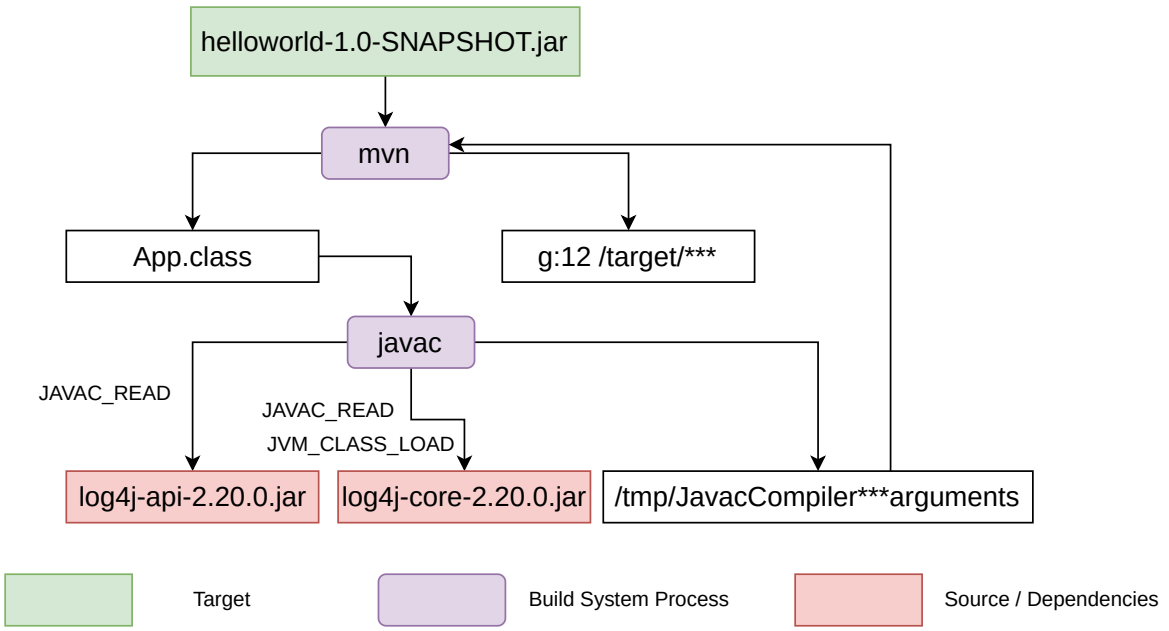

Figure 5.1: Trimmed provenance of Maven "helloworld" project.

dependencies. In contrast, a full provenance includes all reachable edges and vertices. We further group vertices with the same input and output relationships, meaning they are read and written by the same processes. For Maven, Gradle, Go and Cargo, these build tools have a relatively large number of build-time dependencies, so we will demonstrate their trimmed provenance. For Conan, we will present its full provenance since it is relatively simpler.

5.3.1 Build Provenance of Maven

Figure 5.1 is the trimmed provenance of Maven "helloworld" project which has two compiled dependencies *log4j-api* and *log4j-core*. Tracing from the source vertex which is the final artifact *helloworld-1.0-SNAPSHOT.jar*, it is written by Maven process, which packages the class file *App.class* and file group 12 including *log4j2.xml* and *pom.properties*. We can further figure out that *javac* reads both dependencies and produces *App.class*. Interestingly, *log4j-core* is also loaded by *javac* class loader, inferring that it is both compiled and build-time dependency.

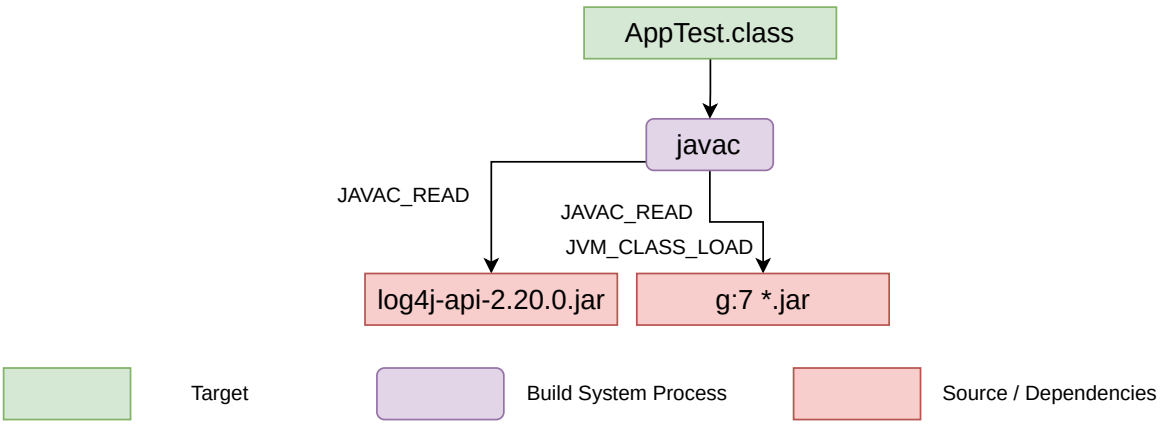

Figure 5.2: Trimmed provenance of Maven "helloworld" project's test dependencies.

By traversing the provenance graph from a test class of Maven, Deptyque can analyze the compiled and test dependencies as shown in Figure 5.2. The methodology is the same as traversing from the artifact JAR file, except that in this case, the provenance processing skips the Maven packaging phase. In the result, group 7 includes three dependencies for testing: *log4j-api*, *junit*, and *hamcrest-core*.

5.3.2 Build Provenance of Gradle

Figure 5.3 shows the build provenance of Gradle, which differs from Maven in the build tool's architecture. The process */127.0.0.1:3436* is the Gradle build tool server, which reads the *App.class* for generating *app.jar*. Meanwhile, it reads build-time dependencies in file group 14, 15, 16, and 17. The group 14 includes *registry.bin* and *registry.bin.lock*, which are generated and used by *gradlew*, the initialization process of Gradle. Tracing from *App.class*, the process */127.0.0.1:3282* is the Javac compilation client process consuming compiled dependencies which reside in Gradle's local cache. For Gradle, we can recognize the Javac process because its class loader dynamically loads the Gradle compiler plugin JAR, which is a build-time dependency not included in this trimmed provenance.

5.3.3 Build Provenance of Go

The build process of Go is more complicated than Maven and Gradle as shown in Figure 5.4. The *go* process takes charge of package downloading and compiling object files with go source codes. Several *asm* and *compile* processes created by the *go* build tool handle the assembling and compiling respectively. At the final stage, a *link* process links the object files and shared libraries into *a.out*, which will be renamed into the artifact's name *go-cmp*.

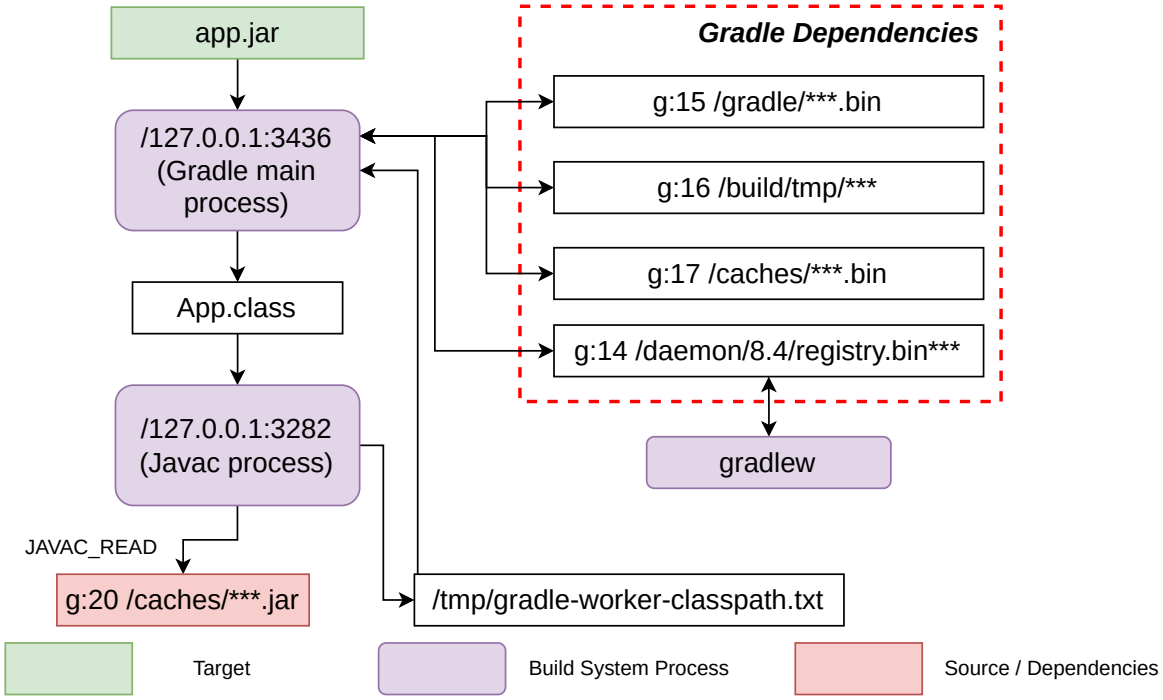

Figure 5.3: Trimmed provenance of Gradle "helloworld" project.

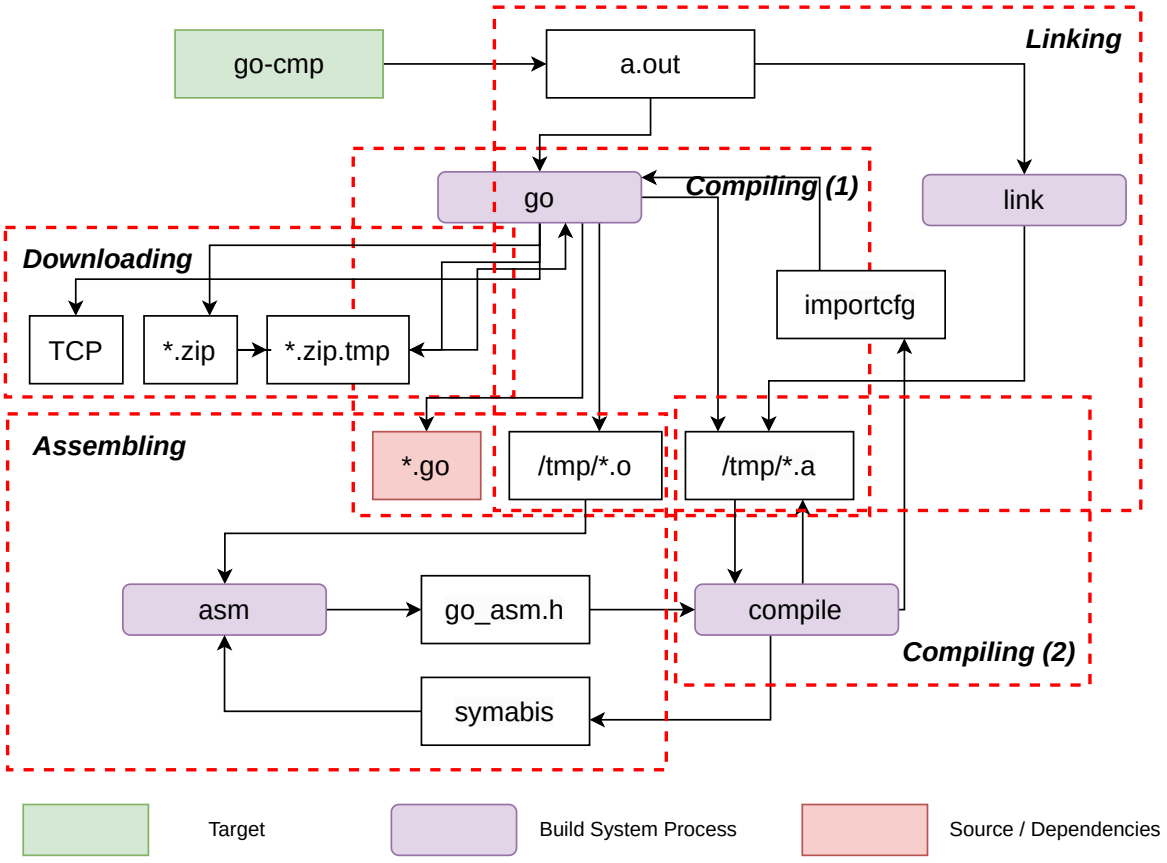

Figure 5.4: Trimmed provenance of Go "helloworld" project.

5.3.4 Build Provenance of Conan

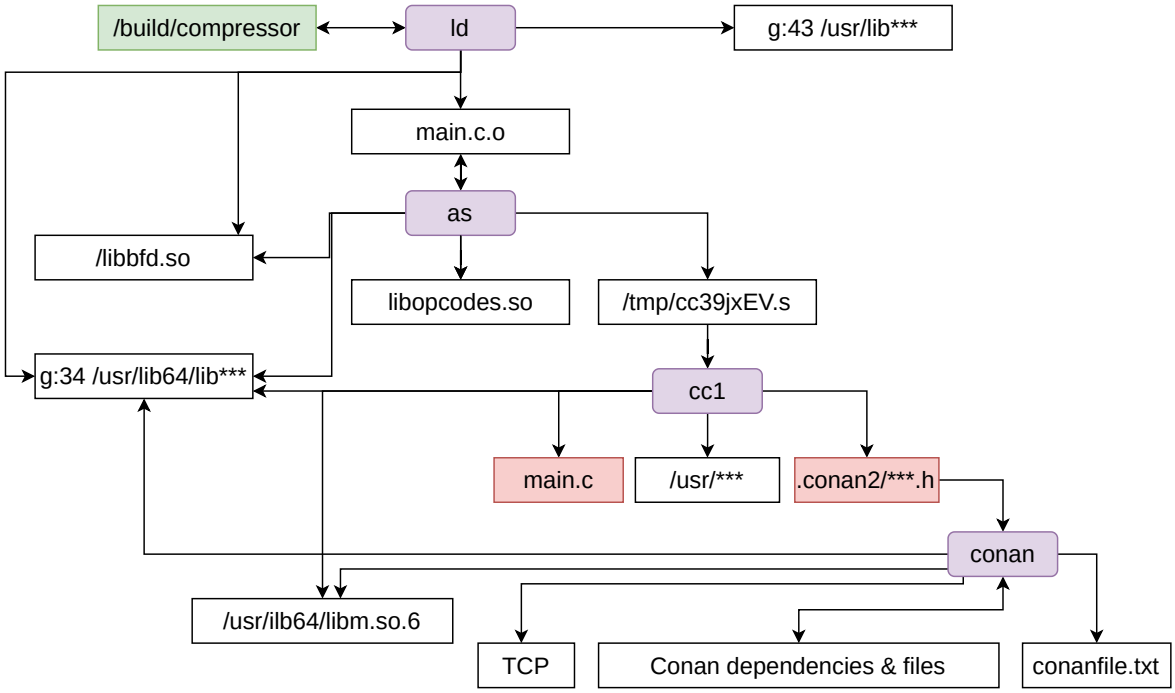

Figure 5.5: Full provenance of Conan "helloworld" project.

For Conan, its full provenance is more interesting to discover as shown in Figure 5.5. Conan separates build phases with different processes which an user can clearly identify from the graph. It also demonstrates the functionality of Deptyque to trace temporary files like *tmp/cc39jxEV.s* which is removed during the build process and is difficult to identify through post-scanning of the container. The *TCP* vertex shows that Conan reads from a TCP socket to download compiled dependencies, while we have discussed the limitation of eBPF-based network provenance in section 4.6.

5.4 Latency Analysis

As depicted in Figure 5.6, Figure 5.7, and Figure 5.8, the additional latency in Deptyque can be attributed to three key components: the eBPF program's impact during Online Tracing, the eBPF load time, and the Python scripts employed for Post Processing.

The end-to-end eBPF Online Tracing overhead is no more than **12.8%** by comparing the time of build process with and without eBPF instrumentation. The increased latency arises from the eBPF programs engaged in identifying build container processes, traversing kernel data structures, managing the call stack, utilizing internal eBPF storage, and submitting data to the Ringbuf ring

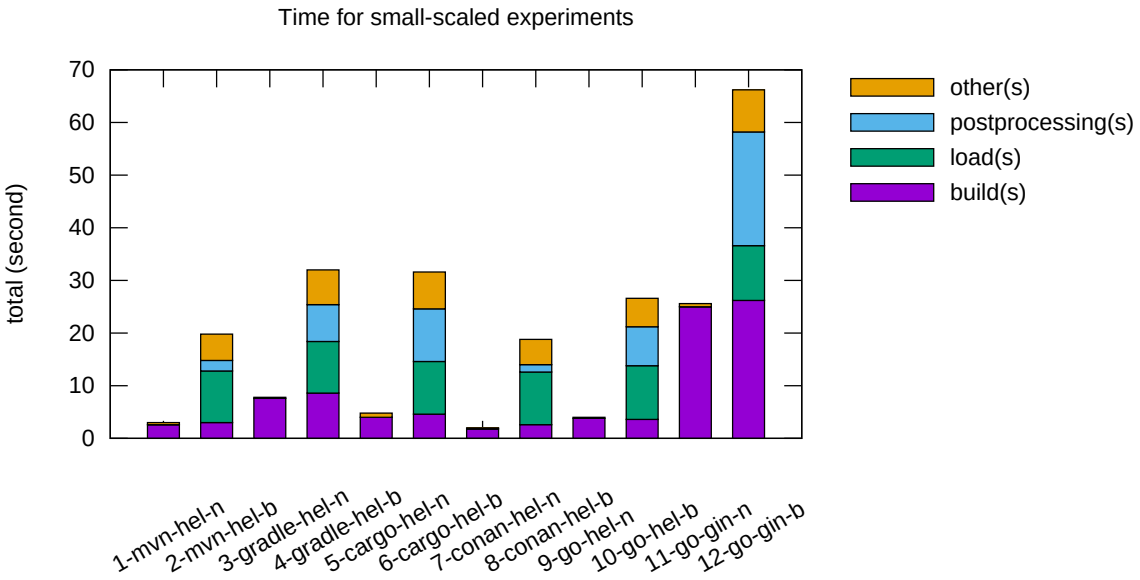

Figure 5.6: Latency of small-scaled examples.
- mvn: Maven; hel: helloworld;
- b: with eBPF; n: without eBPF.

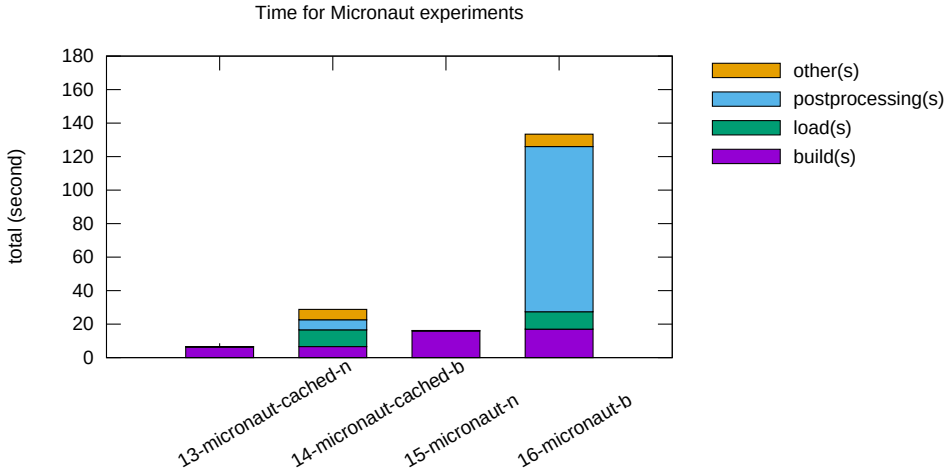

Figure 5.7: Latency of Micronaut builds (for section 5.6).
- b: with eBPF; n: without eBPF.
- Experiment 13 and 14 are builds with packages cached.
- Experiment 15 and 16 are builds without packages cached.

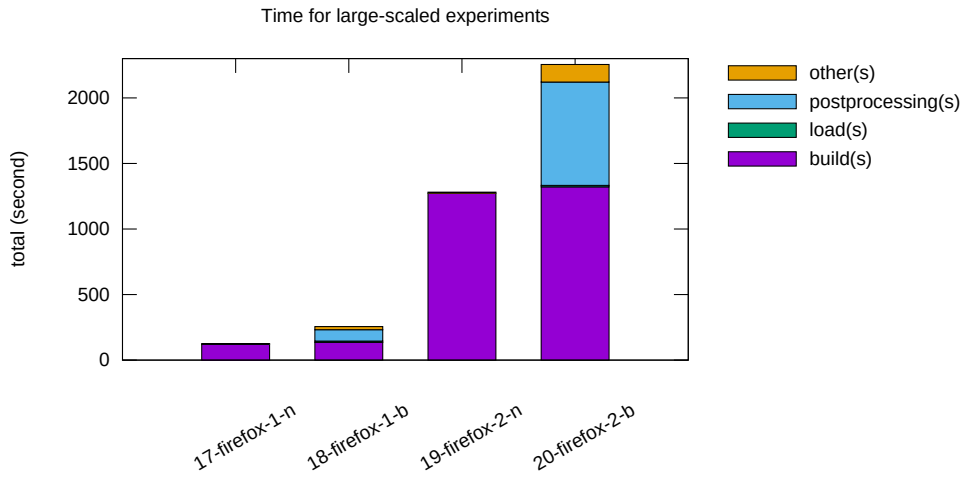

Figure 5.8: Latency of Firefox builds (for section 5.7).
- b: with eBPF; n: without eBPF.
- Experiment 15 and 16 are Firefox Artifact Builds.
- Experiment 17 and 18 are Firefox Source Builds.

buffer.

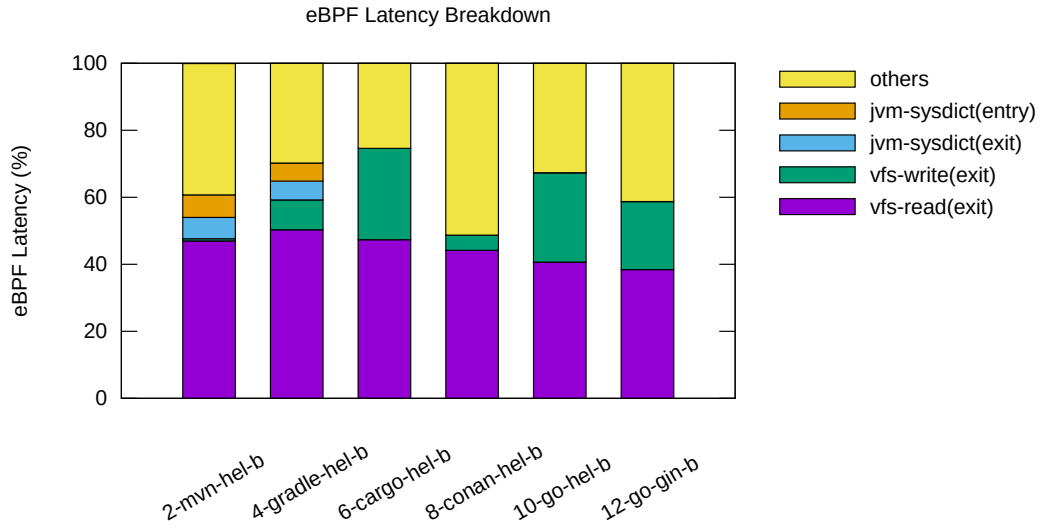

Figure 5.9: Latency of eBPF Programs in small-scaled examples.
- **vfs-read** and **vfs-write**: probes that submit event data.
- **jvm-sysdict**: **SystemDictionary::load_instance_class_impl(Symbol*, Handle, JavaThread*)**, a JVM user-space probe used in JVM class loading. Its entry and exit probes cooperate to maintain the call stack.

Furthermore, we use the kernel's BPF statistics functionality and bpftool [26] to collect eBPF latencies. Figure 5.9 demonstrates the latency of eBPF functions during small-scale experiments. In most experiments, **vfs_read** probe dominates the eBPF latency, contributing to between **38%** to **50%** of the total eBPF latency, because **vfs_read** is not only frequent, but also time-consuming that it submits the data to user space through ring buffers. The contribution of **vfs_write** and **jvm_sysdict** depends on the build tool pattern. For JVM-based build tools like Maven and Gradle, they utilize JVM class loaders to load classes and most data exchange is in JVM memory. Cargo and Go rely more on intermediate files for the communication between different build phases. Their compilers create multiple object files for multi-process compilation and the linker composes the final artifact using these temporary files.

The average latency of **vfs_read** and **vfs_write** is between 2000 and 3000 ns, mainly on kernel data structure traversal, call stack copying, and data transfer to ring buffers. On the other hand, the average latency of entry and exit probes for maintaining the call stack is between 200 and 500 ns.

The current implementation of Deptyque only instruments essential probes for our supporting build tools. While more probes could provide fine-grained event information for more comprehensive understanding of events, they will also incur additional latency. The current result of Online Tracing latency underscores the suitability of eBPF as a promising solution for Linux monitoring and event tracing.

The eBPF load time stands at approximately 10 seconds, representing a static latency for compiling and loading the eBPF program. This effect diminishes as the software scale increases. In the current Deptyque implementation, user space symbol addresses are resolved, and the eBPF program is recompiled for each instance. Although not yet supported by the BCC framework, reusing eBPF bytecode has the potential to reduce the eBPF load time.

Despite these considerations, Post Processing incurs a relatively high latency without further optimizations. Our Python-based implementation of Post Processing allows for further parallelization of most operations to enhance speed. Moreover, there is potential for further improvement through specialized rules for provenance processing, aimed at reducing graph complexity, including the number of cycles and edges.

5.5 Resource Consumption

To assess resource consumption, we employ long-duration Firefox builds as representative samples. Our focus here is on evaluating CPU and memory utilization during Online Tracing.

Figure 5.10 illustrates the average CPU usage for both Firefox Artifact and Source Builds. Firefox Artifact Builds exhibit low CPU usage due to their I/O intensity, predominantly spent on file downloads. The user CPU usage experiences a slight increase from 7.31% to 7.52%, attributed to data

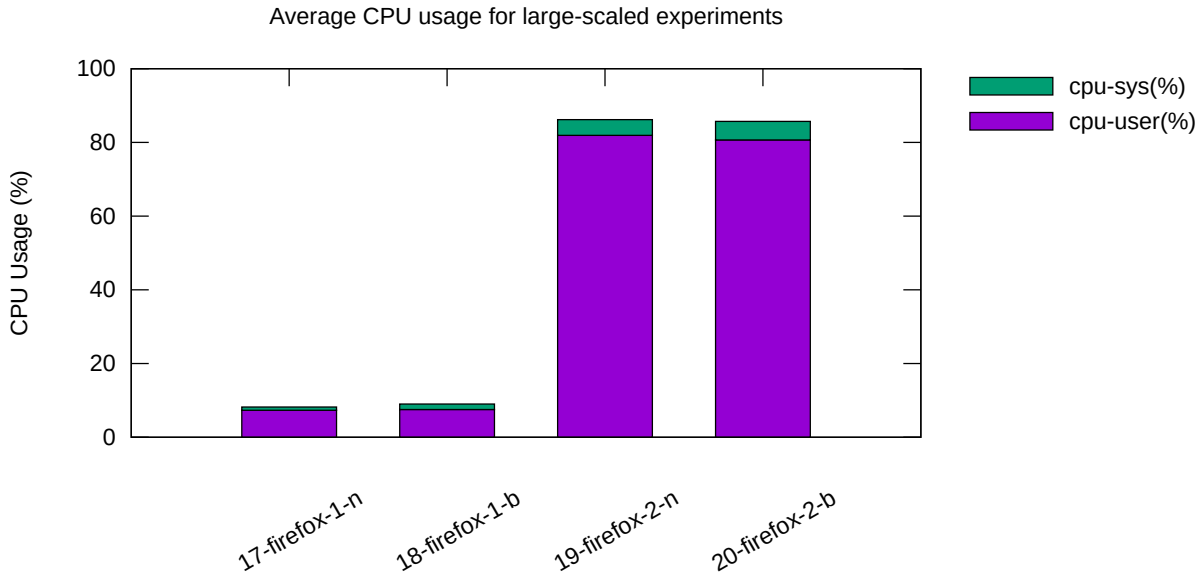

Figure 5.10: CPU usage of Firefox builds.
b: with eBPF; n: without eBPF.

polling from eBPF to user space. Simultaneously, the system CPU usage sees a rise of 0.6%, from 0.89% to 1.49%, reflecting the overhead of eBPF code in the kernel space.

In the case of Source Builds, the primary overhead is associated with parallel compilation, resulting in significantly higher CPU usage compared to Artifact Builds. The user CPU usage marginally decreases from 81.91% to 80.48%, a reasonable outcome considering that enabling eBPF also incurs approximately a 10% increase in build time. The system CPU usage slightly rises by 0.8%, from 4.27% to 5.07%, aligning with the disparity observed in the Artifact Build.

Turning to memory usage, Figure 5.11 depicts the variance in average memory usage. For both builds, the difference is around 3.71 GiB, as Deptyque reserves this storage memory for the ring buffer and dentry state cache. It's noteworthy that inadequate memory allocation can lead to data loss, and eBPF does not support dynamic allocation of kernel memory. Considering the server's total memory of 64 GiB, reserving 3.71 GiB accounts for only 5.8%, also allowing users to modify the size for different server scales.

In summary, eBPF demonstrates minimal impact on user CPU, with less than 0.8% overhead on system CPU. The memory reserved by eBPF is configurable for various builds and server scales. For supporting Firefox builds, allocating around 3.71 GiB, equivalent to 5.8% of the server's total memory, proves sufficient.

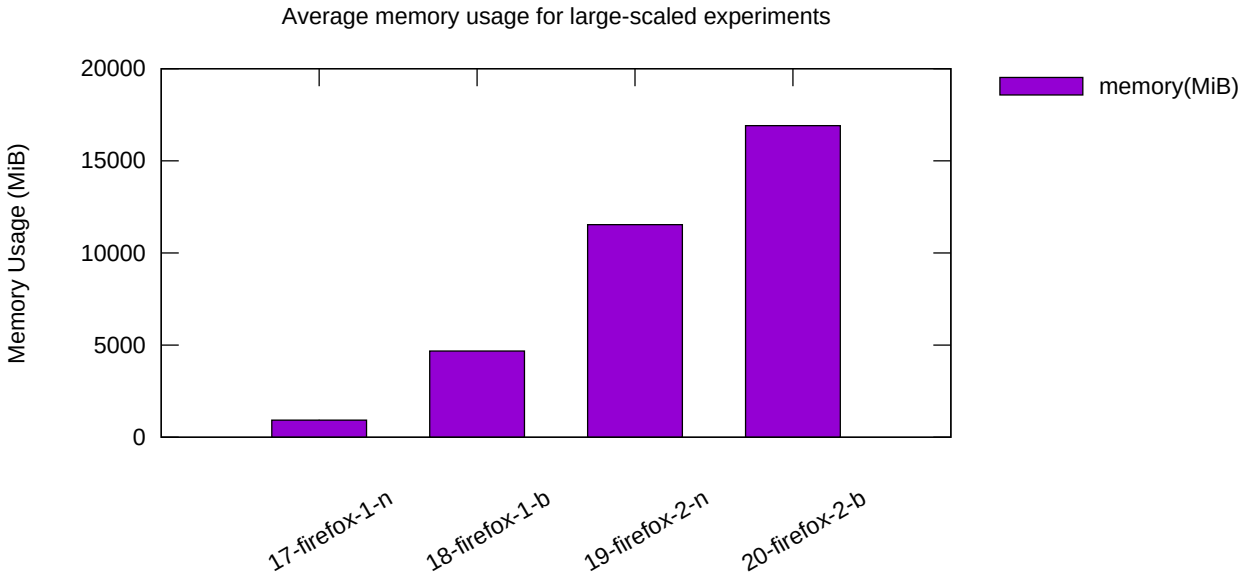

Figure 5.11: Memory usage of Firefox builds.
b: with eBPF; n: without eBPF.

5.6 Increased Complexity Due to Package Downloading

Figure 5.7 illustrates the impact of package downloading on a Maven Micronaut application, with the additional complexity arising from additional connectivity in the provenance graph. While the build time experiences only a **157%** increase, rising from 7.00 to 18.0 seconds, the Post Processing time sees a more substantial **1,714%** increase, escalating from 7.2 to 127.0 seconds. Although the number of provenance records aligns roughly with the build time, the results underscore the significant impact of package downloading on processing complexity.

During package downloads, we introduce provenance tracing from the downloaded JAR to the Maven process that initiates the download as shown in Figure 5.12. The Maven process concurrently downloads multiple files and read the pom files, leading to additional cycles and an increase in graph complexity compared to the case with caches. Implementing solutions such as mounting a package cache or incorporating specialized rules to halt graph traversal can effectively address and mitigate this issue.

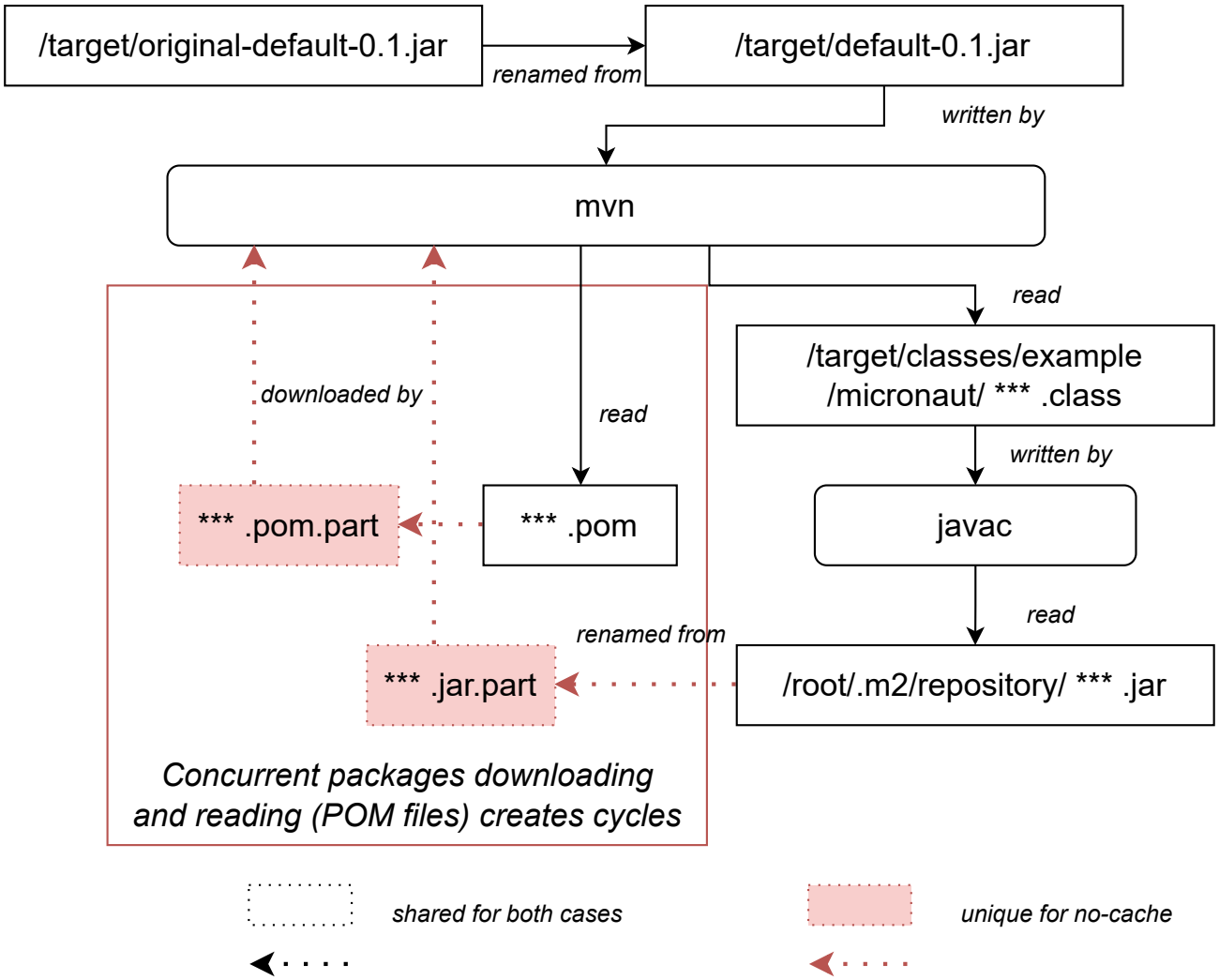

Figure 5.12: Simplified build provenance of Micronaut builds.
The red part shows the additional concurrent downloading of multiple files.

5.7 Case Study: Firefox

In this section, we employ the Firefox browser as a comprehensive example to showcase the performance capabilities of Deptyque. Firefox utilizes a mach Python script to initiate the build dependencies and subsequently invokes the Mozbuild build system. Internally, this build tool utilizes gmake, cargo, clang, and node.js to compile various components. The repository supports the construction of both desktop and mobile versions. Notably, Firefox does not currently provide an SBOM due to its intricate nature.

A Firefox build yields multiple binaries intended for distribution. In our experimentation, we narrow our focus to three specific targets: the firefox executable, the geckodriver executable, and the libgeckoservo library. The firefox executable serves as the primary entry point for the Firefox browser, coded in C++, which dynamically loads other shared libraries at runtime. It provides an illustrative example of the build provenance for a complex C++ executable. The geckodriver, written in Rust, represents an example of a Rust executable. Finally, the libgeckoservo, a Rust library designed for Gecko and Servo, exemplifies a Cargo library.

Our experiments examine both the artifact mode, which involves downloading pre-built C++ components instead of local builds, and the source mode, which entails compiling every component from its source code. Through our exploration, we aim to showcase the build observability achieved with Deptyque.

5.7.1 Case Study 1: Firefox (Artifact Mode)

The Firefox artifact build expedites the build process by downloading pre-compiled components and building from them, foregoing the need of compilation from source. The corresponding additional overhead is illustrated in Figure 5.8. Notably, the Online Tracing overhead is approximately 10.6%, while the contribution of Post Processing diminishes to 38.6%.

Figure 5.13 describes the build provenance in the artifact mode. Our observation reveals that the Python process writes the three binaries after downloading and processing of an archive aligning with the expected build description. Consequently, we infer that the archive */root/.mozbuild/package-frontend/[HASH]-target.tar.bz2* is the dependency of the three targets. Notably, such dependencies are challenging to discern through static analysis of source code and configuration files. Furthermore, we can calculate the checksum of the archives to investigate their metadata and validate their origin.

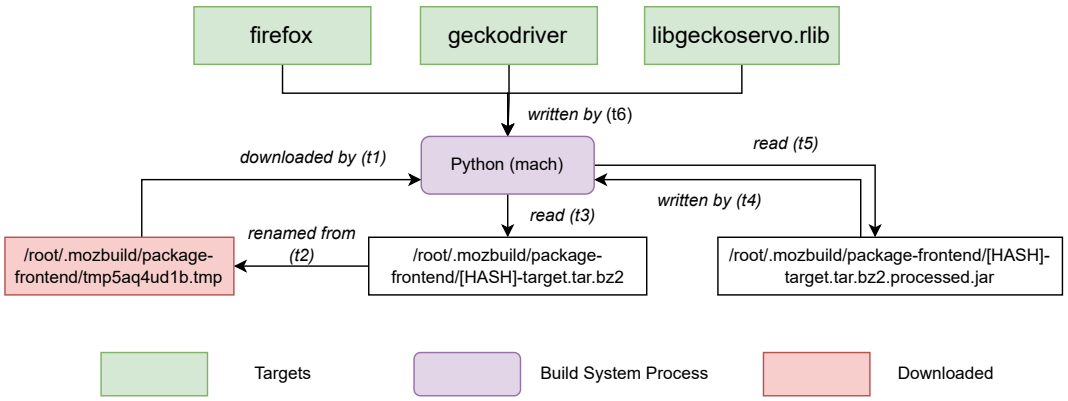

Figure 5.13: Simplified build provenance of Firefox Artifact Build.
- t1: Download and cache to a tmp file.
- t2: The tmp file is renamed to a tar.bz2 file upon the completion of downloading.
- t3 and t4: Python reads the tar.bz2, processes it, and writes the result to processed.jar file.
- t5 and t6: Python reads the processed.jar file and extract the three binaries.

5.7.2 Case Study 2: Firefox (Source Mode)

Firefox Executable

By configuring the analysis target to the Firefox executable, Deptyque efficiently generates comprehensive build provenance, depicted in Figure 5.14. This representation elucidates every step of the build process, a feat challenging to achieve through static analysis alone.

The compilation process employs clang, clang++, and ld.ldd as the compilers and linker. The clang and clang++ instances compile the source code of components, along with shared headers, into object files. Subsequently, ld.ldd links these object files to generate the Firefox executable. Our build provenance unveils intricate details, including temporary files that are removed in the directory post-build. Notably, the build process leverages compiler toolchains managed by mozbuild, the Firefox build system, deviating from the default ones installed on the container base image.

In the figure, the blue boxes represent object files corresponding to C/C++ components. Consequently, Deptyque adeptly discloses locally compiled components and their relationships. While this build provenance operates seamlessly without specialized rules, users can also define custom rules to extract specific, valuable information.

Rust Executable And Library: Geckodriver and libgeckoservo

In addition to C and C++, Firefox also includes components in Rust managed by Cargo. The Cargo configurations are intricate, involving a combination of both remote and local libraries. In this

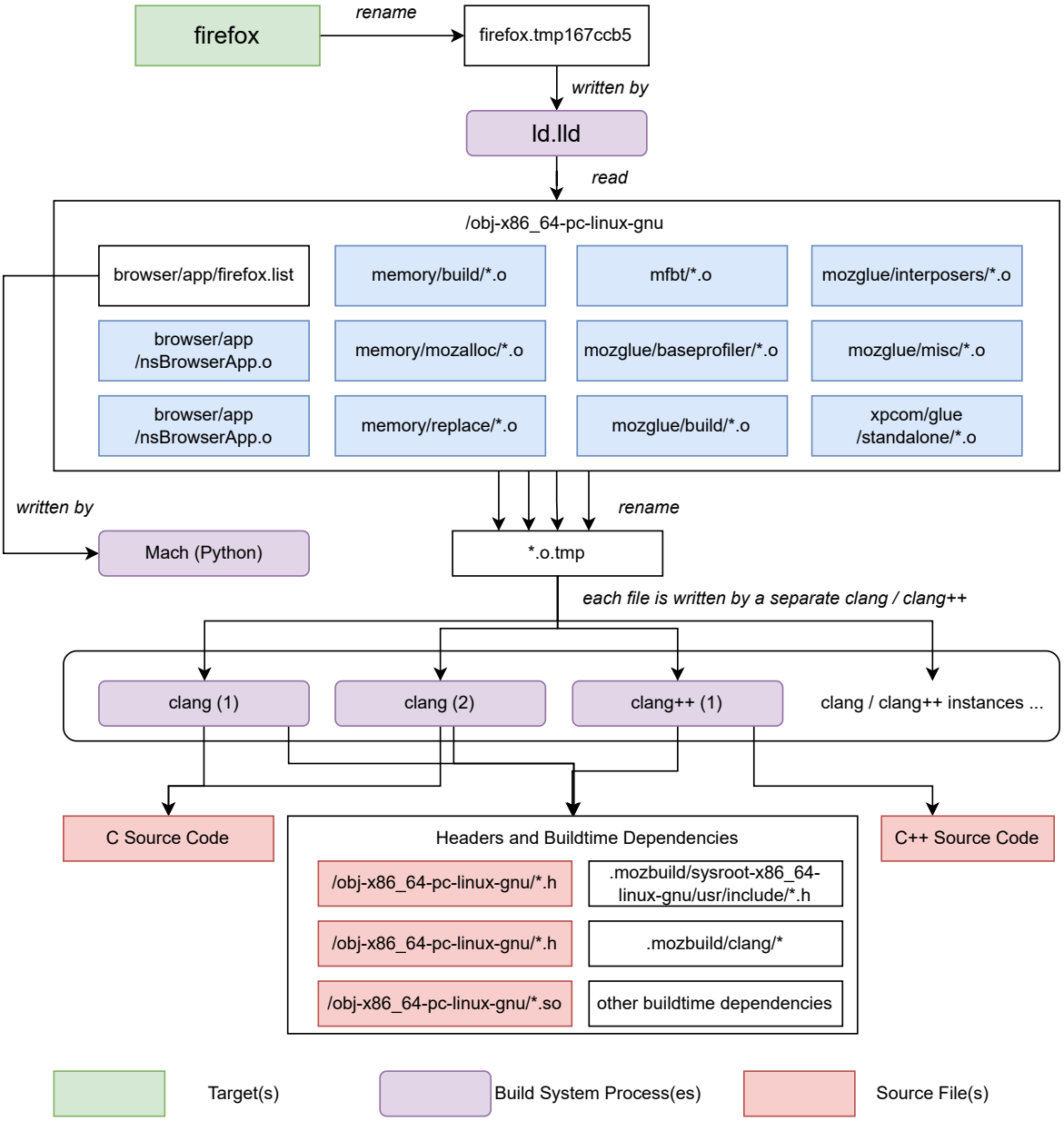

Figure 5.14: Simplified build provenance of Firefox Source Build.

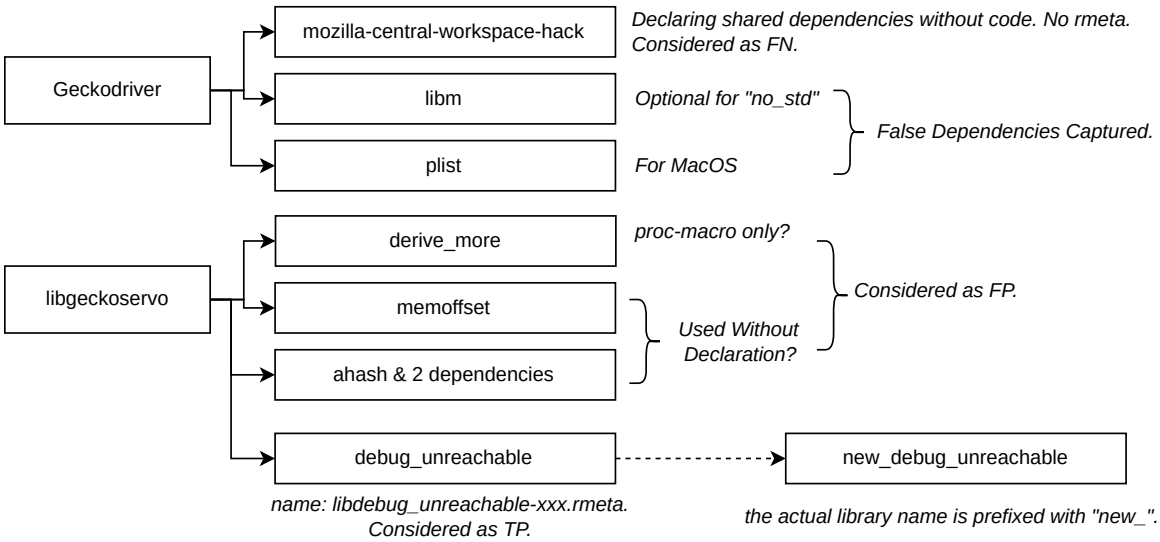

Figure 5.15: Cases for discussion in Firefox Cargo dependencies.

section, we employ Geckodriver and libgeckoservo as illustrative examples to assess the performance of Deptyque within a large-scale Cargo project in a real-world context. For our evaluation, we reference the CycloneDX Rust plugin, which internally leverages fine-grained information from Cargo.

Name	True Positives	False Positives	False Negatives	Precision	Recall
Geckodriver	129	0	1	100%	99.2%
libgeckoservo	62	5	0	92.5%	100%

Table 5.3: Dependency coverage of Cargo components in Firefox.

Table 5.3 displays the dependency coverage for Geckodriver and libgeckoservo. In the case of Geckodriver, Deptyque achieves a precision of 100% and a recall of 99.2%. As shown in Figure 5.15, the sole false negative is "mozilla-central-workspace-hack" based on "rustc-workspace-hack" which declares shared package dependencies for multiple modules in the same workspace. Sharing the same package versions reduce the number of packages and accelerate compilation. However, its source code contains no implementation; consequently, it has no "rmeta" file to be read for its dependent modules, which Deptyque has captured. While it is negotiable for different use cases, we conservatively categorize it as a false positive.

Moreover, we unearthed two challenging false dependencies that prove intricate to eliminate without delving into the source code. Firstly, "libm" serves as an optional dependency for "num-traits" exclusively when it is not part of an "std" build. Secondly, "plist" serves as a dependency for "mozrunner" only when targeting MacOS. Cargo's inherent dependency analysis, by default,

includes both of these dependencies along with their transitive dependencies. In contrast, Deptyque demonstrates its efficacy by excluding these false dependencies, underscoring its capacity to provide observability and accountability in a complex and dynamic build environment.

In the case of libgeckoservo, Deptyque attains a precision of 92.5% and a perfect recall of 100%. As depicted in Figure 5.15, the false positives identified encompass "derive_more," "memoffset," "ahash," and two dependencies of "ahash," the rmeta files of which are read by rustc. According to Cargo's built-in analysis, "derive_more" qualifies as a "proc-macro" (build-time) dependency. Additionally, "memoffset" and "ahash" stand out as two widely-used Cargo libraries that are not covered by Cargo's analysis. Given the intricacies of Firefox's structure, we cannot definitively confirm their role in compilation, leading us to classify them as false positives. Furthermore, there is a single name mismatch for "new_debug_unreachable," even though its file name is "libdebug_unreachable-[HASH].rmeta." Such irregularities are rare, and the build provenance successfully captures this dependency, prompting us to consider it a valid match.

To recap, Deptyque demonstrates an impressive precision exceeding 92.5% and a recall of 99.2% when analyzing large-scale Rust components within Firefox. Delving into the Firefox source code provides a means to elucidate the reasons behind false positives and negatives, enabling the refinement of processing rules for greater precision and recall.

Chapter 6

Related Work

6.1 Universal SBOM Generation

Numerous tools are available for generating universal SBOMs, catering to diverse build tools. The primary approach involves thorough filesystem scanning, the invocation of specific tools, and the consolidation of results.

One such tool is Syft, designed for generating SBOMs for container images and filesystems [3]. Syft conducts a comprehensive scan of the entire image or filesystem and employs a cataloger—an adept file parser for dependency configurations and package lock files—to scrutinize dependencies. The outcome is an aggregated list of dependencies constituting the SBOM for a given image or filesystem.

Another versatile universal SBOM generation tool is Cdxgen, which employs a strategy involving multiple build-tool-specific tools [17]. Similar to Syft, Cdxgen scans entire directories, invokes various build-tool-specific tools, and combines the identified dependencies.

While these tools leverage build-tool-specific dependency analyzes to generate SBOMs for diverse build-tools, it's important to note their limitations. Firstly, they offer the granularity of the whole directory. When the directory includes dependencies of multiple artifacts, these tools fail to identify dependencies of a specific artifact which will result in high false positives. Secondly, they may still struggle with intricate build tools, such as Make, characterized by dynamic computations beyond the capabilities of static parsers. These tools still lack the capability in capturing the intricate relationships and interactions among components across different build tools.

6.2 Alternatives of Operating System Event Tracing

Operating system (OS) event tracing is a highly effective method for generating OS-level provenance, capturing crucial details like timestamps, involved processes and files, activity specifics, and supplementary information at various granularity.

6.2.1 Linux Security Module

The Linux Security Module (LSM) serves as a versatile framework, utilizing LSM hooks that execute handlers in the kernel space to implement security features. Provenance collection based on LSM entails gathering data from these hooks with minimal overhead.

CamFlow [38] is an LSM to capture whole-world data provenance for the purpose of system audit, which users can configure for different metadata granularity. It accesses kernel objects and data using LSM and NetFilter hooks. Then, it generates extended PROV-DM format data for runtime security analysis [39].

One limitation of LSM is its instrumentation flexibility, constrained to a predefined set rather than arbitrary kprobes and uprobes. Extracting information beyond these hooks, especially for user space data structures, can be challenging. Furthermore, compiling and loading LSM is heavier than loading eBPF programs, which does not need the recompilation of Linux kernel.

6.2.2 Intel PIN Framework

The Intel PIN framework facilitates dynamic binary instrumentation (DBI) on IA-32 and x86-64 architectures, allowing the dynamic injection of code at runtime into compiled binary files. This capability proves valuable for tasks such as profiling, performance analysis, and security research.

DataTracker [43] leverages PIN to produce data provenance for individual binary applications, utilizing dynamic taint analysis (DTA) at the byte level for a thorough examination. This tool can instrument a diverse range of well-known applications without access to their source code. By delving into the inner workings of the executing program, DataTracker captures high-fidelity and precise data provenance.

However, a notable constraint of Intel PIN is its reliance on understanding the binary code for developing custom instrumentation tools. This presents a challenge when attempting to instrument an operating system kernel due to its complicated memory management and logic.

6.2.3 Intel Processor Trace (PT)

Intel Processor Trace (PT) stands out as a distinctive feature of Intel processors, offering precise and efficient tracing capabilities for program execution flow. Through its software-hardware co-design, PT captures a sequence of instructions executed by a processor, empowering developers with a tool to analyze and comprehend application behavior with high accuracy.

JPortal [48] emerges as a profiling tool that adeptly bridges the gap between the low-level hardware traces obtained from PT and the higher-level control flow within the Java Virtual Machine (JVM). Utilizing postmortem static analysis, JPortal efficiently scrutinizes the low-level traces, enabling the tracking of dynamic control flow in Java bytecode with minimal runtime overhead.

While PT proves to be a potent tool for analyzing the execution flow of user-space programs, its limitations include challenges in accessing kernel space seamlessly. Additionally, PT exhibits constrained portability to other hardware architectures, limiting its applicability beyond the Intel processor domain.

Chapter 7

Conclusion

In conclusion, we introduce the Deptyque framework designed to generate build provenance using eBPF and process this provenance for comprehensive software dependency analysis. Deptyque offers support for identifying compiled dependencies across various build tools, including Maven, Gradle, Go, Cargo, Conan, and generic Make builds. Notably, Deptyque stands out as the pioneering eBPF-based build provenance generator and the first versatile software dependency analyzer accomplished through real-time monitoring of the build process.

Deptyque is remarkable for its high analysis precision, minimal Online Tracing overhead, and enhanced observability that meticulously accounts for each step within the build process. Our evaluation encompasses small-scale benchmarks, a medium-scaled Micronaut application, and large-scale Firefox builds to validate the performance of Deptyque. The outcomes affirm that eBPF emerges as a promising technology for provenance and generic software dependency analysis, showcasing that achieving these goals with low overhead and specialization efforts is indeed feasible.

Bibliography

[1] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. "Solar winds hack: In-depth analysis and countermeasures". In: *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE. 2021, pp. 1–7.

[2] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. "Techniques for efficiently querying scientific workflow provenance graphs." In: *EDBT*. Vol. 10. 2010. 2010, pp. 287–298.

[3] Anchore. *Anchore/syft: CLI Tool and library for generating a software bill of materials from container images and filesystems*. URL: https://github.com/anchore/syft.

[4] Iosif Arvanitis, Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. "A Systematic Analysis of the Event-Stream Incident". In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec '22. Rennes, France: Association for Computing Machinery, 2022, pp. 22–28. ISBN: 9781450392556. DOI: 10.1145/3517208.3523753. URL: https://doi.org/10.1145/3517208.3523753.

[5] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. "Challenges of Producing Software Bill Of Materials for Java". In: *IEEE Security & Privacy* (2023). DOI: 10.1109/MSEC.2023.3302956. URL: http://arxiv.org/pdf/2303.11102.

[6] Seyed-Mehdi-Reza Beheshti, Hamid Reza Motahari-Nezhad, and Boualem Benatallah. "Temporal provenance model (TPM): model and query language". In: *arXiv preprint arXiv:1211.5009* (2012).

[7] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, et al. "Prov-dm: The prov data model". In: *W3C Recommendation* 14 (2013), pp. 15–16.

[8] Solution Brief. "Executive order on improving the nation's cybersecurity". In: (2021).

[9] Peter Buneman and Wang-Chiew Tan. "Provenance in databases". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 1171–1173.

[10] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper. "A Primer on Provenance: Better understanding of data requires tracking its history and context." In: *Queue* 12.3 (Mar. 2014), pp. 10–23. ISSN: 1542-7730. DOI: 10.1145/2602649.2602651. URL: https://doi.org/10.1145/2602649.2602651.

[11] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. "Efficient provenance storage". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 993–1006.

[12] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. "Provenance in databases: Why, how, and where". In: *Foundations and Trends® in Databases* 1.4 (2009), pp. 379–474.

[13] Cilium. *Cilium/EBPF: EBPF-go is a pure-go library to read, modify and load ebpf programs and attach them to various hooks in the linux kernel.* URL: https://github.com/cilium/ebpf.

[14] Cilium. *Tetragon, eBPF-based Security Observability and Runtime Enforcement.* URL: https://tetragon.io/.

[15] *CVE-2016-0785*. Available from MITRE, CVE-ID CVE-2016-0785. Dec. 2016. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-0785.

[16] *CVE-2020-13956*. Available from MITRE, CVE-ID CVE-2020-13956. Feb. 2020. URL: https://nvd.nist.gov/vuln/detail/CVE-2020-13956.

[17] CycloneDX. *CycloneDX/cdxgen: Creates cyclonedx software bill of materials (SBOM) for your projects from source and container images.* URL: https://github.com/CycloneDX/cdxgen.

[18] CycloneDX. *CycloneDX/cyclonedx-go: Go library to consume and produce CycloneDX software bill of materials (SBOM).* URL: https://github.com/CycloneDX/cyclonedx-go.

[19] CycloneDX. *CycloneDX/cyclonedx-gradle-plugin: Creates cyclonedx software bill of materials (SBOM) from Gradle Projects.* URL: https://github.com/CycloneDX/cyclonedx-gradle-plugin.

[20] CycloneDX. *CycloneDX/cyclonedx-maven-plugin: Creates cyclonedx software bill of materials (SBOM) from Maven projects.* URL: https://github.com/CycloneDX/cyclonedx-maven-plugin.

[21] CycloneDX. *CycloneDX/cyclonedx-rust-cargo: Creates cyclonedx software bill of materials (SBOM) from rust (cargo) projects.* URL: https://github.com/CycloneDX/cyclonedx-rust-cargo.

[22] Susan B Davidson and Juliana Freire. "Provenance and scientific workflows: challenges and opportunities". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1345–1350.

[23] Hailun Ding, Juan Zhai, Dong Deng, and Shiqing Ma. "The case for learned provenance graph storage systems". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 3277–3294.

[24] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. “The matter of heartbleed”. In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475–488.

[25] Falco. URL: https://falco.org/.

[26] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[27] Raphael Hiesgen, Marcin Nawrocki, Thomas C. Schmidt, and Matthias Wählisch. “The Race to the Vulnerable: Measuring the Log4j Shell Incident”. In: *Proc. of Network Traffic Measurement and Analysis Conference (TMA)*. IFIP, 2022.

[28] White House. “Executive Order on Improving the Nation’s Cybersecurity”. In: *The White House* (2021).

[29] Iovisor. *Iovisor/BCC: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more*. URL: https://github.com/iovisor/bcc.

[30] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. “The ORCHESTRA Collaborative Data Sharing System”. In: *SIGMOD Rec.* 37.3 (Sept. 2008), pp. 26–32. ISSN: 0163-5808. DOI: 10.1145/1462571.1462577. URL: https://doi.org/10.1145/1462571.1462577.

[31] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. “Querying data provenance”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 951–962.

[32] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. “Sok: Taxonomy of attacks on open-source software supply chains”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1509–1526.

[33] Łukasz Makowski and Paola Grosso. “Evaluation of virtualization and traffic filtering methods for container networks”. In: *Future Generation Computer Systems* 93 (2019), pp. 345–357.

[34] Luc Moreau and Paul Groth. *Provenance: an introduction to PROV*. Springer Nature, 2022.

[35] Éamonn Ó Muirí. “Framing software component transparency: Establishing a common software bill of material (SBOM)”. In: *NTIA, Nov* 12 (2019).

[36] Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. “Ariadne: Online provenance for big graph analytics”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 521–536.

[37] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. “A Qualitative Study of Dependency Management and Its Security Implications”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1513–1531. ISBN: 9781450370899. DOI: 10.1145/3372297.3417232. URL: https://doi.org/10.1145/3372297.3417232.

[38] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. "Practical Whole-System Provenance Capture". In: *Symposium on Cloud Computing (SoCC'17)*. ACM. 2017.

[39] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. "Runtime Analysis of Whole-System Provenance". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1601–1616. ISBN: 9781450356930. DOI: 10.1145/3243734.3243776. URL: https://doi.org/10.1145/3243734.3243776.

[40] Hendrik Schoettle. "Open source license compliance-why and how?" In: *Computer* 52.8 (2019), pp. 63–67.

[41] Arun Sharma, P. S. Grover, and Rajesh Kumar. "Dependency analysis for component-based software systems". In: *SIGSOFT Softw. Eng. Notes* 34.4 (July 2009), pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1543405.1543424. URL: https://doi.org/10.1145/1543405.1543424.

[42] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. "Karma2: Provenance management for data-driven workflows". In: *International Journal of Web Services Research (IJWSR)* 5.2 (2008), pp. 1–22.

[43] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. "Looking inside the black-box: capturing data provenance using dynamic instrumentation". In: *Provenance and Annotation of Data and Processes: 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers 5*. Springer. 2015, pp. 155–167.

[44] *Strace(1) - linux manual page*. URL: https://man7.org/linux/man-pages/man1/strace.1.html.

[45] Y. Richard Wang and Stuart E. Madnick. "A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective". In: *Proceedings of the 16th International Conference on Very Large Data Bases*. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 519–538. ISBN: 155860149X.

[46] Evan D Wolff, KM Growley, MG Gruden, et al. "Navigating the solarwinds supply chain attack". In: *The Procurement Lawyer* 56.2 (2021).

[47] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell DE Long. "Evaluation of a hybrid approach for efficient provenance storage". In: *ACM Transactions on Storage (TOS)* 9.4 (2013), pp. 1–29.

[48] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. "JPortal: Precise and efficient control-flow tracing for JVM programs with Intel Processor Trace". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1080–1094.