Over the Shoulder: Improving SBOM Accuracy by Watching the Build

Sanchit Sahay
New York University
New York, NY, USA
s.sahay@nyu.edu

Vyom Yadav
Independent
Chandigarh, India
yadav.vyom28@gmail.com

Abhishek Reddypalle
Purdue University
West Lafayette, IN, USA
areddypa@purdue.edu

Uk Jang
New York University
New York, NY, USA
uj322@nyu.edu

Marco De Vincenzi
New York University
New York, NY, USA
md6796@nyu.edu

Santiago Torres-Arias
Purdue University
West Lafayette, IN, USA
santiagotorres@purdue.edu

Justin Cappos
New York University
New York, NY, USA
jcappos@nyu.edu

Abstract

Software Bills of Materials (SBOMs) are increasingly used as authoritative software inventories but popular generators use techniques that may not expose every input that ends up in or influences a build artifact. This paper presents SBOMit, an OpenSSF project that generates and enriches SBOMs using evidence collected while the software build workflow executes. `witness-ebpf` uses low-overhead tracing methods to record inputs like filesystem and network activity as in-toto attestations, which are later processed into package identities for an enriched SBOM. We evaluate SBOMit across 98 CNCF projects, measuring a median runtime overhead of 8.1% without requiring any modifications to the project source. We further compare SBOMit with static SBOM generation methods to identify build inputs and contexts that static methods fail to recover. Our results show that build observability driven SBOM generation is practical at scale and provides a more trustworthy and auditable source of information.

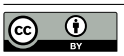
CCS Concepts

• **Security and privacy → Software security engineering**; *Systems security*; • **Software and its engineering → Operating systems**; *Software maintenance tools.*

Keywords

Software Bill of Materials, Software Supply Chain Security, Build provenance, Build observability, SBOMit, eBPF, in-toto, witness

ACM Reference Format:
Sanchit Sahay, Vyom Yadav, Abhishek Reddypalle, Uk Jang, Marco De Vincenzi, Santiago Torres-Arias, and Justin Cappos. 2026. Over the Shoulder: Improving SBOM Accuracy by Watching the Build. In *Conference on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '26), October 06, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3848003.3848006



SCORED '26, Prague, Czech Republic

ACM ISBN 979-8-4007-3009-2/2026/10
https://doi.org/10.1145/3848003.3848006

1 Introduction

Software supply-chain attacks have become a central threat to modern software ecosystems. In 2024, Sonatype reported that more than 704,000 malicious packages were discovered across major ecosystems between 2019 and 2024 [34]. At the same time, attacks such as SolarWinds [8] and Log4j [24] have shown that many organizations and projects have limited visibility into the software they consume, how their own software was built, and which external components it relies on. This lack of visibility makes it harder to accurately respond to these threats.

Software Bills of Material (SBOMs) have emerged as a key mechanism to provide this visibility. In the United States, Executive Order 14028 and NIST encourage the use of SBOMs [27]. In the European Union, the Cyber Resilience Act (CRA) introduces mandatory cybersecurity requirements that mandate the use of SBOMs [11]. The effect is already measurable in a 2026 ENISA survey of 334 EU-based organizations preparing for CRA compliance [12]. Respondents rated achieving SBOM completeness as their most difficult technical challenge (62%), followed by data quality. Their most requested form of guidance was a profile defining what a "good enough" SBOM contains as there is still no agreed baseline for SBOM accuracy or completeness.

However, even with this push toward SBOM adoption, key practical concerns remain unresolved. SBOMs are often consumed as authoritative inventories even though the mechanism used to generate and distribute them may provide limited evidence for accuracy, completeness, or resistance to tampering. Popular SBOM generators such as Trivy [4] and Syft [3] rely on Software Component Analysis (SCA) techniques that infer software composition from sources such as lockfiles, manifests, and container image contents. While these methods are useful, empirical studies show that their outputs can vary significantly across tools, formats, and ecosystems.

O'Donoghue et al. generated SBOMs for 2,313 Docker images and found high variability with identical vulnerability counts observed for only 2–12% of images when just the generator is varied [29]. Balliu et al. compare the component inventories themselves and report similar disagreement across generators for Java projects [5].

SBOMit addresses this inconsistency of SBOM generation by defining and implementing a stronger standard for quality of information that SBOMs rely on to generate a final document. It does so by relying on tracing kernel events generated by the build pipelines. This allows existing SBOM generator tools to be augmented with reliably tracked information about dependencies through filesystem and network accesses.

This paper presents SBOMit's architecture and an evaluation of its practicality across real open-source build pipelines.

In particular, we make the following contributions:

- We describe the architecture of SBOMit, a resolver pipeline that ties build-time observability to SBOM generation, converting low-level evidence into package identities.
- We present `witness-ebpf`, low-overhead eBPF extensions to witness to improve filesystem and network activity tracing for both containerized and non-containerized builds.
- We evaluate SBOMit on two practical metrics: Ease of integration into existing well-maintained projects and the accuracy of SBOMit SBOMs.

2 Background and Motivation

2.1 Motivating Observability-driven SBOMs

Prior work on SBOMs reports frequent disagreements among SBOM generators when they're run against the same software. These disagreements reduce the usefulness of SBOMs given that they obscure the components of a software artifact which in turn reduces the usefulness of downstream consumers like vulnerability detectors that rely on SBOMs being authoritative. In order to study why these disagreements appear, we must recognize that the central challenge of SBOM generation is not merely the choice of an individual tool or format, but the evidence available to the generator.

Existing techniques can be categorized according to what evidence they collect and at what point in the artifact's lifecycle they collect it.

Pre-build declarations. These methods rely on dependency metadata such as package manager manifests and lockfiles. This information is inexpensive to inspect and integrates well into CI workflows. When manifests and lockfiles are complete and up-to-date, they provide a consistent view of a project's declared dependency graph. However, ensuring standardized lockfiles is often not trivial and relies on an ecosystem's support for it [39]. More importantly, these can only inform us about the intentions of a build, not necessarily what the build actually downloads, generates, or embeds.

Artifact inspection. These methods analyze the software artifact produced by the build such as a package archive or container image. This is attractive because it examines the actual deliverables to users. However, artifact inspection is incomplete as a record of build inputs since it only sees components that remained after the build completed. It has no information about how components were fetched or transient build inputs that influenced the generation. In practice, traditional container scanning methods used for vulnerability detection are also prone to miss application packages and non-OS dependencies [16].

Build time evidence. Instead of analyzing declarations or finished artifacts, build time approaches collect evidence when the build executes. Build system specific plugins are the most common form of this which derive a dependency graph as they are resolved and populated by the package manager. These often provide more accurate information about software components and are commonly used as a baseline for evaluating SBOM generators, Balliu et al. [5] being one such work comparing Java SBOMs against `maven-dependency-plugin`. The primary limitation to this is portability since a plugin written for Maven cannot observe npm or Go, nor can it capture activity that falls outside the package manager itself such as accesses to system libraries or vendored source trees.

In order to overcome this limitation, evidence must be gathered at a layer common to all software builds. Filesystem, process, and network activity serve as such a foundation. Although build systems differ substantially in their implementation, they interact with files and network resources through conventions that are stable across individual ecosystems such as package metadata directories, cache layouts, or package registry URLs. Therefore, kernel-level observations of these activities for the build processes can be used to derive package identities while remaining largely independent of any particular ecosystem.

CI runtime-monitoring agents like Harden-Runner rely on the same principle of kernel-level build observations, though its main use case is for anomaly detection [35]. Similar tracing techniques have also been used to target license compliance [38] and reproducible containers [28].

2.2 Provenance & Policy Frameworks

An SBOM can describe the components that appear in a software artifact, but it usually says little about how that artifact was produced. This results in a provenance gap where consumers may know what components are listed but get few guarantees about what inputs were available to track how these components were introduced.

Provenance frameworks address this by recording metadata about supply-chain steps. in-toto provides a general model for attestations and their layouts which describe what happened in a step and how various steps connect to each other. in-toto also provides support for policies which can consume these provenance records [37].

Supply-Chain Levels of Software Artifacts (SLSA) builds on this idea with a more opinionated framework for supply-chain integrity. Its predicate records build definitions, resolved dependencies and other run details for a particular artifact [32].

These systems are complementary to SBOM generators. SBOMs describe software composition, while provenance describes the process that produced the software. Connecting these two is important because component inventories are much more useful and easier to analyze when consumers can also inspect the context of their builds.

3 SBOMit Architecture

The key observation of this work is that the most accurate way to understand what happens in the process of making software is by actually watching that process. This is in comparison to popular SBOM generation tools that use Software Composition Analysis (SCA) and only inspect the resulting binary and/or source files used as inputs.

When a compiler like `gcc` opens a shared object library it does so by calling the `open()` system call. SBOMit captures this call and records the path and content hash of the library. Similarly, when a package manager downloads a package, we store information about the request and response. These records are then used to identify the components involved in the build for the final SBOM.

To achieve this, SBOMit is designed as a pipeline realized in two phases. The first phase captures evidence from a build as it executes, and the second analyzes and converts it into an SBOM document.

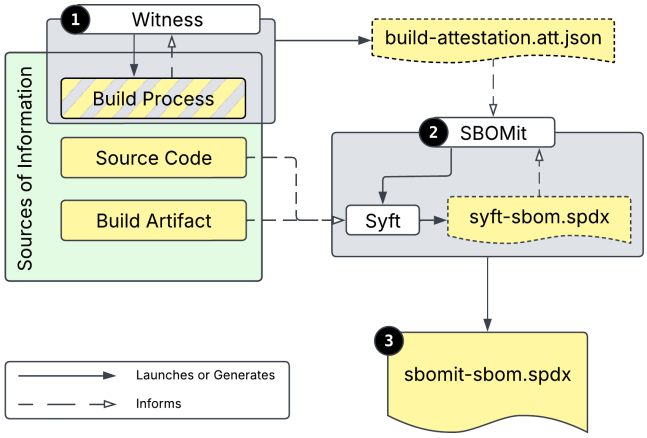


Figure 1: SBOMit uses `witness` to produce in-toto attestations which record the build process and `sbomit` generates the final enriched SBOM.

SBOMit uses `witness` for the first stage which wraps the build command for a software artifact to emit in-toto attestations that record spawned processes, opened files, network fetches, and the generated artifacts.

These attestations are then passed to the `sbomit` generation tool which converts them into package identities which are used to produce a final SBOM or to enrich an existing SBOM produced by SCA tools.

This separation of concerns provides us with some key advantages. First, the in-toto attestations remain first-class output that can be consumed by existing supply-chain tooling for provenance policy enforcement. Second, the recorded build activity can be processed later by improved or alternative analysis algorithms without rerunning the original build. Finally, because the SBOM generation step relies only on in-toto spec attestations, it lowers the barrier for adoption. Existing SBOM generators can incorporate SBOMit by implementing only the attestation analysis phase.

The rest of this section describes in detail the implementation of these two phases. §3.1 begins by laying out SBOMit's requirements for build observability. §3.2-§3.5 describe `witness-ebpf` which acts as a substrate to enable fast, accurate, and detailed tracing. §3.6 describes how `sbomit` consumes evidence to generate enriched SBOM documents.

3.1 Build Evidence Requirements

The primary requirement for build evidence is identification of inputs that contribute towards the construction of a software artifact. These inputs can be local like local libraries, host libraries, vendored packages and caches, or they can be fetched remotely from package registries.

These inputs are consumed through well-known operating system interfaces regardless of the build system. Local inputs are accessed through FS syscalls, while remote inputs are retrieved through socket communication. SBOMit's tracing scope is therefore defined by filesystem and network activities initiated by a build process that rely on these kernel paths which allows our evidence gathering to remain universal. Techniques that by-pass these kernel paths such as shared memory IPC or RDMA traffic are explicitly considered to be out of scope.

To delineate this further, consider the following Python project:

```
examplepkg/
pyproject.toml
src/examplepkg/
    main.py
vendor/helperlib/helper.py
```

Its package metadata declares a dependency on a private package:

```
[project]
dependencies = ["internal-auth==1.0.0"]
```

The release workflow imports `vendor/helperlib/helper.py`, installs `internal-auth==1.0.0` from a private registry, and then runs `python -m build`. Observing the resulting filesystem syscall and network request tell us about the presence of these components and the resulting SBOM can therefore report both the vendored and the publicly available package.

In addition, the following guarantees must hold for this process to be reliable and trustworthy:

(1) **Complete coverage of in-scope activity:** Every filesystem access, lifecycle event, and network interaction for an in-scope process must always be recorded. Failure to record these should cause the build to fail. This guarantee applies to all in-scope activities regardless of whether the resolver knows how to interpret them as package claims allowing the observation and resolution phases to remain independent.

(2) **Retaining Usage Context:** In order to convert recorded data to package identities sufficient context must be stored when observing kernel events. For instance, a recorded `open` syscall should be able to incontrovertibly point to the correct file on disk, regardless of perturbations caused by how the build environment or workflow was hosted. Examples of these being a container's mount namespace which can make the recorded path resolve to a different file, or a process inside a build container carrying multiple PIDs, requiring the tracing layer to determine which one to use and report.

(3) **Provenance and Reusability:** To support SBOMit's broader goal of interoperability with existing supply-chain security practices, build observations must be stored in a portable and verifiable format instead of being proprietary or transient. We choose to represent these as signed in-toto attestations. The build layer should ideally have native support for generating and consuming these attestations.

3.2 Evidence Capture with Witness

To meet the requirements described in §3.1, we built `witness-ebpf`, an eBPF-based observability tool that is capable of executing and observing arbitrary processes and recording these observations as signed in-toto attestations. `witness-ebpf` builds on `witness`, a command-line attestation tool for recording software supply-chain steps [15]. This allows SBOMit to focus on improving evidence collection mechanisms while reusing established provenance infrastructure.

We implement two observability mechanisms, the first replaces the existing process and filesystem tracer with a low-overhead eBPF implementation, extended to support containerized builds. The second introduces a new `network-trace` attestor that records network activity performed during a build.

eBPF was a natural choice for the tracing mechanisms in this setting because it provides the ability to safely observe a broad range of kernel activity without imposing a prohibitive runtime overhead. It works by running verified programs at well-defined kernel hooks allowing tracing logic to execute close to the kernel interfaces that mediate the event [14]. The verifier provides safety guarantees before programs are loaded, and the hook-based execution model keeps userspace processing off the critical path for each observed event. Compared to `ptrace`, `witness`'s current tracing backend, eBPF provides a much lower overhead and allows for more versatile tracing and modification of application behavior, with tools such as `bpf2go` [6] and BPF CO-RE [26, 36] further easing development and portability across kernel versions.

We now define the architecture used to improve performance of the `command-run` attestor and the new `network-trace` attestor which captures network traffic with a high level of granularity. We also describe how these mechanisms extend to tracing containerized builds.

3.3 Low-Overhead Filesystem Tracing

This component records the filesystem activity and process lifecycles that occur during a build. The goal is to identify which files contributed to artifact generation by preserving sufficient context to attribute each observation to a process and a filesystem location. Achieving this requires solving four related problems which we discuss below.

Isolating build activity. We must distinguish build activity from unrelated system processes to ensure build inputs are not mischaracterized. To achieve this, each traced build is launched inside a dedicated cgroup. All attached eBPF programs filter events using this cgroup's identifier ensuring only activity originating from the build command or its descendants contributes to the attestation.

Capturing filesystem activity. Identifying build inputs requires observing both filesystem accesses and process lifecycle events. To accomplish this, eBPF programs are attached to `sys_enter` and `sys_exit` tracepoints for `open`, `openat`, and `openat2`, along with `sched_process_exec` and `sched_process_exit` tracepoints. Tracepoints were chosen over alternatives such as kprobes because they expose stable interfaces across kernel versions and architectures through BPF CO-RE.

Each monitored operation is emitted into a ring buffer shared between the eBPF programs and userspace as an *Event* which contains process identifiers and syscall arguments to reconstruct the context. In addition to normal events, the eBPF programs emit explicit error events when they encounter a fatal error, i.e. any condition that prevents the tracer from faithfully reporting an observed event such as a failure to read a userspace buffer. These error events cause the program to exit with an error.

Retaining execution context. FS observations are only truly useful if these observations can be used to recreate the surrounding context, namely the process that called it and the file that was accessed. This in turn breaks down into two subproblems:

Linux processes can have different PID values at different levels of a PID namespace hierarchy [18]. For example, a process executing inside a container has a different PID within the container and for the host. To retain consistency with other workflows and OS interfaces, the innermost PID namespace is retrieved and used for reporting. This is equivalent to using `bpf_get_ns_current_pid_tgid`.

Additionally, the raw syscall arguments for filepaths in an `open*` syscall are not sufficient to uniquely identify the file that was accessed because it may be interpreted relative to the process's working directory, root directory, or a directory file descriptor [19–21]. Therefore before recording these paths, `witness-ebpf` canonicalizes these paths through `/proc/<pid>/{root,cwd,fd}`.

An alternate approach could attach to Linux Security Module (LSM) hooks where programs can be run after relevant kernel file structs have been populated allowing inodes or `bpf_d_path` to retrieve canonical file paths [22, 23]. However this is not portable since BPF LSM is not consistently enabled by default across Linux distributions [7, 30].

Preventing observation loss. The final challenge is ensuring observations are not silently lost. The explicit error events described above address failures that occur within eBPF programs. A separate challenge is ensuring generated events are not lost due to the inability of the userspace collector to consume events before the ring buffer reaches capacity.

To prevent this, we perform only minimal processing before removing events from the ring buffer and defer expensive work such as I/O bound processing to background goroutines. We also provision the ring buffer conservatively with a capacity of 512 MiB. The userspace consumer is ready before the build starts and keeps consuming events until every build process has exited. Despite this if backpressure prevents a program from reserving space for an event, a separately maintained "drop counter" (implemented as an eBPF map) is incremented which fails the build on non-zero values.

To validate our design, we compared `witness-ebpf`'s file observations with witness's existing `ptrace` backend across 93 CNCF

projects. Both these backends produced matching file records across all projects.

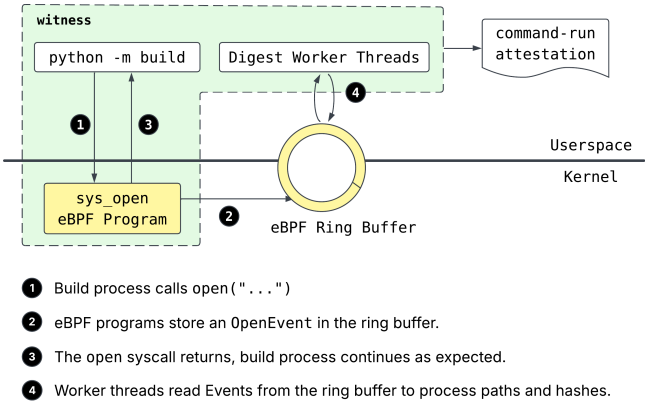


Figure 2: Lifecycle of a file access system call observed by `witness-ebpf`

Figure 2 illustrates the lifecycle of a file access made during a traced build. At a high level, the tracing pipeline proceeds as follows:

(1) Witness creates a new cgroup and launches the build command inside it.
(2) eBPF programs attach to `open*`, `exec`, and `exit` syscalls.
(3) Each monitored event generates an event that is stored in an eBPF ring buffer.
(4) The witness userspace process continuously drains the ring buffer parallel with build execution.
(5) File events are processed asynchronously to compute file digests which are then stored in the attestation.

3.4 Network Tracing

The `network-trace` attestor records external inputs retrieved over the network during a build. The challenge here is that package identity is not trivial to recover because socket events can only reveal remote endpoints, while registry URLs and other package metadata exist in the encrypted application layer [31]. In order to access this encrypted TLS data, `network-trace` uses eBPF programs to redirect traffic to a local man-in-the-middle proxy.

Redirecting traffic through the proxy makes application-layer data visible but it also requires preserving connection context across the redirection. The proxy must be able to recover the original destination and attribute each request to the originating process. The attached `cgroup/connect[4/6]` and `sockops` programs maintain intermediate state about a connection's context through eBPF maps to reconstruct the original intent. The resulting flow is as follows:

(1) The build process (client) initiates an HTTPS connection to a remote package registry like PyPI.
(2) At the `cgroup/connect*` hook, the original destination IP and port, and PID are stored in a BPF map keyed by the client socket cookie.
(3) The same program overrides the original destination IP and port to that of the local MITM proxy. This redirection is transparent i.e. the client still assumes that it is talking to the originally intended destination.
(4) When the client-side TCP connection is established, the `sockops` program receives an `ACTIVE_ESTABLISHED` event and stores the client socket cookie keyed by the redirected connection's TCP four-tuple.
(5) When the proxy accepts the redirected connection, the same program receives a `PASSIVE_ESTABLISHED` event. Here, it reconstructs the corresponding four-tuple to recover the client socket cookie and in turn the original destination information.
(6) Once the original context for the connection has been retrieved, the `sockops` program stores it under the proxy's accepted socket cookie which allows the MITM proxy to recover it directly from the accepted connection.
(7) The proxy terminates the client-side TLS and opens a separate TLS connection to the original destination.
(8) Any subsequent request from the client over this established TCP connection is now handled by the proxy which can decrypt and record HTTPS requests and responses before forwarding them.

Figure 3 charts the lifecycle of an outbound TCP connection request for a proxied build process, and Figure 4 illustrates what information is available to a BPF program at each stage of this process and how the original network context gets stored and reconstructed.

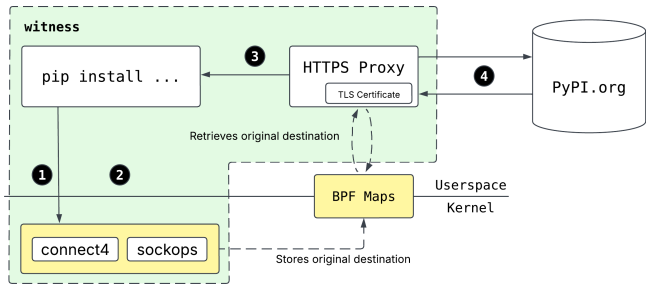


Figure 3: Man-in-the-middle proxying of TCP and TLS handshakes.

To inspect HTTPS traffic, the MITM proxy transparently becomes the client's TLS endpoint and must therefore present a certificate trusted by the build process during the handshake. This can be done either by adding a temporary certificate authority to the build environment's trust configuration or by using a user provided certificate authority through command-line arguments.

This is a requirement inherent to TLS interception and relies on the build environment honoring the configured trust store. Clients that bypass this trust configuration for example through certificate pinning will cause the connections to fail.

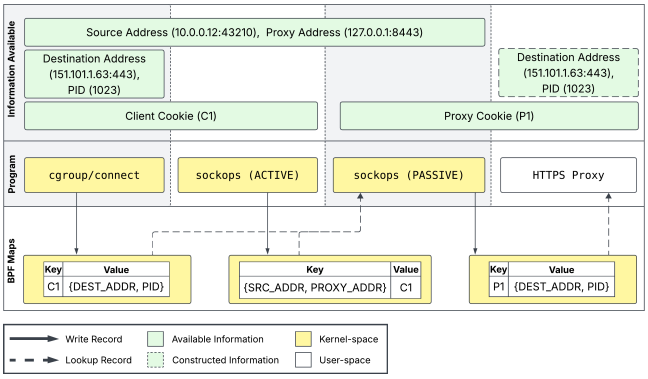


Figure 4: Connection information available at each stage of the proxying pipeline and the BPF maps associating the proxy's accepted socket with the client's intercepted socket.

3.5 Tracing Containerized Builds

The tracing mechanisms described above define the scope of a build to be the process tree spawned by the initial build command. This assumption holds for most conventional build workflows but breaks down when substantial work is delegated to background daemons or executed inside isolated environments. Docker and BuildKit are prime examples of this where the user-visible `docker build` command submits a request to a BuildKit backend which creates a temporary build container to perform the build [9, 25].

In this setting, the requirements of filesystem and network observations remain the same, in fact, the architecture described in the previous sections also stays the same in principle. The challenge instead becomes to accurately expand the build scope beyond the initiating client process without indiscriminately tracing activities not of interest to the build.

We describe below the challenges faced in addressing this gap and the resulting changes, using Docker builds as a concrete example.

Discovering delegated workloads. For a Docker build, the intended scope includes both the client process and the build container created by dockerd which carries out its request. The latter is not in scope by default since it does not belong to the client process's ancestry and therefore is also not a part of the cgroup we observe.

To extend the scope to include daemon-drive builds, the user provides the daemon's process ID via a command-line argument, and `witness-ebpf` additionally tracks the daemon's process tree. The userspace component enumerates all thread IDs under `/proc/<pid>/task/` for `dockerd` and populates a per-thread BPF allowlist at startup. Thereafter, the `sched_process_fork` tracepoint propagates tracking automatically, building the tree transitively throughout the daemon's process hierarchy.

Tracking TIDs instead of a blanket cgroup requires special handling of the `execve()` syscall. `execve` de-threads the process by terminating every thread execpt the caller, including the group leader. Each terminated thread is removed from the allowlist by the exit handler. When the new program image is loaded, the calling thread assumes the group leader's TID, with `old_pid` holding its own former TID. This causes the newly exec'd process to go untracked, since neither its assumed TID not the `old_pid` remains on the allowlist. To avoid this, we save a mapping from the thread's host-level ID to its namespace-local ID in a bridge structure before execve() runs. This is used afterwards to add the new group leader TID to the allowlist before removing the old one ensuring that a concurrent `connect()` on the new TID never finds neither entry present. Figure 5 illustrates this swap sequence in detail.

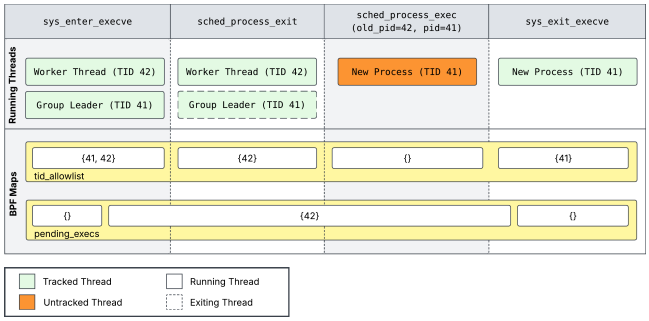


Figure 5: Preserving process tracking across an `execve()` call in a multi-threaded program.

Tracking container boundaries. Since a Linux container can be tracked through cgroups and namespaces, `witness-ebpf` monitors the host for newly created cgroups, PID namespaces, and network namespaces from the moment the initiating command runs, treating this as the start of a bounded observation window. The filesystem and network tracers then use these boundaries to extend their respective scopes, as described below.

New cgroups are detected by the `cgroup_mkdir` tracepoint which emits the cgroup identifier and path whenever a new cgroup is created. Once a cgroup creation event is emitted, an optional acceptance policy can be applied to them before making them visible to the individual tracing mechanisms.

Because this mechanism relies on Linux isolation primitives rather than instrumentation specific to Docker, it applies to any container-based build workflow that uses the same abstractions, such as podman, containerd, or CRI-O. The current implementation assumes that no unrelated container builds execute concurrently during the observation window.

Extending filesystem scope. The `command-run` syscall tracepoints execute for every `open*`, exec, and exit event on the host and determine whether an event is relevant by checking the originating process's cgroup against a BPF map of accepted cgroups.

For an ordinary build, this map only contains the cgroup assigned to the command executed by `witness`. For a containerized build, cgroups discovered and accepted during the observation window are added to this map, extending the existing filter to processes running inside the build container.

The remainder of the filesystem tracing pipeline is unchanged. Retained events include the host-visible PID and the path relative to the container's mounts.

Extending network scope. The scope of network observations is governed by the connections that can be redirected through a userspace MITM proxy. Since a build container typically executes in a separate network namespace [10] with its own loopback interface, from which the proxy listening on the host is unreachable [17], network tracing must instead ensure that a reachable proxy is available inside each relevant namespace before any connection can be intercepted.

To address this, `witness-ebpf` provisions a proxy within each newly created network namespace before allowing the namespace's processes to initiate an outbound network request. Transitions into a new network namespace are intercepted through the `sys_exit_clone`, `sys_exit_clone3`, `sys_exit_unshare`, and `sys_exit_setns` tracepoints. These hooks fire on syscall exit, after the namespace change has taken place but before the task returns to userspace and can issue a `connect()` call. For `clone` and `clone3`, only the child path is acted upon, since the child is the task that may have been placed into a new network namespace. The resulting flow is as follows:

(1) A tracked task creates or enters a network namespace. The syscall exit tracepoint fires and checks whether the new namespace already has a ready proxy.
(2) If no proxy is ready, the eBPF program publishes a gate entry recording the task's host-level TID and a timestamp. It then re-checks proxy readiness: if the userspace sweep completed proxy setup in the interval between the initial check and the publish, the task deletes its own gate entry and proceeds without being frozen.
(3) If the proxy is still not ready after the re-check, `SIGSTOP` is delivered via `bpf_send_signal`, freezing the task before it can execute any userspace instruction.
(4) The userspace sweep loop reads the gate map and, for each namespace with pending entries, enters the namespace using `setns` and binds a proxy listener on the specified interface. The namespace is then marked as ready in a proxy state map that is never subsequently cleared.
(5) The suspended tasks are resumed via `SIGCONT` and their subsequent `connect*` calls are redirected to the now-ready namespace-local proxy.

This protocol avoids race conditions because of how `SIGSTOP` is delivered at the syscall exit boundary. The tracepoint fires on the kernel-to-user return path, and `bpf_send_signal` queues `SIGSTOP` to the current task. When the kernel drains the pending signal queue before returning to userspace, the task is frozen before it can execute any subsequent syscall, including `connect()`.

Each namespace's proxy uses the same certificate authority, which must be trusted in the build container's own trust store.

3.6 SBOM Generation

The second phase of the pipeline is performed by `sbomit`, which consumes the attestations produced during the build and emits an SBOM document. The goal of this phase is to interpret attestations that record low-level events such as file accesses and network requests, while an SBOM must make package-level claims. Generation is therefore guided by three design principles:

(1) **Evidence preservation.** Observed build inputs should not be omitted even if they cannot be mapped to a package. Such inputs are retained as unresolved file entries along with their hashes.
(2) **Package-level attribution.** An SBOM that lists thousands of file paths is technically accurate but is not practical due to its size and incompatibility with SBOM consumers. Wherever a path follows package manager conventions, it should be listed as a package identity.
(3) **Format validity.** The final document emitted must adhere to existing SBOM standards so that it remains consumable by existing tooling.

To motivate the generation pipeline, consider the Python example from §3.1. A static scan of the source tree would identify the declared dependency `internal-auth==1.0.0`, whereas the attestation additionally records the vendored libraries that were accessed and the package index that these dependencies were fetched from. The generation layer preserves this evidence into a useful SBOM representation. For instance, a package URL such as `pkg:pypi/internal-auth@1.0.0` is attached to the package metadata path, which preserves important information about how the dependency entered the build.

The implementation is organized into three components that adhere to the design principles mentioned above. Each of these components is designed to be easily extensible for adding support for new attestors, ecosystems, or SBOM formats. These are described in detail below:

Attestation Resolvers. The first stage parses the input (provided either as direct in-toto statements or DSSE envelopes) and normalizes each attestor's records into typed evidence entries without making any package-level claims. These include: `material` and `product` record the files present before and produced by the build step respectively, `command-run` records the processes spawned and the files they accessed, and `network-trace` records outbound network requests. Individual evidence classes can be included or excluded from the command line.

Ecosystem Resolvers. The second stage lifts raw evidence into package identities. File evidence flows through an ordered resolver chain where each resolver receives the current set of unattributed files, claims those it recognizes as belonging to a package pattern it understands, and passes the remainder forward.

Network evidence follows the same pattern and serve two roles. First is that of provenance, recording which registry or mirror a package from fetched from (illustrated in §4.2). Second, for ecosystems like `npm`'s node_modules whose install paths do not encode a version, the version is recovered from the request URL.

The current implementation includes resolvers for common Python, Go, Rust, and JavaScript patterns, along with noise filters that discard build-transient paths such like temporary files before resolution begins.

This design relies on the assumption that package manager paths encode enough information to recover a package name and version. When that assumption holds, SBOMit emits a normalized package record containing the ecosystem, name, version, PURL, optional hashes, and a `FoundBy` value identifying the resolver. When it does

not hold, it is still listed in the SBOM as a file record, per the first design principle.

SBOM Document Generation. The final stage emits a standards-compliant SBOM, using `protobom` as a format-neutral intermediate model. Deduplication is applied here by merging package records recovered by different resolvers by PURL, and files attributed to a package are removed from the unresolved set.

The inventory produced so far is flat since the attestation cannot detect the dependency relationships between observed packages. To recover that, SBOMit can optionally merge its output with a base SBOM produced by an external cataloger such as Syft or Trivy. The base document is parsed into the same `protobom` model and nodes are matched by PURL, with attestation backed metadata taking precedence when the two sources disagree. This recovers dependency edges only for components that both tools identify (more in §5). In the final document, every observed input is accounted for. Evidence that maps to a package appears as a normalized component, and everything else is retained as a file entry with its hash.

4 Evaluation

We evaluate SBOMit along two dimensions:

1. What effort is required to adopt SBOMit for maintainers of popular open-source projects, and what runtime overhead does an SBOMit build workflow cause?
2. Under what scenarios does SBOMit generate meaningfully different SBOMs when compared to SCA tools?

4.1 Adoption Effort and Runtime Overhead

In order for SBOMit to be a viable solution, it must not require significant engineering effort to adopt into existing workflows, and it must also not impose a prohibitive runtime overhead.

To evaluate this, we targeted projects in the CNCF Landscape [13] spanning all three maturity tiers (Graduated, Incubating, and Sandbox). Out of the 225 total projects (as of May 26, 2026), we ran our experiments on 98 randomly sampled projects (14 Graduated, 23 Incubating, and 61 Sandbox). This subset spans all eight top-level landscape categories represented in the matched CNCF Landscape metadata and all maturity tiers, and we therefore regard it as sufficiently representative.

Adoption Effort. Since SBOMit wraps existing build commands, we use the ease of identifying a repeatable build command as a stand-in to measure ease of integrating SBOMit into a given workflow.

For most projects this was provided by Makefile targets such as `all` or `build`. When absent, we used the language specific build command like `go build` or `cargo build --release` etc.

We were able to identify a build entry point without modifying any of these projects. 84 of the 98 (85.7%) exposed a standard Makefile target. The remaining 14 used a build command: `go build` (6), `python -m build` (3), `cargo build` (2), `docker build` (2), and `uv build` (1). These identified commands were then wrapped with `witness` without making any modifications to the project's source tree.

Runtime Overhead. To measure performance overhead, we ran each project's build under three conditions: uninstrumented build, build instrumented by the ptrace-backed witness and build instrumented by witness-ebpf.

Each of these runs was provided with the same environment. All experiments were conducted on a single Google Cloud Platform (GCP) virtual machine with 16 vCPUs and 64 GB of RAM, running Ubuntu 22.04.5 LTS (x86-64, kernel 6.8.0-1058-gcp). Dependency and build caches were cleared before each run. This ensures their results are comparable and also forces the build to perform dependency resolution so that the final attestations were representative and complete.

Table 1 summarizes the results of these runs. Tracing overhead is reported as the relative increase in wall-clock build time over the established baseline. Along with mean and median overhead for both the instrumentation techniques, we also report the 90th percentile overhead.

Out of the 98 selected projects, 5 generated Docker images as final artifacts for which `witness-ebpf` added a median overhead of 6.9% and average overhead of 13.6%. Since these do not have a `ptrace` equivalent, they were excluded from the overhead calculations.

Across the remaining 93 projects, `witness-ebpf` incurred a median overhead of 8.1% and a mean overhead of 9.1% over uninstrumented builds (8 runs, 1.1% co-efficient of variance), while `witness-ptrace` incurred a median overhead of 564.7% and a mean overhead of 1045.9%.

Case	Median	Mean	P90	Maximum
witness-ebpf	8.1%	9.1%	12.7%	69.3%
`witness-ptrace`	564.7%	1045.9%	2049.0%	9766.7%

Table 1: Tracing overhead summary for CNCF projects.

Since the principal barriers to adoption are the effort required by project maintainers and the risk of slowing existing release workflows, these results demonstrate that SBOMit is a practical workflow to integrate into existing projects. In most cases, adoption requires only simple changes to a GitHub Actions workflow file, and the low overhead keeps SBOM generation out of contributors' way.

4.2 SBOM Information Gaps

SBOMit's value also hinges on it providing information that materially improves the final SBOMs. We therefore examine the kinds of dependency information available to a build time observability generator like SBOMit and the circumstances under which they enrich an SBOM.

We evaluate a set of representative build scenarios, each of which is designed to expose a distinct source of divergence between source inspection, artifact inspection, and build observability. These scenarios are intentionally small to facilitate of manual inspections.

For each project, we generate SBOMs using Syft v1.45.0 and Trivy v0.71.2 over both the source tree and the final build artifact. We also generate an SBOM using SBOMit on the same project which allows us to compare these techniques based on the resulting SBOMs. This comparison is qualitative and is not intended to establish accuracy scores or tool rankings.

Case	Expected Source of Divergence
Build time dependency resolution	A direct unpinned dependency whose version is resolved at build time.
PEP 517 Build Tool	Transient dependencies used to produce an artifact.
Private Packages	Packages that are fetched from a private registry and may not exist on PyPI.
Conditional Dependencies	Target and platform sensitive dependencies (e.g. Python version, OS, Makefile target)
Transient Docker Dependencies	Build dependencies installed in a transient stage of a Dockerfile.

Table 2: Descriptions of controlled examples for observing sources of divergences between static SBOM generators and SBOMit.

Example	Static SBOMs	Build-observed SBOM
Unpinned Dependency	`packaging` 24.0, unpinned `idna` ignored.	`packaging` 24.0, `idna` 3.18, build tools dependencies.
PEP 517 Build Tool	no build-system packages.	`wheel` 0.47.0 and `setuptools` 83.0.0.
Private Index	`private-pkg` 1.0.0	`private-pkg` 1.0.0 and private registry metadata.
Conditional dependency	All branches reported: `idna` 3.18, `colorama` 0.4.6, and `importlib-metadata` 8.7.1.	Only the Linux/Python 3.10 requirement, `idna` 3.18.
Docker Build Stage	Final image dependencies. No `build` stage libraries.	Build stage includes `Cython` 0.29.36, `wheel` 0.45.1, `setuptools` 80.9.0, `python3-dev`, and `gcc` used to generate the native extension.

Table 3: Summary of the dependency information available to static SBOM generators and SBOMit across the controlled builds.

Table 2 summarizes the representative build scenarios used in this evaluation and Table 3 summarizes the resulting package inventories generated by each tool. Together these scenarios isolate common mechanisms responsible for SBOM divergence between SCA tools and SBOMit. We discuss our observations below and relate them to similar patterns observed when running SBOMit on the CNCF projects where appropriate.

Build time Resolution. Most package managers allow dependency versions to be determined during the build rather than being completely defined in the manifest. Common examples of this are version ranges, generated dependencies, or inconsistent lockfile management. When such a project is built, the dependency versions used to produce an artifact cannot necessarily be reconstructed from the repository source or the final artifact.

In our example, we used a Python project that specifies a `requirements.txt` with `packaging==24.0` and unpinned `idna`.

Both Syft and Trivy only report packages for which they have concrete version information and both tools silently ignore the dependency otherwise. Syft allows for optional flags which try to estimate the version. In contrast, SBOMit records the dependency resolution as it occurs and therefore reports `idna` with the correct version.

Transient Build Inputs. Modern build systems often create isolated environments and install transient tools that are required only for artifact generation. While these packages do not end up in the final artifact, they influence the resulting artifact regardless.

To evaluate this, we generate a Python wheel through a build script that adheres to the PEP 517 [33] interface and requires the use of `wheel` and `setuptools`. Neither Syft nor Trivy report these packages in the final SBOM, while SBOMit does.

Multi-stage Docker builds present this loss of build provenance as well. In our example, the build stage installs `gcc` and `libffi-dev` and uses them to build a `cffi` wheel. Syft and Trivy report the contents of the final image that includes the Python runtime and base system packages, but does not include build inputs like `Cython`, `wheel`, `setuptools`, `python3-dev`, or `gcc`.

The same issue appears in `python-tuf` where a `python3 -m build` requires 7 build dependencies all of which are reported by SBOMit and not by Syft.

Such information is relevant because these host binaries and libraries can themselves become an attack vector. For example, the XZ backdoor demonstrated that widely trusted system libraries used during software builds are prime targets for attacks [2].

Package Sources. A package name and version do not completely identify a package. Organizations frequently use private registries and mirrors instead of the public registry in which case the same name and version may refer to different artifacts.

Our private registry example contains a package that shadows a publicly available one on PyPI and another that only exists in the private index. Syft and Trivy report both these packages but cannot report the source registry. SBOMit reports their sources computed by observing the interactions with remote registries allowing a way to differentiate between two otherwise identical packages.

Even when the package inventory itself is complete, provenance about the source can still provide us with meaningful safeguards. In `jaeger`, for example, Syft correctly identified all package components while SBOMit was able to add information about the Go module cache entries that were used to satisfy them.

This matters because caches are active build inputs. Cache-poisoning techniques can cause malicious or stale contents introduced by one workflow to be reused by a later one. Recording this information therefore helps post-incident investigations identify compromised software.

Target Sensitivity. Finally, we construct a Python project that uses conditional dependency markers based on the operating system and the Python version. The project declares `idna==3.18` for Linux, `colorama==0.4.6` for win32, and `importlib-metadata==8.7.1` for python_version < "3.10".

Syft and Trivy report all three declarations. In the Linux/Python 3.10 build used for this experiment, pip only installs `idna==3.18` and the remaining dependencies are ignored. SBOMit therefore only reports the package selected by the observed build. This highlights a distinction between possible and realized dependencies.

Another manifestation of the same limitation is when a single source tree can produce multiple artifacts. For example, `headlamp`'s frontend and backend Makefile targets are built from the same repository but exercise completely different dependency sets. A full-repository scan with a static tool reports 3,266 packages of which only 331 are Go packages that the backend target touches, while the rest are npm packages that belong to the frontend targets. Source-based tools cannot adequately distinguish between these targets and overreport packages in this case.

4.3 Takeaways on SBOM Policy & Scope

Across these examples, we observe that SBOM policy and generator implementation have become closely intertwined. Decisions like omitting unresolved dependencies or reporting all conditional dependencies appear as if they are intentional decisions about what an SBOM should contain, but in practice they are often shaped by what a particular collection technique can readily observe. This risks allowing SBOM policy decisions to be implicitly governed by limitations of current tools.

We argue that these concerns should be separated. Collection mechanisms should be implemented and evaluated to serve predetermined security, compliance, and auditability goals. SBOMit enables this separation by collecting a broad set of information about builds and allowing the inventory to be constructed from it independently without having to trigger a new release.

Our results further suggest that SBOMs should retain richer provenance information and be held to a higher standard for completeness. Information about package sources and other build influences can help SBOMs function not merely as passive compliance artifacts but as active defenses against supply-chain attacks.

Because incomplete inventories can misdirect vulnerability analysis and incident response, SBOM claims should be grounded in unambiguous evidence from build activity rather than from sources which can have multiple conflicting interpretations.

5 Discussion & Future Work

Upstream Integration. The eBPF-based filesystem and network tracing mechanisms introduced in `witness-ebpf` have been upstreamed into `witness` and integration of support for containerized builds is underway. Relaxing our current assumption that no concurrent docker builds occur is a goal for future work with discussions currently underway on the shape of the solution.

Since SBOMit's native output is a flat inventory without any dependency relationships (§3.6), we are also in discussions with the Syft team to upstream our resolution techniques. This would allow build time evidence to inform SBOMs without requiring post-hoc merging.

Making these capabilities available through upstream `witness` and Syft alike allows their existing user and contributor communities to adopt them without having to rely on a separate implementation.

Threat Model and Scope. Although SBOMit and `witness` can observe arbitrary processes, they are designed to target conventional build pipelines rather than adversarial workloads that actively evade tracing. We also assume that the build executes on a trustworthy kernel and tracing infrastructure. These assumptions have held for every build system and environment we have seen.

SBOMit is designed for observability and not policy enforcement. Detecting or blocking spurious processes and network activities are out of scope.

Package Identification Limitations. Moving evidence collection below individual package managers does not eliminate the need for ecosystem-specific interpretation, since identifying package metadata still depends on a resolver for each ecosystem's spec.

Moreover, these conventions are often unreliable. For instance system package managers like `apt`, `rpm`, and `apk` do not carry any meaningful version information in their path at all. For this, path-to-package mapping offers a partial solution through indices such as Debian's Contents which map paths to the parent packages but cannot pin a version since the same path ships unchanged across releases. A more reliable signal is the per-file hash that our attestations which could let a resolver identify exact package versions from file hashes. However, no public database currently maps individual file hashes to package versions. Existing lookup services operate at the artifact level, indexing hashes of whole wheels, jars, or tarballs. Content-addressed archives such as Software Heritage [1] resolve a hash to a blob rather than to a package identity.

Cataloging Limitations. As a standalone tool SBOMit is only able to infer presence of packages but not their dependency relationships or role in the build. Utilizing SBOMs generated by SCA tools allows us to solve for the dependency graph, but we still need a way to distinguish whether a resolved package was a delivered artifact, a build tool, or a host library.

Separately, a large number of files accessed during a build typically are either redundant or cannot be associated with any package at all, e.g. `jaeger` accessed over 31,000 such files. Refining our cataloging to better handle both cases remains future work.

6 Conclusion

SBOMit demonstrates that grounding SBOM generation in tracing build pipelines is not only a stronger basis for accurate and auditable software, but also technically feasible and readily adoptable in real-world projects. SBOMit replaces brittle and bespoke collection mechanisms of traditional SCA tools with a catch-all technique. Through our work on `witness-ebpf`, we find that this can be done with a minimal build overhead, with our evaluations demonstrating a median overhead of 8.1% across CNCF projects of varying maturity. These results suggest that generating SBOMs through build time observability can be made practical for CI/CD pipelines without imposing prohibitive performance costs.

SBOMit is designed to fit into existing SBOM workflows and standards instead of attempting to replace them. This is enabled by the architectural separation between `witness` that collects evidence and the analysis tool that interprets it. This split also makes the system extensible. New attestors can improve coverage while new resolvers can interpret previously unresolved evidence without changing the core model. Through all this, SBOMit provides a clearer path toward SBOMs that are designed to be trustworthy.

Availability

We present our contributions under the Apache-2.0 license:
`sbomit`: https://github.com/SBOMit/sbomit (tag: v0.2.1)
`witness-ebpf`: https://github.com/stupendoussuperpowers/go-witness (tag: v2026-scored)
Evaluations: https://github.com/stupendoussuperpowers/sbomit-scored-26/

Acknowledgments

We thank the reviewers for their feedback, along with John Kjell and the `witness` maintainers for their support throughout this project.

This work was supported by the National Science Foundation (NSF) through Grant No. 2346219. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. 2018. Building the Universal Archive of Source Code. *Commun. ACM* 61, 10 (2018), 29–31. doi:10.1145/3183558

[2] Akamai Security Intelligence Group. 2024. XZ Utils Backdoor. Akamai Blog. https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know.

[3] Anchore, Inc. 2026. Syft: CLI Tool and Library for Generating a Software Bill of Materials (SBOM). https://github.com/anchore/syft. Last accessed July 17th, 2026.

[4] Aqua Security. 2026. Trivy: Comprehensive Security Scanner for Containers, Kubernetes, Repositories, and More. https://github.com/aquasecurity/trivy. Last accessed July 17th, 2026.

[5] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. 2023. Challenges of Producing Software Bill of Materials for Java. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[6] Cilium. 2026. bpf2go. https://github.com/cilium/ebpf/tree/main/cmd/bpf2go. Last accessed July 17th, 2026.

[7] Kees Cook. 2020. UBUNTU: Enable `CONFIG_BPF_LSM`. Ubuntu Mailing List https://lists.ubuntu.com/archives/kernel-team/2020-December/115438.html.

[8] Cybersecurity and Infrastructure Security Agency. 2020. Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations. Cybersecurity Advisory AA20-352A https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a. Originally published December 17, 2020; updated April 15, 2021.

[9] Docker Inc. 2026. Docker Build Overview. https://docs.docker.com/build/concepts/overview/. Accessed July 2026.

[10] Docker Inc. 2026. Docker Networking. https://docs.docker.com/engine/network/.

[11] European Parliament and Council of the European Union. 2024. Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act). Official Journal of the European Union, OJ L, 2024/2847. https://eur-lex.europa.eu/eli/reg/2024/2847/oj.

[12] European Union Agency for Cybersecurity (ENISA). 2026. *SBOM Adoption State of Play - 2026.* Technical Report. European Union Agency for Cybersecurity (ENISA). https://www.enisa.europa.eu/sites/default/files/2026-06/SBOM%20Adoption%20State%20of%20Play%202026.pdf

[13] Cloud Native Computing Foundation. 2026. CNCF Landscape. https://landscape.cncf.io.

[14] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunstrom. 2024. The eBPF Runtime in the Linux Kernel. *arXiv preprint arXiv:2410.00026* (2024). doi:10.48550/arXiv.2410.00026

[15] In-toto Project Contributors. 2026. Witness. https://github.com/in-toto/witness. Last accessed July 17th, 2026.

[16] Omar Javed and Salman Toor. 2021. An Evaluation of Container Security Vulnerability Detection Tools. In *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing* (Liverpool, United Kingdom) *(ICCBDC '21)*. Association for Computing Machinery, New York, NY, USA, 95–101. doi:10.1145/3481646.3481661

[17] Linux man-pages project 2026. *network_namespaces(7) — Linux manual page*. Linux man-pages project. https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[18] Linux man-pages project 2026. *pid_namespaces(7) — Linux manual page*. Linux man-pages project. https://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[19] Linux man-pages project 2026. *proc_pid_cwd(5) — Linux manual page*. Linux man-pages project. https://man7.org/linux/man-pages/man5/proc_pid_cwd.5.html.

[20] Linux man-pages project 2026. *proc_pid_fd(5) — Linux manual page*. Linux man-pages project. https://man7.org/linux/man-pages/man5/proc_pid_fd.5.html.

[21] Linux man-pages project 2026. *proc_pid_root(5) — Linux manual page*. Linux man-pages project. https://man7.org/linux/man-pages/man5/proc_pid_root.5.html.

[22] Linux Maintainers. 2026. Helper function `bpf_d_path`. https://docs.ebpf.io/linux/helper-function/bpf_d_path/.

[23] Linux Maintainers. 2026. LSM BPF Programs. https://docs.kernel.org/bpf/prog_lsm.html.

[24] MITRE Corporation. 2021. CVE-2021-44228: Apache Log4j2 Remote Code Execution Vulnerability (Log4Shell). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228. Last accessed July 17th, 2026.

[25] Moby Project. 2026. BuildKit. https://github.com/moby/buildkit. Accessed July 2026.

[26] Andrii Nakryiko. 2020. BPF Portability and CO-RE. https://nakryiko.com/posts/bpf-portability-and-co-re/. Last accessed July 17th, 2026.

[27] National Institute of Standards and Technology. 2021. Executive Order 14028: Improving the Nation's Cybersecurity. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.

[28] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 167–182. doi:10.1145/3373376.3378519

[29] Eric O'Donoghue, Brittany Boles, Clemente Izurieta, and Ann Marie Reinhold. 2024. Impacts of Software Bill of Materials (SBOM) Generation on Vulnerability Detection. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. ACM, 67–76. doi:10.1145/3689944.3696164

[30] Oracle. 2026. BPF-LSM Enabled at Boot. https://docs.oracle.com/en/operating-systems/uek/7/relnotes7.3/7.3-feature-bpf-lsm.html.

[31] Eric Rescorla. 2018. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. https://datatracker.ietf.org/doc/html/rfc8446.

[32] SLSA. 2026. SLSA Specification v1.2. https://slsa.dev/spec/v1.2/. Last accessed July 19th, 2026.

[33] Nathaniel J. Smith and Thomas Kluyver. 2015. PEP 517 – A build-system independent format for source trees. https://peps.python.org/pep-0517/. Python Enhancement Proposals. Accessed September 2026.

[34] Sonatype. 2024. State of the Software Supply Chain: 10 Year Look. https://www.sonatype.com/state-of-the-software-supply-chain/2024/10-year-look.

[35] StepSecurity. 2026. Harden-Runner. https://github.com/step-security/harden-runner. Accessed September 2026.

[36] The Linux Kernel Documentation. 2026. BPF Type Format (BTF). https://docs.kernel.org/bpf/btf.html. Last accessed July 17th, 2026.

[37] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1393–1410. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[38] Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. 2014. Tracing software build processes to uncover license compliance inconsistencies *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 731–742. doi:10.1145/2642937.2643013

[39] Li Zhou, Marc Dacier, and Charalambos Konstantinou. 2026. A Reality Check on SBOM-based Vulnerability Management: An Empirical Study and A Path Forward. In *Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy* (Frankfurt am Main, Germany) *(CODASPY '26)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3800506.3803490