International Journal of Cloud Computing and Cyber Security (IJCCCS)
ISSN: XXXX-XXXX / Volume 1 Issue 2 November-December 2026
Paper Id: JCCCS -V1I2P105/ Page No: 59-72
Doi: XX.XXXXX/XXXXXXX/ IJCCCS -V1I2P105

Original Article

Securing Software Supply Chains by Correlating Cloud Build Logs and Dependency Risk Signals

Dawn Song

University of California, Berkeley.

Received Date: 22 November 2026 Revised Date: 26 November 2026 Accepted Date: 30 November 2026

Abstract: *Software supply chain security has emerged as a major area of focus for many organizations due to increasing reliance on cloud-based build environments, automated continuous integration/continuous delivery (CI/CD) pipelines, open source libraries, and third-party development tools. While traditional security assessments often look at each of these areas independently (i.e., assessing source code, evaluating software dependencies, and reviewing infrastructure configurations), doing so does not account for the fact that some of the most sophisticated types of attacks arise out of interactions among different aspects of the software development process, namely the build activity itself and dependency related risks. In response to this need, this research paper presents a holistic approach to security that links cloud-based build logs to dependency-related risk signals in order to provide better detection, investigation, and prevention capabilities against software supply chain attacks. This approach will utilize several components including: build event monitoring; vulnerability assessment of software dependencies; package provenance verification; behavioral anomaly detection; and contextual risk scoring. Organizations can use these components collectively to help identify when a software production process may have been compromised by identifying unusual build identity; unexpected modifications to a workflow; suspicious downloads of packages; vulnerable dependency versions; and deviations from normal build practices. In addition to presenting an architectural component view, the research also explores correlation techniques, operational issues, and evaluation criteria for deploying this type of approach in a cloud native development environment. Specifically, the research addresses explainability; scalability; reduction of false positives; and how it can integrate into existing DevSecOps practices. Ultimately, this research provides a conceptual framework for achieving greater transparency throughout the entire software production process as well as the dependency ecosystem.*

Keywords: *Software Supply Chain Security, Cloud Build Logs, Dependency Risk, DevSecOps, Build Provenance, Vulnerability Detection, Behavioral Analytics, Continuous Integration, Risk Correlation, Cloud Security.*

I. INTRODUCTION

Software development is transitioning from a series of independent development activities to an interconnected production ecosystem consisting of cloud computing platforms, automated build services, source code repositories, package registries, container images, third party libraries and continuous integration/continuous delivery (CI/CD) pipelines. This new ecosystem offers improved development speed, increased collaboration, and more consistent deployments; however, it also presents many additional security challenges beyond just the application's source code. A software product can have hundreds, even thousands, of external dependencies, which can each present vulnerabilities, malicious functionality, outdated components or poorly maintained practices. Additionally, cloud based build environments are used to execute workflows that pull source code, download dependencies, compile applications, run tests, package artifacts, and release products. Attackers can therefore gain control over the resulting software through various means other than directly modifying the principal source code of the application.

For example, attackers could compromise a dependency, abuse a build identity, modify a pipeline configuration, or misuse excessive permissions. Therefore, software supply chain security must address not only what components are included in a software product but also how those components are obtained, processed, validated and transformed throughout the production process. Traditionally, security approaches typically focus on identifying vulnerabilities in dependencies and build infrastructures separately. While each of these approaches provides valuable insight into potential security concerns, they create blind spots in visibility when individually non-suspicious actions combine to create suspicious activity.Cloud build logs contain documentation of what was done as part of the process for producing software. Depending upon the platform and how the logging is configured,

This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/2.0/)

they will include, among other things; timestamps, build identifiers, execution identities, workflow stages, source repository references, package retrieval events, permission changes, network connections, and artifact publication outcomes. Risk signals generated from dependencies offer additional context regarding potential risks associated with using particular components. These signals may include known vulnerabilities, malicious package indicators, abandoned/unmaintained components, suspicious version changes, provenance inconsistencies, and deviations from established dependency policy.

Each type of signal can individually provide an incomplete picture. A dependency containing a publicly documented vulnerability may pose minimal immediate risk if it is not used within a released artifact. Conversely, a dependency that has no known vulnerabilities may pose significant risk if it is introduced via an unexpected build workflow or downloaded from an unapproved registry. An unusual build event may simply be the result of routine maintenance rather than malicious intent. By combining both types of signals, organizations can begin to identify relationships between build behaviors, dependency characteristics, artifact lineage, and release decisions. These types of relationships can then form the basis for performing contextual risk assessments which allow security teams to determine incident priorities based upon a combination of technical evidence, operational circumstances and potential impacts as opposed to simply relying on isolated alerts or static vulnerability ratings.

The main goal of this project was to establish a conceptual model to secure software supply chains through the use of correlation between cloud build logs and dependency risk indicators. To achieve this, we developed a single conceptual process for securing the software supply chain. We combined log collection, dependency inventorying, provenance verification, event normalization, temporal and identity-based correlation, contextual risk scoring and investigative support. This process would allow us to detect potential threats in the form of vulnerable dependencies being introduced into modified pipelines; unknown packages being introduced by builds using privileged credentials; unexpected changes to dependencies made prior to production release; and artifact publication without proper verification evidence. Our work also considered some of the practical issues related to implementing our process such as data quality; cross platform compatibility; privacy protection; computational scalability; and distinguishing between legitimate development activities and malicious acts. Rather than assuming that all anomalies are indicative of attacks, our process focused on aggregating evidence; providing explanation for alerts; and prioritizing based upon risk. In summary, our contribution to the literature was a methodical way to associate software component risks with the processes that produce, build and distribute those components.

II. LITERATURE REVIEW AND THEORETICAL FOUNDATIONS

A. Evolution of Software Supply Chain Security and Cloud Build Monitoring

The use of third party libraries in most modern software systems as well as the reliance upon automated development environments and distributed software delivery methods have made software supply chain security a significant research area. Traditionally, security approaches were focused on identifying vulnerabilities in source code, performing application tests and protecting the underlying infrastructure. However, traditional approaches are not always able to detect vulnerabilities that are introduced into the software development process through compromised dependencies, unauthorized changes to packages used in the development process, or changes made to the build process. With the growing use of cloud-based development platforms, there is now a requirement to provide security mechanisms that address the entire software development life cycle. Secure development frameworks provide best practices for developing secure software including access control, vulnerability assessments, code reviews, release integrity verification, and continuous monitoring. Build provenance provides details regarding how a particular artifact was created, what source materials were utilized to create the artifact, the steps involved in creating the artifact, and the environment in which the artifact will be executed. Both of these types of mechanisms enable organizations to verify if artifacts were created using authorized processes, and whether the creation history of the artifact can be verified. Cloud build logs complement build provenance by providing details regarding operational events that occur during the compilation process, dependency resolution, testing, and publication of artifacts.

However, logs can be created in different locations (services), format inconsistently, and contain insufficient contextual information to conduct effective investigations. Therefore, providing integration between build monitoring and software component analysis is an essential step in increasing visibility into the supply chain. Research in this area should consider how to link separate security observations together into meaningful relationships, without creating excessive alerts for development teams or placing too many processing burdens on them. Therefore, a comprehensive security strategy should involve collecting evidence continuously, interpreting events consistently, and validating software development activities on a regular basis.Another significant development is the integration of automated security controls in DevSecOps environments. Software

composition analysis tools find vulnerabilities or out-of-date dependencies; cloud monitoring systems find suspicious authentications, odd resource accesses, and unexpected workflows. These tools help protect your systems from attacks, however the alerts they generate alone will not explain the context surrounding an event. For example, a dependency scanning tool will find a dependency that has a known vulnerability, but cannot determine if the dependency was added to a system during a suspicious build. A cloud monitoring system will find a suspicious service account, but cannot determine which artifacts were used by the account. When we combine the findings from both types of tools we can find the connections among dependencies, builds, source revisions, and releases. Finding those connections will enable security teams to make better decisions about what to prioritize and provide evidence for incident response. To make this work effectively, we need accurate timestamps, consistent identification, complete dependency inventory lists, and traceability of artifacts. Security researchers also need to be aware of legitimate changes in how developers operate, such as emergency patches, upgrading dependencies, and temporarily changing build settings. Therefore, the theoretical basis of the proposed methodology includes software supply chain risk management, security information correlation, behavioral anomaly detection, and provenance verification. This integration will enable organizations to go beyond reporting vulnerabilities individually and evaluate security in terms of context—evaluating the components used to create a software product and the processes used to build and distribute the software product.

B. Dependency Risk Signals and Contextual Security Correlation

Traditionally, evaluating the risk of using a dependency involves evaluating multiple indicators such as: publically known vulnerabilities in the package; its maintenance status; its version history; whether there are any reported examples of malicious behavior associated with the package; any limitations related to licensing; and whether it meets the security standards set forth by the organization. Development teams can use information from common vulnerability databases and software composition analysis systems to determine which packages require remediation. However, simply knowing that a vulnerability exists does not necessarily mean that the associated application can be exploited; similarly, simply because no vulnerability has been reported doesn't mean the associated package can be trusted. There are numerous ways a package could become compromised including unauthorized publication, dependency confusion, typosquatting, suspicious behavior by the maintainers of the package, or unexpected changes in how the package is distributed. Therefore, assessing the risk of utilizing a dependency requires using context-based evidence in addition to evaluating the severity of the vulnerability. Cloud-based build logs can serve as the needed context for assessing dependency risks by showing: when a dependency was obtained; who started the build process; from what repository was the package downloaded; and did the build follow the expected workflow. Cloud-based build logs can then be combined with dependency metadata and inventory lists to help determine if a high-risk package entered into a software product through a legitimate and verifiable process. The proposed methodology treats dependency risk signals as evidence that must be assessed along with build activities, source provenance, and deployment relevance. This methodology will enable developers to conduct a broader evaluation of possible threats while accounting for limited logging and inconsistent metadata that may affect the validity of each observation.

Contextual correlation offers a way to theoretically connect different types of data about the same thing (observations) to provide usable information about potential threats. It can do this through using common characteristics of data such as package identification, version numbers, build identifications, repository links, service account identifications, time stamps, and artifact digests to find connections between what may appear to be unrelated occurrences. Temporal correlation can find unusual patterns of events; for example, if someone changes something in a workflow that they don't normally change and then downloads a dependency they don't typically download and publishes a production artifact, that might indicate there was a problem. Identity-based correlation can find problems with builds that were started by people who aren't allowed to start them or builds that used service accounts that have too much power. Provenance correlation can find discrepancies between the revision of source code that was approved and the inputs used to create the final product.

A risk score can then take into consideration all of those things - dependency severity, build anomaly indicators, provenance confidence, exposure, and potential business impact - to help prioritize which problems need attention first. All risk scores should be able to explain why each problem is being identified as a problem, so that whoever receives the alert knows exactly why it was generated. Additionally, organizations need to have processes in place to verify whether alerts are real problems, document the results of investigations into those problems, and adjust their correlation rules as their development processes change. Ultimately, the success of this method of detecting problems will depend upon the quality of the data supporting the analysis, how well the correlation rules fit the data being analyzed, and whether organizations can separate out actual problems from normal variations in daily operations.

III. PROPOSED FRAMEWORK FOR CORRELATING CLOUD BUILD LOGS AND DEPENDENCY RISK SIGNALS

A proposed model for establishing a comprehensive security architecture that integrates cloud build monitoring, dependency intelligence, software provenance validation, and contextual risk evaluation to detect potential software supply chain attacks has been developed. There are five main steps in the model: data collection, event normalization, dependency assessment, correlation analysis, and risk-based response. Data collection involves obtaining relevant events from cloud-based Continuous Integration/Continuous Deployment (CI/CD) environments, source code management systems, package registries, software composition analysis tools, and artifact repositories. Events that can be collected may include build start times, authentication attempts, workflow modification attempts, download attempts of dependencies, compilation completion times, test completion times, and publishing times of artifacts. Additional data related to dependencies, such as vulnerability identification, package reputation, maintenance history, unexpected version updates, and available provenance evidence can be obtained through dependency intelligence sources. All collected data will be normalized into a single format containing timestamps, package IDs, build IDs, source revision IDs, execution IDs, and artifact digests.

Normalization will allow the system to establish links among evidence provided by various tools and platforms. In cases where data is incomplete, the system will record uncertainty as opposed to assuming that missing data indicates malicious intent. During data ingestion, sensitive information (credentials, personally identifiable information) should either be omitted or properly protected. Secure storage, access control, retention policy, and integrity protection are essential to prevent attackers from modifying the evidence collected by the detection mechanism. The normalized data set serves as the basis for further analysis and will enable security teams to reconstruct software production activity and evaluate relationships among dependencies, build processes, and released artifacts.The second key area of performance of the correlation analysis is accomplished through the integration of dependency risks with operational data collected from the cloud based build environment. Dependency records are first correlated with the builds based on the package name, version, repository location, timestamp and possibly the software bill of material information available. After the initial correlation is completed, the system will determine if any of the dependencies that have been identified were created within the scope of an approved process; also if the associated build identity, source revision and package source are in compliance with the organizations policies.

For example, a dependency created with a known critical vulnerability would receive a higher priority if it was downloaded from an unapproved registry as part of a build initiated by a non-standard service account. On the other hand, if a dependency was added immediately following a non-standard workflow change, and was then included in a production artifact, this would warrant further investigation. A risk score can be assigned to a particular dependency based upon its severity, behavioral anomalies, inconsistencies in provenance, exposure to deployments, and potential operational impacts. This can be expressed conceptually as $R = w_vV + w_bB + w_pP + w_eE$, where V represents a normalized vulnerability risk, B represents build behavior anomalies, P represents provenance issues, E represents exposure issues, and the weights represent the organization's priorities. It is important to normalize each component to a common scale, and calibrate and validate the weights prior to operational use. Initially, correlation rules can explain clearly why a build or dependency is flagged as suspicious. However, as the volume of activity increases, statistical or machine learning approaches may be utilized to identify unusual activities occurring across multiple builds.

The last part of the process translates correlation results into proportional security responses and ongoing improvements. High-confidence findings may result in temporary release restrictions, verification of additional artifacts, isolation of affected build jobs, and/or revocation of compromised credentials (with appropriate authorization and recovery processes). Medium-risk findings may require manual reviews, dependency updates, and/or additional provenance checks; low-risk observations can be documented for future trend analysis and investigations. Automated blocking should be used only for appropriately validated policies as over-restrictive policies could impede legitimate development efforts. After an incident is identified, organizations should identify affected artifacts, trace downstream deployments, collect relevant logs, evaluate the extent of dependency exposure, and, if required, rebuild software from verified sources. The framework should also enable continuous evaluation through metrics including detection precision, recall, false positive rate, mean time to investigate, and the percentage of released artifacts with verifiable provenance. Comparative experiments can assess whether combining correlation detects threats that are missed by either individual vulnerability scanners or cloud monitoring systems. Further testing should assess performance in high-event volume environments, with incomplete logs, in dynamic dependency ecosystems, and with evolving attacker behavior. As there are no experimental metrics presented in this conceptual chapter, the effectiveness of the framework is proposed and will need to be empirically validated.

IV. IMPLEMENTATION STRATEGIES AND OPERATIONAL CONSIDERATIONS

A. Cloud Build Log Collection and Event Normalization

The first step of implementing a successful security program is to gather event data that are related to security from the various components of the software development life cycle (SDLC). Examples of these components include: cloud-based build services, source code management (SCM) systems, continuous integration/continuous delivery (CI/CD) systems, package registries, and artifact storage systems. Examples of events that would be of interest to gather include: when a build was initiated; authentication attempts; changes to workflows; dependencies downloaded; failed compilations; permissions modified; and when artifacts were published. It is important to collect as many relevant attributes as possible for each event, such as: timestamps; build IDs; executing identity; reference to the SCM repository; and the digest value of the artifact being produced, if applicable. Because there are many different types of platforms used in the SDLC, each with its own way of formatting and naming log entries, it will be necessary to create an event normalization layer to provide a uniform format for the data collected. Two of the most critical aspects of event normalization are to ensure that the time stamp values are synchronized across all platforms, and that reliable identifier mapping occurs. This is essential for correlating events that occur on multiple platforms so that the sequence of events can be accurately reconstructed. The collection infrastructure used to capture events from the SDLC should be configured with authenticated connections, least-privileged access control, secure storage, and retention policies appropriate for the organization. All sensitive information (e.g., access tokens, credentials) should be removed or obfuscated prior to being processed. Organizations should also track when logs fail to be captured or when event sources are missing, as incomplete log files can negatively impact the effectiveness of any subsequent investigations. Creating a standardized approach to collecting log files provides the baseline required to correlate events reliably, as well as providing full visibility into software production activities.

B. Dependency Intelligence and Risk Classification

Dependency intelligence provides organizations with the ability to determine if there are any known vulnerabilities, suspect behaviors or compliance issues introduced by the use of software components. Organizations should implement dependency intelligence by integrating the findings of Software Composition Analysis (SCA) tools with other data sources including, but not limited to: Package Metadata; Vulnerability Databases; Software Bills-of-Materials (SBOMs); Package Integrity Information; and Provenance Attestations (if available). Every dependency used should be identified based upon trusted attributes, i.e., package ecosystem, package name, package version, and cryptographic digest (if applicable). When classifying risks associated with each dependency, organizations must take into account the severity of each vulnerability identified within that dependency; the degree to which those vulnerabilities can be exploited; the current state of maintenance of each dependency; the source of each dependency; and whether or not each dependency will ultimately reside within a production service. For example, a development-only dependency which contains a vulnerability would likely need to be handled differently than a dependency containing an exploitable vulnerability which will reside in a production environment. Risk assessments conducted against each dependency should be reviewed regularly due to the fact that new vulnerabilities are being disclosed continuously and package reputations can change over time. Organizations must also develop processes for identifying whether a specific indicator is truly malicious or simply has incomplete metadata or is otherwise unrated. Conducting risk assessments in conjunction with build records allows organizations' security teams to identify exactly when and how potentially hazardous components were introduced into the development process, thereby allowing for improved prioritization and resolution planning.

C. Correlation Algorithms and Contextual Risk Scoring

The Correlation Engine correlates dependency risk information to build activity to find dependencies that have a high probability of being compromised, and which traditional security tools don't pick up on. For example, the engine can find unexpected package downloads, unauthorized modifications to a workflow, unusual activity by a service account, or publishing an artifact after a suspicious change was made to one of its dependencies. Temporal analysis will establish the sequence of events; while Identity-Based correlation will link activity to users, service accounts, and permissions. Provenance Analysis will compare artifact metadata to approved source code revisions and expected build configurations. A Contextual Risk Score can be calculated by combining vulnerability severity, anomalous behavior, discrepancies in provenance, and deployment exposure through the use of normalized indicators and calibrated weights. The engine must create explainable alerts that provide the underlying evidence, affected components, and suggested investigative steps. Machine learning techniques can be used as supplements to the rule-based approach, provided that enough representative training data exists. However, machine learning should never replace transparent security controls unless there is adequate validation.

V. COMPARATIVE ANALYSIS OF SOFTWARE SUPPLY CHAIN SECURITY MECHANISMS

Software supply chain security is best accomplished through a variety of complementary security measures designed to detect vulnerabilities, monitor the build process for suspicious activity, and verify the authenticity of the final product. Most traditional security systems operate independently in specific areas (e.g., dependency vulnerability detection; cloud infrastructure monitoring; source code analysis; or artifact verification). While each of the individual security measures provide value to protecting the organization, operating each measure independently may prevent the identification of more complex attacks that involve the relationship of software components to the build process. A correlation based framework will address this issue by integrating dependency risk data with cloud build log data; provenance record data; and behavioral context indicator data. Table 1 compares six key security mechanisms based on their main function; input data; detection capabilities; limitations; and contributions to the proposed framework. This comparison is conceptual in nature and is meant to demonstrate the complementary nature of the six mechanisms rather than providing empirical measurements of their effectiveness. Combining the six mechanisms will provide additional investigative context; enable organizations to prioritize security alerts; and create a unified method for protecting software throughout the entire development and distribution process.

Table 1: Security Mechanisms and Their Roles in Software Supply Chain Protection

Security mechanism	**Primary function**	**Input data**	**Detection capability**	**Key limitation**	**Role in proposed framework**
Dependency vulnerability scanning	Identify known component vulnerabilities	Package names, versions, vulnerability databases	Detects vulnerable dependencies	Limited visibility into suspicious build behavior	Supplies dependency risk indicators
Cloud build log monitoring	Monitor build activities and events	Build logs, timestamps, identities, workflow events	Identifies unusual build operations	May not determine dependency exploitability	Supplies operational evidence
Software bill of materials (SBOM)	Inventory software components	Package metadata, versions, component relationships	Identifies included dependencies	Inventory alone does not establish runtime exploitation or trustworthiness	Connects components with affected artifacts
Software provenance verification	Verify software origin and production history	Source revisions, build identities, attestations	Detects provenance inconsistencies	Depends on trustworthy evidence and verification policies	Validates build and artifact relationships
Behavioral anomaly detection	Identify deviations from established activity patterns	Historical logs, identity activity, build behavior	Highlights unusual actions and sequences	Can generate false positives for legitimate changes	Identifies suspicious behavior for correlation

Comparative analysis reveals that no one mechanism can provide full visibility into software supply chain risks. While dependency scanning uncovers known vulnerabilities; build monitoring allows organizations to understand how their software is built and delivered. Additionally, SBOMs create inventories of components used in software products, and provenance verification creates evidence regarding the source of artifacts as well as the build process. Behavioral anomaly detection uncovers deviations from anticipated behaviors; however, its results must be evaluated within context to determine if they are legitimate or potential threats. The proposed framework integrates these various capability sets through use of common identifiers, temporal relationships, dependency metadata, and contextual risk scoring. As an illustration, if a package containing a

vulnerability is introduced via a non-standard workflow and is subsequently included in an artifact developed by an un-authorized build identity, then that package will likely be assigned a higher priority alert. On the other hand, if a suspicious event occurs, but there is verified evidence indicating that it was caused by a legitimate maintenance activity, then that event may be assigned a lower priority. In addition to collecting accurate data, developing explainable correlation rules, establishing appropriate access control mechanisms, and continuously assessing performance, successful implementation requires that organizations consistently evaluate the efficacy of the proposed framework. Future empirical studies should evaluate the proposed framework against each individual security mechanism based upon representative attack scenarios and typical development work loads. Evaluations such as these can demonstrate whether correlation increases detection rates, decreases the amount of time required to investigate events, and enhances artifact integrity without adding unacceptable operational overhead.

VI. SECURITY CHALLENGES, RISK MITIGATION, AND FUTURE RESEARCH DIRECTIONS

Organizations will face many technical and operational challenges as they attempt to adopt the use of cloud build log correlation with dependency risk signaling to secure their software supply chains. The first challenge is the fragmentation of security information among various cloud platforms, development tools, package registries, and artifact repositories. The information used by each platform may contain different formats, timestamps, identifiers, and retention policies; consequently, it can be extremely challenging to create a chronological sequence of all the software production activity that occurred during the process. In addition, the correlation accuracy can be negatively affected by missing logs, inconsistent dependency metadata, and/or incomplete provenance records. Another challenge is the rapid evolution of software dependencies. After a software build occurs, new vulnerabilities can arise, malicious packages can be discovered, versions can change, and/or maintaining behaviors can evolve, thereby altering the previous assessment of risk.

Therefore, organizations should implement standardized event schemas, reliable time synchronization, automated dependency inventories, and continuous intelligence updates. Organizations should also protect the integrity of the data collected since attackers could modify or delete build logs, hiding suspicious activity, and potentially impeding investigations. Access controls should be implemented, immutable logging when possible should be utilized, cryptographic verification should be performed, and secure storage should be used to provide confidence in the reliability of evidence collected. Finally, organizations should develop clear guidelines for protecting sensitive source code, credentials, employee identifiers, and proprietary build metadata. By overcoming these challenges, organizations can create the operational basis for effective security correlation and incident investigation.Another area that presents a significant challenge is differentiating legitimate operational modifications (such as changing the pipeline configuration; adding/updating/rotating/migrating packages/dependencies/credentials/cloud environments) from potentially malicious activities. Because development teams frequently make operational changes, a single unusual occurrence does not necessarily represent a supply chain attack. However, excessive sensitivity of detection mechanisms could lead to excessive numbers of false positive reports, resulting in increased analyst workload and delayed response to real threats. On the other hand, excessively relaxed detection mechanisms could allow for low-profile attacks to go unnoticed. The proposed framework will attempt to find a balance between these two extremes through contextual risk scoring; transparent correlation rules; environment-specific baseline metrics; and risk-based response procedures.

Alerts will describe what characteristics of dependencies, builds, identities, or provenance discrepancies caused the alert to be classified as such. Security teams will also verify automated decisions prior to taking any disruptive actions (i.e., releasing or blocking production software). Additionally, scalability is a major concern for larger organizations who produce millions of build events and dependency observation points across their distributed development environments. To minimize computational overhead, organizations should employ efficient indexing, event filtering, incremental processing, and analysis prioritization. Organizations should also consider the robustness of models used in machine-learning applications; the quality of training data used to train such models; the explainability of model outputs; and how models perform over time as development practices evolve. Humans will still play an important role in interpreting ambiguous evidence and adjusting detection policies. Through these approaches, organizations can enhance the practical reliability of detection mechanisms while maintaining that security controls do not impede, but rather enable, software delivery.

Future studies should assess the proposed methodology in a variety of ways; i.e., in laboratory settings, through attacks that resemble real-world situations, and by assessing performance in different cloud-native development environments. Examples of relevant scenarios include compromised dependencies, unauthorized changes to workflows, dependency confusion, suspicious package downloads, stolen build credentials, and published artifacts that lack proper documentation of their origins. The assessment should evaluate the benefits of correlating events with dependencies compared to evaluating them independently

(i.e., using dependency scanners) and conventional cloud log analysis. This evaluation will use metrics such as precision, recall, false-positive rate, mean time to detect, mean time to analyze, and computational overhead. Researchers should also determine whether and how incomplete logs, previously unidentified malicious packages, and the fast-moving nature of dependencies influence the performance of the proposed method.

A third avenue of research would involve creating a graph-based model of all the elements involved in developing software — developers, identities, repositories, dependencies, build jobs and released artifacts. This would allow for better tracking of the attack path and provide analysts with the ability to find hidden links between compromised components and the software that uses those components. Additionally, there are many opportunities to incorporate verifiable attestations and standardized software supply chain metadata into automated policy enforcement. In addition, future systems may wish to explore privacy-preserving collaborations where organizations can share relevant threat indicators without revealing any confidential development information. However, the viability of each approach must be established via reproducible experimentation and independently verified data sets. Through exploring these potential areas of study, researchers can develop a stronger basis for the real-world utility of correlating build activity with dependency risk. Ultimately, the objective is to create scalable, understandable, and adaptable security mechanisms that enhance the transparency of the software development process, enhance the integrity of the resulting artifacts, and support trustworthy delivery of software throughout the entire lifecycle of the supply chain.

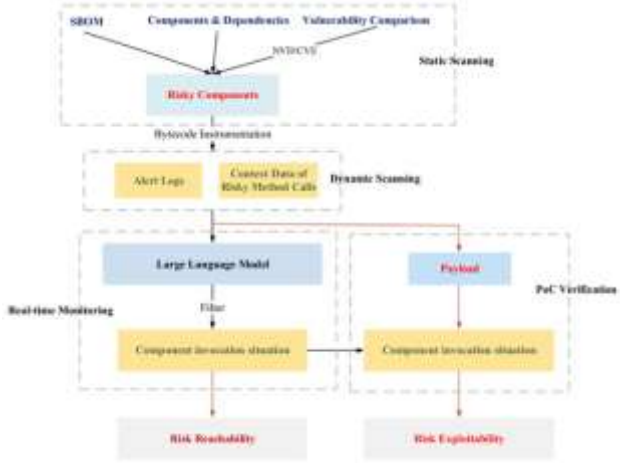


Figure 1: Workflow for Dependency Risk Assessment And Reachability Analysis

VII. PERFORMANCE EVALUATION FRAMEWORK AND EXPECTED SECURITY OUTCOMES

An evaluation of the effectiveness of a correlation-based software supply chain security framework will require a formal evaluation plan that assesses the extent to which the framework is able to detect suspicious activity, assess risk and provide timely response to security incidents. An evaluation of the framework should take place within a controlled cloud-based development environment that includes source code repositories, automated build pipelines, package registries, dependency scanners, and artifact storage. Event data relevant to security will be collected from both normal development activities and simulated attacks. Such attacks may include unauthorized changes to workflows, introduction of vulnerable dependencies, download of suspicious packages, compromise of service identity, and publication of unverified artifacts. The event data collected will be analyzed by the proposed correlation engine and compared to results produced by other types of dependency scanner and conventional build logging tools. Evaluation datasets will vary based upon the size of dependencies used, frequency of builds, severity of vulnerabilities, and operating environment, which will allow for analysis of the performance of the framework under various representative workloads.

Ground truth labels for each event will be determined through documentation of tests performed and verification of event records as opposed to relying on anomaly score determination alone. This methodology will help establish whether the integration of operational and dependency-related evidence enhances the ability to identify significant threats while maintaining adequate insight into legitimate software development processes.The evaluation of the system must include the following measures: detection precision (the percentage of generated alerts that relate to actual simulated threats), recall (the percentage of known test threats detected), false-positive rate (the number of legitimate actions identified as being suspicious), detection latency (the amount of time from when a test event occurs until an alert is generated that allows action to be taken),

investigation time (how much time is needed for analysts to find related dependencies, what artifacts were exposed, and what caused the incident), and computational overhead (the amount of processing time, storage used, and resources consumed during higher volumes of events). Experiments that compare the effectiveness of the individual security tools with the overall integrated framework should be done using the same data sets, detection thresholds, and testing environments. Additionally, ablation experiments can be performed to remove specific components of the framework (such as provenance verification or behavioral analysis) to see how each component contributes to the overall detection performance. Appropriate statistical methods and repeat trials should be used to determine the significance of any observed differences. As no experimental evaluations have been performed in this conceptual paper, there are no reported performance gains or empirical results.

The intended results of this assessment are to provide a better understanding of how contextual correlation affects the detection of software supply chain threats and operational responses. The framework can help analysts identify individual security weaknesses versus possible coordinated attacks by correlating vulnerabilities in dependencies with the identity of the build, changes to workflows, origin of packages and artifact provenance. Additionally, explainable alerts may enhance the efficiency of investigation processes through providing the necessary evidence and affected software components in a single location. However, the success of the benefits provided by the framework will rely upon the quality of the collected logs; the accuracy of the dependency inventories; the completeness of the provenance records; and the appropriateness of the correlation rules. In addition, performance of the framework may vary based on organizations' use of different cloud architectures, development practices and security maturity levels. Therefore, subsequent evaluations of the framework should involve multiple types of platforms, realistic development work loads and previously unobserved attack patterns. Additionally, researchers should assess if automation of response policies reduces exposure without negatively affecting release cycles. Finally, a documented evaluation process, documented dataset, and transparency regarding limitations will increase the scientific value of subsequent studies. Ultimately, empirical evaluation will verify if the proposed framework improves detection capabilities, investigation processes and software release assurance compared to traditional security methods.

VIII. INTEGRATION, DEPLOYMENT, AND GOVERNANCE OF CORRELATION-BASED SECURITY

A. Integration with Existing DevSecOps and Cloud-Native Environments

Integrating the proposed framework with an organization's existing DevSecOps environment will depend on its ability to integrate with the organization's existing development tools, cloud services, and automated deployment processes. An organization should integrate the correlation engine with their source code management systems (SCMS), continuous integration systems (CIS), software composition analysis (SCA) tools, vulnerability databases (VDBs), and artifact repositories (ARs) via either secure API connections or supported event-streaming mechanisms. When integrating, the organization should maintain their current development processes while adding verification checkpoints at key points throughout the development lifecycle such as when retrieving dependencies; building the product; signing artifacts; and releasing products. Policies regarding what dependencies are allowed; what builds are allowed; how long evidence is needed to prove origin; and when human review is required can be defined by the organization's security team. When first deploying the framework, it should run in "observation" mode to identify baseline behavior patterns and evaluate whether the alerts generated are useful prior to applying enforcement policies to production releases. After validating the framework, an organization can apply gradual enforcement policies to high-confidence security violations.

Standardized software bill of materials (SBOMs) and verified provenance attestations will help create better links among an organization's dependency inventory; build process; and published products. Finally, organizations can use centralized dashboards to view all findings related to the organization's development process; affected components; risk classification; and suggested remediation actions. A successful integration will rely on reliable delivery of events; consistent identification of individuals involved in the development process; secure configuration of the framework; and coordination among an organization's development; operations; and security teams. Organizations should perform regular testing of their integrations; review permissions; and document recovery processes. Organizations can incorporate correlation capabilities into their existing workflow so that protecting the software supply chain becomes an ongoing development responsibility instead of a separate security function performed solely before release.

B. Security Governance, Accountability, and Policy Enforcement

Organizations must create formal governance models to hold individuals accountable for managing dependencies, verifying build integrity, accepting risks, and responding to incidents in a timely manner. To do this, organizations need to clearly identify the roles of developers, platform engineers, security analysts and software release managers so that each person understands

how they will be held responsible for addressing identified risks. Organizations should also develop formal approval criteria for accepting packages into their builds; these criteria should include approved package registries, timeframes for remediating vulnerabilities found in those packages, requirements for verifying package provenance (i.e., verifying the source of the package), and methods for managing packages that are either unsupported or suspected of containing malware. Organizations should implement least-privileged access controls when assigning identity to builds, and limit access to logs, artifacts, and security configuration data to only those employees who require it based upon their job functions. Additionally, organizations should maintain an audit trail of all security alerts, investigation results, policy exceptions, and remediation efforts through their correlation framework. When organizations choose to accept a risk, they should document the reasons why the risk was accepted, obtain approval from the proper level of management, and establish a deadline for reviewing the accepted risk to prevent the acceptance from becoming permanent vulnerabilities.

Finally, organizations must consider how they will address the risks associated with third-party service providers and external dependencies since supply chain risks can extend beyond an organization's internal environment. By conducting regular security audits, incident response drills, and compliance assessments, organizations can evaluate if their current policies continue to effectively protect against evolving technology-based threats. Performance metrics should measure the speed of vulnerability remediation, the percentage of packages that have been verified as coming from a trusted source (provenance), the number of unresolved high-risk dependencies, and the outcome of investigations into security incidents. Most importantly, organizations should use automated risk scores to inform employee decision making instead of replacing employee decision making when dealing with uncertain or significant risk. By combining technical monitoring with transparent governance, organizations can establish a sustainable operational model that protects software development processes, enhances organizational resilience, and instills confidence in software artifacts throughout their entire lifecycle.

IX. ADVANCED ANALYTICAL TECHNIQUES FOR SOFTWARE SUPPLY CHAIN PROTECTION

A. Graph-Based Correlation and Attack Path Reconstruction

Graph-based correlation is a useful technique to analyze the relationships between software dependencies, developers, cloud identities, build pipelines, source repositories and release artifacts. This technique represents each of the above entities as a "node" within the graph, and the relationship between them as an "edge" (e.g. dependency inclusion, build execution, package retrieval, artifact generation, deployment). Analysts can create a graph using normalized cloud build logs and dependency intelligence, and then examine the influence that each event had on the security of a whole software product. For instance, if an analyst examines a graph created by graph-based correlation, they might find out that an unauthorized service identity changed a build workflow, retrieved a package never used before, generated an artifact and then distributed that artifact to production environments. This example demonstrates how analysts can see the bigger picture when examining related events rather than just analyzing individual alerts. Graph-based correlation also enables analysts to discover indirect exposures, such as a vulnerable dependency that was part of an intermediate library that was subsequently included in many applications. Nodes and edges in a graph can have temporal attributes associated with them so that analysts can establish sequence of events and distinguish between past and current activity. Analysts can use graph traversal, relationship scoring, and anomaly detection to determine which paths in the graph are most suspicious. However, the reliability of a graph depends on reliable identification of all entities, completeness of dependency inventories, and sufficient details of build logs. If analysts miss any relationship in the graph, they will end up with incomplete attack paths. In addition, if analysts attempt to build a graph for a large software ecosystem, they may encounter computational issues. Therefore, analysts should focus on representing relevant entities, keeping track of evidence, and periodically updating their graphs. Graph-based correlation supplements traditional vulnerability scanning by connecting technical weaknesses with real production activity.

B. Machine Learning and Behavioral Anomaly Detection

Machine learning can augment traditional rules-based analysis of cloud build activities and dependencies by identifying oddities in those builds. A behavioral analysis system can develop models of what constitutes normal behavior for each type of build (e.g., how often it runs, what service accounts are used, which dependencies are fetched, how long it takes to complete, what sources are used to create the build, etc.). If a build deviates significantly from its baseline behavior, this could be indicative of some form of unauthorized activity. In addition to assessing whether the build has deviated from the norm, the system would also need to assess whether the resulting build contains any high-risk dependencies. If both conditions are met, further investigation would likely be required. An example would be if a build begins downloading dependencies from an unknown registry, and runs as a user who has never run a build before. These types of behaviors should warrant further investigation. There are two primary approaches to applying machine learning in this context: supervised learning and unsupervised learning.

Unsupervised learning can be utilized to identify unusual builds without requiring labeled data. However, as stated earlier, unusual behavior is not always synonymous with malicious behavior. As software development processes continue to evolve, what was once considered normal behavior may now be perceived as unusual. Therefore, machine learning systems should include contextual information, explain why they flagged certain builds as suspicious, and receive ongoing evaluations based on changing workload characteristics. Additionally, the training data sets must include examples from a wide variety of project types, dependency ecosystems, and build configurations so as to minimize systematic detection errors. Finally, the performance of the machine learning system should be measured using metrics such as precision, recall, false-positive rate, and detection latency instead of simply relying on accuracy. Security professionals should be able to evaluate alerts generated by the system, and provide feedback that will enable controlled model adjustments. Machine learning should not be used as the sole means of authorizing high-impact actions without adequate controls. When combined with deterministic security rules, provenance verification, and contextual risk scoring, behavioral analytics can provide broader detection capability while preserving transparent verification mechanisms and accountable incident response processes.

C. Automated Risk Prioritization and Intelligent Remediation

Risk prioritization using automation allows security professionals to prioritize those aspects of the software supply chain that have the highest potential for harm. All vulnerabilities and/or anomalous build events should not be given equal treatment. The proposed model utilizes dependency severity, exploitability, build characteristics, confidence regarding the provenance of the software being built, degree of exposure in terms of where the software is deployed, and business relevance to the organization as inputs to determine the risk level of the dependency. An urgent response would likely be required for a high-risk vulnerability present within a dependency utilized by an Internet-facing production system; conversely, a low-risk vulnerability within a non-production component would be evaluated at a different risk level. Likewise, a medium-risk dependency could potentially merit rapid investigation if it is introduced via an unauthorized change to the workflow or if there was suspicious behavior related to downloading packages. To enable effective decision-making regarding risk assessment, the risk prioritization process needs to generate understandable reasons for why certain decisions were made based upon evidence and policies.

Remediation options available to organizations include but are not limited to upgrading vulnerable dependencies; replacing dependencies that cannot be trusted; limiting the number of locations from which packages can be downloaded; verifying signatures on artifacts; revoking compromised credentials; and/or re-building software from trusted source material. Organizations should begin implementing automated processes slowly, first providing recommendations and notifications and then implementing limited types of enforcement actions. Decisions that have significant consequences, such as preventing software from releasing into production or shutting down key build systems, should be made based on sufficient evidence and authorizations. Finally, organizations should use this model to track remediation efforts, outstanding issues, and repeat security issues so they can continually improve their security practices. Organizations must validate the efficacy of this model through controlled experiments designed to measure if risk prioritization results in fewer investigative hours spent by analysts, shorter periods of time between discovery and remediation, and no increased disruption to operations. By combining risk-based scoring with well-defined remediation procedures, organizations can achieve greater consistency in responses to incidents, minimize exposure windows, and increase the integrity of software releases throughout their diverse development ecosystems.

X. RISK ASSESSMENT AND MITIGATION STRATEGIES FOR CORRELATED SOFTWARE SUPPLY CHAIN THREATS

Software supply chain security is a methodical approach to evaluating and addressing the risks created by the relationship between the development environment used in the cloud and the use of third party dependencies. While there are several types of security tools (such as vulnerability scanners and cloud monitoring), each type offers its own perspective on security issues. Therefore, organizations must develop a single, integrated risk assessment method to evaluate how different types of vulnerabilities could affect both the creation and delivery of software products. This paper proposes a methodology that integrates dependency intelligence, build-event analysis, provenance verification, and risk-scoring based on context to assess potential dangers resulting from relationships in the software supply chain. In assessing the risk of a security incident, organizations should take into account the severity of a vulnerability, the credibility of the source of the package being used, the permissions associated with a build identity, the validity of workflow configurations, and the availability of affected packages. These considerations allow organizations to decide if they need to immediately investigate, verify further, or remediate an issue. The table below lists examples of common software supply chain threats, the corresponding indicators, possible impacts, and recommended remediation strategies. This table serves as a baseline for developing correlative security controls within cloud-native environments.

Table 2: Software Supply Chain Threats, Risk Signals, and Mitigation Strategies

Identified threat	Relevant risk signals	Potential impact	Mitigation strategy	Verification method
Vulnerable dependency introduction	Known vulnerability, affected package version	Exploitation of application weaknesses	Upgrade or replace the affected component	Dependency rescan
Malicious package installation	Suspicious package origin, unusual download activity	Unauthorized code execution	Restrict registries and verify package integrity	Package provenance review
Build identity compromise	Unusual authentication, unexpected privileged actions	Unauthorized software modification	Apply least privilege and rotate credentials	Identity and access audit
Pipeline configuration manipulation	Unapproved workflow changes, abnormal build sequence	Compromised build artifacts	Require configuration reviews and protected branches	Workflow integrity verification
Artifact provenance inconsistency	Missing attestations, unexpected source revision	Distribution of untrusted software	Verify signed provenance and artifact digests	Attestation validation
Dependency confusion	Unexpected registry selection, package name collision	Retrieval of attacker-controlled components	Configure trusted registries and namespace controls	Registry and dependency-source audit

The table illustrates how different types of supply chain threats require separate detection and mitigation strategies. For instance, dependency-related vulnerabilities are most effectively addressed through assessing the components of your dependencies and then remediating those vulnerabilities as quickly as possible. On the other hand, the threat posed by installing malicious packages is best handled through better control over registries used to store packages, and verification that the packages have not been tampered with. If an attacker has compromised the identity of a build system, it would be appropriate to use identity monitoring techniques, limit permission levels, and implement secure methods for managing credentials. Additionally, when there is evidence of pipeline manipulation, a combination of protection of configuration files, mandatory reviews, and verification of authorized workflow changes may be used to reduce risk. Similarly, if there are discrepancies in provenance, a combination of creating reliable artifact attestations and validating that the relationship between source code and generated artifacts is correct may be used to reduce risk. Finally, dependency confusion risks can be minimized through explicit registry configuration, namespace protection, and controlled package resolution.

XI. CONCLUSION

Software supply chain security has evolved into a vital element of contemporary software engineering. The interconnections created by cloud-based development environments, automated build pipelines and third party dependencies provide multiple avenues for possible security compromises. In this study we investigated the value of associating cloud build logs with dependency risk indicators to enhance our ability to recognize, evaluate and mitigate potential security threats within the software supply chain. While traditional security measures typically assess dependency vulnerabilities, cloud activity and artifact integrity independently, there may exist relationships that could expose much larger attacks. Our proposed framework attempts to resolve these limitations by combining event collection, log normalization, dependency intelligence, provenance verification, behavioral analysis and contextual risk scoring. By combining all of these elements organizations will be able to determine not only if a software component contains a known vulnerability but also how it came into a project, who introduced it into a project, what build activities took place prior to the creation of the software component, and if the final product was created using the established production procedure. Our framework provides a theoretical basis for increasing the visibility of the software production process and enhancing security controls throughout the entire software development and distribution lifecycle.

The researchers found that each of these mechanisms (cloud build monitoring, software bills of materials, dependency vulnerability assessment, provenance verification, and behavioral anomaly detection) has a unique role to play; however, as an integrated group, they provide a greater amount of security context. For instance, if there is a known vulnerability in a particular

dependency, this will be of greater concern to organizations if the dependency is used in combination with unapproved workflow modifications, suspicious package downloads, or inconsistencies with the artifact's provenance. Graph-based correlation can assist in identifying connections among developers, identities, repositories, dependencies, and published artifacts. Machine-learning methods can find anomalous patterns that would otherwise go unnoticed by traditional detection rules. Risk scores based on context can enable organization to prioritize investigations based on technical evidence, deployment exposure, and potential business impacts. However, successful use of the framework requires that organizations collect reliable event data, create accurate dependency inventories, verify provenance information, and appropriately configure detection rules. The framework must also consider false positives, missing log data, evolving developer practices, privacy regulations, and computational resource constraints. Therefore, integrated correlation should enhance existing security controls rather than substitute for basic vulnerability management, access control, secure development, and artifact verification practices.

Operationalizing the proposed model will help organizations better coordinate their security programs. It does so through linking detection, investigation, remediation, and continual improvement. Organizations can add correlation features to existing DevSecOps practices. They can then add automatic responses based on how confident they are in the risk level of the threat. Some actions that may be taken in response to potential threats include limiting access to unknown packages; confirming that the artifacts have valid signatures; reviewing any unrequested changes to workflow; safeguarding build identities; and, if the software has been compromised, rebuilding it using only verified source material. Organizations need to create policies regarding responsibility for validating alerts, accepting risks, escalating incidents, and tracking remediations. Evaluations of the effectiveness of the system should include assessment of detection accuracy, recall, false positive rates, time required to detect, time required to investigate, amount of provenance information available, and amount of overhead used. This paper describes a possible conceptual framework; however, it does not provide evidence of its efficacy. Thus, we cannot assume that using this approach will result in better performance than prior methods. Additional research using controlled experiments, reproducible data sets, realistic attacks simulated in a laboratory environment, and comparison with other security tools will be required to demonstrate whether the integrated model results in measurable improvements when applied in real-world environments.

Future studies should examine the effectiveness of the framework across various types of cloud platforms, dependencies ecosystems, software architectures, and organizational settings. Special focus should be placed on unmapped malicious packages, compromised build credentials, indirect dependencies relationships, and attacks designed to avoid traditional detection methods. Examples of areas that may provide opportunities for future research include graph-based investigations, explainable machine-learning models, standardized provenance attestations, and privacy-preserving threat intelligence sharing. Additionally, researchers should consider the impact of correlation mechanisms under conditions such as incomplete logging data, rapidly changing dependency information, or high-volume event generation from development workloads. Further research could also address adaptive risk thresholds and carefully controlled automated remediations that do not cause undue disruptions to valid software releases.

XII. REFERENCES

[1] Adams, B., De Schutter, K., Tromp, H. and De Meuter, W. (2007) 'The evolution of the Linux build system', Electronic Communications of the EASST, 8.

[2] Allamanis, M., Barr, E.T., Devanbu, P. and Sutton, C. (2018) 'A survey of machine learning for big code and naturalness', ACM Computing Surveys, 51(4), Article 81.

[3] Bass, L., Weber, I. and Zhu, L. (2015) DevOps: A Software Architect's Perspective. Boston, MA: Addison-Wesley Professional.

[4] Böhme, M., Szekeres, L. and Woo, M. (2017) 'Transparent and efficient vulnerability assessment for software supply chains', in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM.

[5] Cárdenas, A.A., Manadhata, P.K. and Rajan, S.P. (2013) 'Big data analytics for security', IEEE Security & Privacy, 11(6), pp. 74–76.

[6] Cloud Native Computing Foundation (2021) Software Supply Chain Best Practices. San Francisco, CA: Cloud Native Computing Foundation.

[7] Cox, B., Collins, J., Goel, S., Hart, M., Jensen, A., Lenz, B., Lindner, M., McIntosh, S., Nguyen, J. and Rafiq, M. (2021) Securing the Software Supply Chain: Recommended Practices Guide for Developers. Washington, DC: National Security Agency.

[8] Decan, A., Mens, T. and Grosjean, P. (2019) 'An empirical comparison of dependency network evolution in seven software packaging ecosystems', Empirical Software Engineering, 24, pp. 381–416.

[9] Dey, A.K. (2001) 'Understanding and using context', Personal and Ubiquitous Computing, 5, pp. 4–7.

[10] ENISA (2021) Threat Landscape for Supply Chain Attacks. Athens: European Union Agency for Cybersecurity.

[11] European Union Agency for Cybersecurity (2023) ENISA Threat Landscape 2023. Athens: ENISA.

[12] Fenton, N.E. and Neil, M. (2018) Risk Assessment and Decision Analysis with Bayesian Networks. 2nd edn. Boca Raton, FL: CRC Press.

[13] Garfinkel, S.L. and Rosenberg, B. (eds.) (2005) RFID: Applications, Security, and Privacy. Upper Saddle River, NJ: Addison-Wesley.

[14] German, D.M., Adams, B. and Hassan, A.E. (2013) 'The evolution of the R software ecosystem', in Proceedings of the 17th European Conference on Software Maintenance and Reengineering. Piscataway, NJ: IEEE.

[15] Ghasemisharif, M., Ramesh, A., Checkoway, S., Kanich, C. and Polakis, J. (2018) 'O Single Sign-Off, where art thou? An empirical analysis of single sign-on account hijacking and impersonation attacks', in Proceedings of the 27th USENIX Security Symposium. Berkeley, CA: USENIX Association.

[16] Humble, J. and Farley, D. (2010) Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley.

[17] ISO/IEC (2022) ISO/IEC 27001:2022 Information Security, Cybersecurity and Privacy Protection—Information Security Management Systems—Requirements. Geneva: International Organization for Standardization.

[18] Kula, R.G., German, D.M., Ouni, A., Ishio, T. and Inoue, K. (2018) 'Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration', Empirical Software Engineering, 23, pp. 384–417.

[19] Le, D.M., Link, D., Shahbazian, A. and Medvidovic, N. (2015) 'An empirical study of architectural decay in open-source software', in Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture. Piscataway, NJ: IEEE.

[20] McGraw, G. (2006) Software Security: Building Security In. Boston, MA: Addison-Wesley.

[21] National Institute of Standards and Technology (2022) Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication 800-218. Gaithersburg, MD: NIST.

[22] National Institute of Standards and Technology (2022) Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. NIST Special Publication 800-161 Revision 1. Gaithersburg, MD: NIST.

[23] National Institute of Standards and Technology (2022) Guide to Computer Security Log Management. NIST Special Publication 800-92. Gaithersburg, MD: NIST.

[24] Open Source Security Foundation (2023) SLSA: Supply-chain Levels for Software Artifacts, Version 1.0. San Francisco, CA: Open Source Security Foundation.

[25] Pashchenko, I., Plate, H., Ponta, S.E. and Sabetta, A. (2018) 'Vulnerability detection in open-source dependencies with minimal effort', in Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings. New York: ACM.

[26] Perl, H., Dechand, S., Fahl, S., Acar, Y., Backes, M. and Smith, M. (2015) 'VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits', in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York: ACM.

[27] Scarfone, K. and Mell, P. (2007) Guide to Intrusion Detection and Prevention Systems (IDPS). NIST Special Publication 800-94. Gaithersburg, MD: National Institute of Standards and Technology.

[28] Souppaya, M., Scarfone, K. and Dodson, D. (2022) Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication 800-218. Gaithersburg, MD: National Institute of Standards and Technology.

[29] Torres-Arias, S., Afzali, H., Kuppusamy, T.K., Curtmola, R. and Cappos, J. (2019) 'in-toto: Providing farm-to-table guarantees for bits and bytes', in Proceedings of the 28th USENIX Security Symposium. Berkeley, CA: USENIX Association.

[30] Zimmermann, M., Staicu, C.-A., Tenny, C. and Pradel, M. (2019) 'Small world with high risks: A study of security threats in the npm ecosystem', in Proceedings of the 28th USENIX Security Symposium. Berkeley, CA: USENIX Association.