Improving software development life cycle using data-driven approaches

Florent Moriconi

THESE DE DOCTORAT DE SORBONNE UNIVERSITE

École doctorale EDITE de Paris n°130

Spécialité « Sciences de l'information et de la communication »

Amélioration du cycle de développement logiciel via des approches basées sur les données

Thèse présentée et soutenue à Biot, le 14 juin 2024, par

Florent MORICONI

Président	**Prof. Elena Cabrio**, Université Côte d'Azur
Rapporteur	**Prof. Coen De Roover**, Vrije Universiteit Brussel
Rapporteur	**Prof. Maurizio Morisio**, Politecnico di Torino
Examinateur	**Prof. Jean-Rémy Falleri**, Enseirb-Matmeca/Bordeaux INP
Examinateur	**Dr. Melek Önen**, EURECOM
Directeur de thèse	**Prof. Aurélien Francillon**, EURECOM
Co-encadrant	**Dr. Raphäel Troncy**, EURECOM,
Co-encadrant	**Dr. Jihane Zouaoui**, AMADEUS

To everyone who supported me along the journey of this thesis.

Acknowledgements

I would like to express my gratitude to all who have supported me throughout my PhD journey. I am incredibly grateful to my supervisors, Prof. Aurélien Francillon, Dr. Raphäel Troncy and Dr. Jihane Zouaoui, for their advice, ideas and continuous support throughout the years. The initiation of this thesis in April 2020 coincided with challenging circumstances, yet through the dedicated efforts of Dr. Jihane Zouaoui and René Jullien, the project was successfully launched a few months later. Over the past three and a half years, I have worked in an exceptional research environment, supported by both Amadeus and Eurecom. Their provision of invaluable resources and data has been instrumental in my research journey. I am especially grateful for the insightful and enriching discussions with colleagues and friends from both institutions, spanning not only technical matters but also broader topics that have contributed to my personal and professional growth. A special thanks to my family and my girlfriend for their unwavering support, encouragement, and constant inspiration throughout this journey. To everyone mentioned here — thank you!

Biot, 14 June 2024 ***Florent Moriconi***

Abstract

This thesis explores data-driven approaches for automated root cause analysis of CI/CD build failures, focusing on identifying non-deterministic failures, locating root cause messages in build logs, and characterizing CI/CD systems' performance and security. Grounded on public and industrial datasets, we explore CI/CD workflow properties, such as execution times and failure patterns. The research introduces the use of Natural Language Processing (NLP) and Knowledge Graphs Embeddings (KGE) for classifying build failures with a 94% accuracy. Additionally, we introduce ChangeMyMind, a new method based on Recurrent Neural Networks (RNNs) to accurately locate root cause messages in build logs without prior labeling of root cause messages. We propose X-Ray-TLS, a generic and transparent approach for inspecting TLS-encrypted network traffic in CI/CD environments. Finally, the thesis also revisits security vulnerabilities in CI/CD systems, demonstrating the potential for undetectable long-term compromises. This work has resulted in three publications and two under-review submissions, contributing significantly to CI/CD system analysis and optimization.

Résumé

Cette thèse explore les approches basées sur les données pour l'analyse automatique des causes profondes des échecs de construction dans les systèmes d'intégration continue et de déploiement continu (CI/CD), en se concentrant sur l'identification des échecs non déterministes, la localisation des messages de cause profonde dans les journaux de construction, et la caractérisation de la performance et de la sécurité des systèmes CI/CD. Basée sur des ensembles de données publics et industriels, nous explorons les propriétés des flux de travail CI/CD, telles que les temps d'exécution et les modèles d'échec. La recherche introduit l'utilisation du traitement du langage naturel (NLP) et des embeddings de graphes de connaissances (KGE) pour classifier les échecs de construction avec une précision de 94%. De plus, nous introduisons ChangeMyMind, une nouvelle méthode basée sur les réseaux neuronaux récurrents (RNNs) pour localiser avec précision les messages de cause profonde dans les journaux de construction sans étiquetage préalable des messages de cause profonde. Nous proposons X-Ray-TLS, une approche générique et transparente pour inspecter le trafic réseau chiffré TLS dans les environnements CI/CD. Enfin, la thèse revisite également les vulnérabilités de sécurité dans les systèmes CI/CD, démontrant le potentiel de compromissions à long terme indétectables. Ce travail a abouti à trois publications et deux soumissions en cours de révision, contribuant de manière significative à l'analyse et à l'optimisation des systèmes CI/CD.

Contents

Chapter 1

Introduction

The importance of software engineering in the contemporary technological landscape cannot be overstated; serving as the backbone of innovation, efficiency, and competitive advantage across industries. At Amadeus IT Group, a leading provider of IT solutions to the global travel and tourism industry, software engineering plays a central role in achieving business objectives. From booking flights to managing hotel reservations, Amadeus enhances operational efficiency and significantly improves the end-user travel experience. This has profound implications for the travel sector and the economy, demonstrating how software engineering drives sectoral advancements and global connectivity.

As businesses strive for faster time-to-market and superior product quality, software development frameworks, such as Continuous Integration and Continuous Deployment (CI/CD), facilitate automated testing and deployment, enabling swift feedback and iterative improvements. However, the complexity of these systems also introduces challenges, particularly in diagnosing and addressing failures. Automated root cause analysis emerges as an essential tool in this regard, allowing teams to quickly identify and rectify issues, thereby sustaining the pace of innovation and maintaining the reliability of services.

1.1 Software Development Life Cycle

Contemporary software development methodologies like DevOps aim to compress the system development life cycle (SDLC) to enable regular and secure updates. This approach seeks to synchronize feature deployment with business goals and facilitate rapid response to incidents (for instance, GDPR mandates prompt remediation of data breaches). The DevOps philosophy advocates for Continuous Integration (CI) to minimize the feedback loop, ensuring developers receive immediate notifications of issues arising from code modifications.

Finding software defects sooner rather than later significantly reduces the development cost.

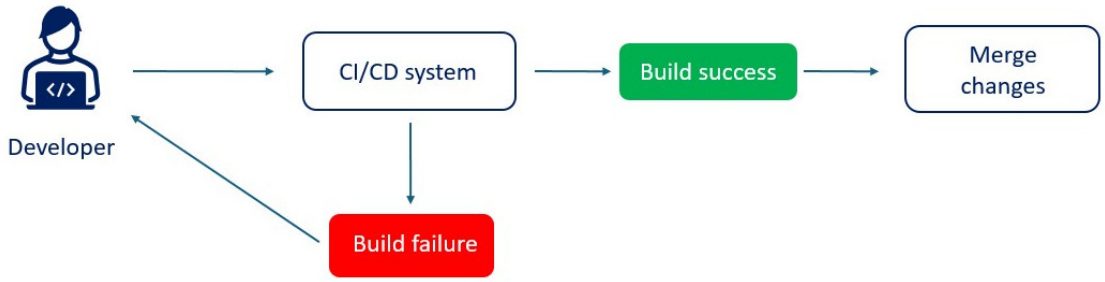

Figure 1.1: Illustration of the CI/CD feedback loop.

Early detection typically occurs before the defective code is deeply integrated into the system, making fixing it easier and less costly. As the software progresses through its development lifecycle, the complexity of addressing defects increases due to dependencies, additional code, and integration points, leading to exponentially higher costs and effort for correction. Moreover, late discovery of defects can lead to cascading issues affecting other components, potentially necessitating widespread revisions or redesigns, further escalating costs and delaying project timelines. IBM's research [31] highlights the escalating financial repercussions of rectifying a bug at various stages of the Software Development Life Cycle. Specifically, the cost of resolving a bug during the implementation phase is approximately sixfold higher than in the design phase. This cost increases up to 15 times during the testing phase and can surge to 100 times during the maintenance phase. This trend underscores the exponential growth in the financial impact of bugs as they progress through the development cycle. Consequently, adopting a shift-left approach has become a paramount priority for numerous organizations, aiming to address defects early in the development process to mitigate these escalating costs.

Continuous Integration/Continuous Deployment (CI/CD) facilitates the "shift-left" approach by integrating security and testing early in the software development lifecycle. Figure 1.1 illustrates the interplay between developers and a CI/CD system. The process initiates when developers commit new code to a source code management system, such as Git. This action typically triggers an automatic CI/CD build process to validate the quality of the newly submitted code. If the new code meets the established criteria, such as passing unit tests and adhering to coding standards, it is approved for integration into the primary codebase. Conversely, if the code fails to meet these requirements, the build is marked as unsuccessful. Consequently, developers are prompted to address the issues before integration into the main codebase can proceed. By automating the integration and deployment process, CI/CD enables developers to detect and fix bugs, vulnerabilities, and integration issues more rapidly and frequently. In the following section, we delve into the details of CI/CD.

1.2 Continuous Integration and Continuous Deployment

Continuous Integration (CI) focuses on the automated integration of code changes from multiple contributors into a single software project. In contrast, Continuous Deployment (CD) extends this automation to include the release of the software to production environments. In practice, CI (Continuous Integration) and CD (Continuous Deployment/Delivery) workflows are frequently deployed within the same orchestrator, utilizing analogous tools. Consequently, numerous characteristics inherent to CI pipelines, such as failure categories, also apply to CD pipelines. In the following, we refer to CI or CD workflows as CI/CD workflows.

Continuous Integration (CI) systems adhere to common architectural patterns across the industry. We present below their major properties:

- **Pipeline as Code**: Workflow tasks are defined in code files (e.g., YAML files in GitHub Actions). This allows for version control, peer review, and history tracking of the pipeline configurations.
- **Event-Driven Triggers**: CI/CD systems often use event-driven triggers to initiate workflows. These events can include branch events (e.g., git push) or pull requests. In addition, CI/CD systems usually allow trigger runs manually or based on a schedule.
- **Modular Design**: Workflow definitions are designed to be modular, allowing easy integration of off-the-shelf tasks (e.g., Actions in Github Actions). This lets developers easily integrate tasks common to many code repositories, lowering the maintenance burden.
- **Parallel and Sequential Execution**: To optimize performance and reduce build times, CI/CD systems support parallel execution of tasks where feasible and sequential execution for tasks that depend on other tasks.
- **Isolation**: CI/CD systems often rely on containerization or virtualization to isolate build environments, ensuring consistency across different executions.
- **Feedback Loops and Notifications**: They include mechanisms for feedback and notifications, providing real-time alerts on the status of builds, tests, and deployments. This helps teams quickly address issues and maintain high-quality codebases.

Table 1.1 compares widely available major CI/CD systems, focusing on their licensing, ownership, and operational environments. GitHub Actions and GitLab offer cloud and self-hosted (SH) options for control and operation, with GitHub Actions being proprietary and owned by GitHub (Microsoft). At the same time, GitLab uses both MIT and proprietary licenses and is owned by GitLab Inc. Under an MIT license, Jenkins is community-owned and operates exclusively in self-hosted environments. Travis CI, proprietary and owned by Idera, Inc., operates

Name	License	Owning Company	Controller	Runners
GitHub Actions	Proprietary	GitHub (Microsoft)	Cloud, SH	Cloud, SH
GitLab	MIT, proprietary	GitLab Inc.	Cloud, SH	Cloud, SH
Jenkins	MIT	N/A (Community)	SH	SH
Travis CI	Proprietary	Idera, Inc.	Cloud	Cloud, SH
Drone	Apache 2.0, proprietary	Harness	Cloud, SH	Cloud, SH

Table 1.1: Comparison of major CI/CD systems. Gitlab and Drone propose both open-source and commercial versions. All, except Jenkins, are owned by companies. Controllers and runners can be provided as a service only (Cloud) or managed by users (Self-Hosting, abbreviated as SH).

on the cloud but supports cloud and self-hosted runners. Drone offers a dual license (Apache 2.0 and proprietary) and is owned by Harness, providing both cloud and self-hosted options for control and operation.

We now focus on the architecture employed by Amadeus, a world-class software engineering firm. The orchestrator at the heart of their CI process is Jenkins, an MIT-licensed tool developed in Java. In practice, developers typically configure one or several jobs per software project to integrate into the CI workflow. Each job is associated with a Git repository and is designed to run automatically upon new commit pushes or the opening of pull requests. The job's operations are specified in a JenkinsFile, a configuration document outlining the required actions. To promote efficiency and reuse, workflows—collections of predefined steps—are commonly leveraged, facilitating the sharing of a standard procedure among jobs. This approach is particularly intriguing because it suggests that jobs utilizing the same workflow likely perform analogous tasks, ranging from compiling source code to running unit tests and deploying files to remote servers.

A build is the execution instance of a job, meaning a job can accumulate multiple builds over time. It is standard practice to manage storage to retain only a recent subset of builds, such as the latest ten. The outputs of commands executed during the build, namely `stdout` and `stderr` in Linux, are captured in a build log. For clarity, a Jenkins build and its corresponding log are referred to interchangeably, given that each build produces a unique log. Build durations vary significantly, from seconds to hours, depending on the configured actions. A build is deemed failed if any command within it exits with a non-zero status, halting the process immediately. Failures can stem from legitimate reasons, such as unmet prerequisites, coding standard violations, or transient issues like network disruptions or disk space shortages. Pinpointing the exact cause of a build's failure, referred to as root cause analysis, often demands manual investigation, which can be complex and time-consuming. Although build failures are an inherent aspect of the CI process, illegitimate failures introduce unnecessary delays as developers seek to identify and rectify these issues, frequently finding them to be ephemeral or unrelated to their direct responsibilities. Consequently, there is a pronounced interest in

developing automated tools for the root cause analysis of build failures, offering a path toward more efficient resolution and minimizing the manual effort required from developers.

CI/CD logs differ in many settings from production or system logs. In the next section, we will focus on the specificities of CI/CD logs.

1.3 CI/CD build logs specificities

Continuous Integration/Continuous Deployment (CI/CD) logs and system logs serve different purposes and thus exhibit distinct characteristics. CI/CD logs focus on the software development lifecycle, capturing detailed information about software projects' build, testing, and deployment processes. These logs provide insights into the success or failure of automated tasks, compile errors, test results, deployment statuses, and version control events, facilitating the debugging and optimization of the CI/CD pipeline.

On the other hand, production logs are records generated by software applications and systems during their operation in a production environment, which is the live setting where users actively interact with the application. These logs contain detailed information about the system's activities, errors, transactions, and performance metrics, capturing the real-time functioning and issues of the software. Unlike CI/CD logs, which document the build and deployment process in development pipelines, production logs focus on the actual running of the software over long periods. This distinction is crucial as production logs help monitor the application's health, troubleshoot problems, and optimize performance based on live usage data rather than the development and deployment stages.

Figure 1.2 shows an example of a build log from the Jenkins CI/CD orchestrator. In the same log, there are log messages from different software. However, there is no clear separation between the different parts of the log that would allow easy identification of which software the log message is coming from. As each software has its logging format, this significantly increases the complexity of log parsing, as later described in Section 2.3.1.

Table 1.2 summarizes the key differences between the logs. The table provides a comparative analysis between CI/CD logs and production logs across three distinct properties. Firstly, CI/CD logs are generated by many independent programs (e.g., scripting, CLI tools, test frameworks), whereas production logs usually originate from a single program. Secondly, the number of log messages per log event (i.e., the template used to generate the log message) in CI/CD logs is low, unlike production logs, which exhibit a high volume of log messages per log event. This is because CI/CD logs usually contain many programs that run for a short period. Lastly, the time frame for CI/CD logs is restricted to the duration of the build process, typically measured in minutes, whereas production logs are continuous, operating 24/7. While

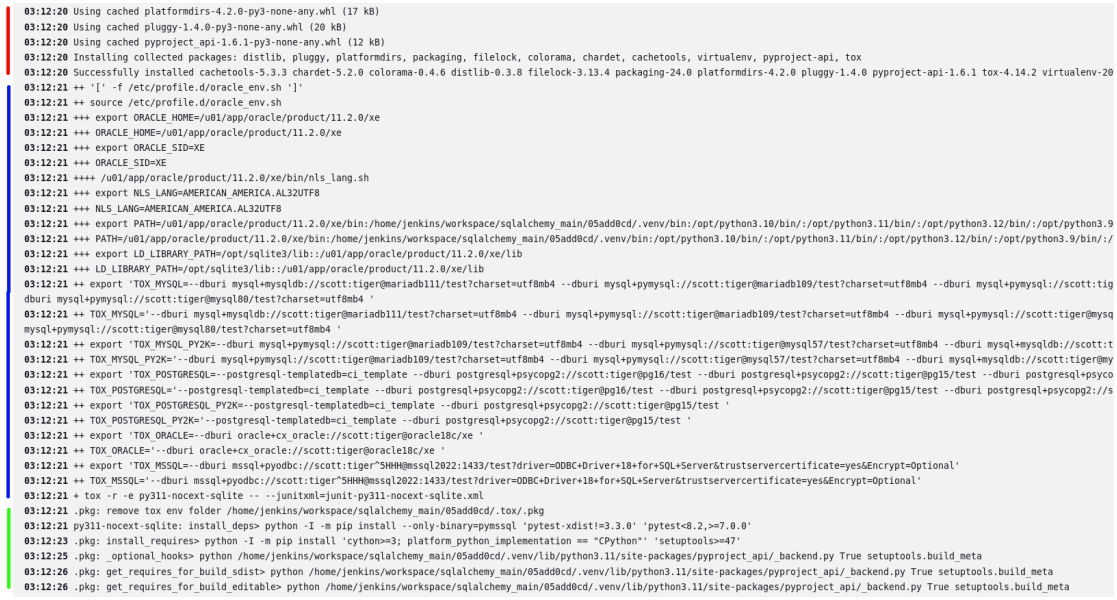

Figure 1.2: Example of a build log from Jenkins. There are log messages from different software: the red part comes from pip, the blue part from the shell, and the green part from tox.

Properties	CI/CD Logs	Production Logs
# independent programs	Many	One
# log messages per log event	Low	High
Time-frame	Build duration (mins)	24/7

Table 1.2: Comparison between CI/CD Logs and production logs.

CI/CD logs are essential for developers and DevOps teams to streamline software delivery, production logs are crucial for system administrators to ensure system stability and security. CI/CD logs significantly differ from production logs, making it difficult or impossible to use log analysis methods developed for production logs.

1.4 Understanding CI/CD failures

In the context of Continuous Integration and Continuous Deployment, understanding build failures is crucial for maintaining the efficiency and reliability of the software development process. Build failures in CI/CD can generally be categorized into several types, each with distinct characteristics and implications. We propose to distinguish two classes of failures: deterministic failures and non-deterministic. Deterministic failures in CI/CD pipelines refer to errors or issues that consistently produce the same outcome under the same conditions. These failures are predictable and can be reproduced by repeating the same steps or inputs. They typically arise from static flaws in the code, configuration errors, or consistent environmental

issues across runs. On the other hand, flaky or non-deterministic failures occur sporadically due to dynamic factors like network instability or resource contention. Below, we present different categories of root causes of deterministic and non-deterministic failures. We also detail failures that can be deterministic or non-deterministic depending on the context.

Deterministic failures include:

- Compilation Errors: If the source code contains syntax errors or other compilation issues, it will consistently fail to compile. This type of error is deterministic because the failure occurs every time the compiler encounters the problematic code.
- Linting Errors: Code linting tools enforce coding standards and styles. If the codebase violates these rules, linting tools will report errors. These failures are deterministic, as they occur every time the code with the same violations is linted.
- Configuration Errors: Misconfigurations in the CI/CD pipeline, such as incorrect environment variables, paths, or permissions, can lead to consistent failures. These errors are deterministic because they produce the same failure whenever the pipeline runs with the faulty configuration.
- Security Vulnerability Detection: If a pipeline includes steps to detect security vulnerabilities, and such vulnerabilities are present in the code, it will lead to deterministic failures. These checks produce consistent results when the same codebase is scanned.

Deterministic failures can often be easily replicated, making them relatively easy to troubleshoot and fix. On the other hand, non-deterministic failures can also occur, presenting more complex challenges.

Non-deterministic failures include:

- Infrastructure Failures: These occur due to issues with the underlying infrastructure supporting the CI/CD pipeline. This could involve problems with networking, storage, or the CI/CD tools themselves (e.g., Jenkins, GitLab CI). This category also includes when there are problems with external libraries or services that the software depends on, such as failures in external services that the application relies on (e.g., network timeout).
- Flaky Tests: Tests that sometimes pass and sometimes fail without any changes to the code or test. Flakiness can be due to timing issues, dependencies on external services, or state leakage between tests.
- Resource Contention and Limits: Failures due to resource constraints such as memory, disk space, or network bandwidth. These failures can occur sporadically depending on

the load and the number of processes running concurrently. This might include rate limiting, downtime, or changes in the external service that were not anticipated.

- Race Conditions: Issues that occur when the outcome depends on the non-deterministic order of execution of threads or processes. Race conditions can lead to failures in multi-threaded or distributed systems where timing and order of operations cannot be guaranteed.

Addressing non-deterministic failures often requires extensive logging and monitoring, using retries and timeouts, isolation of tests, and ensuring consistent environments for testing.

Failures that can be deterministic or non-deterministic include:

- Dependency Failures: Failures attributable to external dependencies, such as third-party services, libraries, or APIs, which exhibit inconsistent behavior, are classified as non-deterministic failures. Conversely, third-party services may exhibit deterministic failures, such as when a resource no longer exists or because of permanent breaking changes.

We have identified three main categories that explain the reasons for failure. While employing additional categories for a more detailed classification is feasible, these categories are often inadequate for comprehensively understanding the root cause of a failure and rectifying the build. Consequently, our attention will now shift towards defining the concept of a root cause message within a build log.

A root cause message in CI/CD build logs is a specific piece of information that accurately identifies the primary reason for a failure in the CI/CD process. Unlike generic error messages or warnings that may appear throughout the build logs, the root cause message pinpoints the exact issue that led to the build failure. These messages are critical for developers and operations teams because they provide the key to resolving the underlying problem and helping to prevent future occurrences of the same issue.

Pinpointing the precise root cause of a failure in Continuous Integration/Continuous Deployment (CI/CD) processes presents a significant challenge. This difficulty arises from the abundance of error messages embedded within CI/CD build logs, as detailed in Chapter 3. These logs often include non-fatal errors—instances where operations such as remote server requests may initially fail, only to be retried successfully later on. Despite being non-fatal to the build, these errors are recorded in the build logs.

It is crucial to differentiate between errors fatal to the build process and those not. Fatal errors are defined by their capacity to halt the build process entirely. However, it is important to

note that fatal errors do not necessarily appear chronologically as the first or last recorded error in the build logs. This is because when a fatal error occurs, multiple components within the system may log this error, leading to a situation where the critical error that caused the build to stop is not the last one logged. Likewise, fatal errors can appear in the log while being successfully retried later in the build, thus not being directly responsible for the failure of the build. Therefore, identifying the fatal log message inside errors logged in the build log is usually not as simple as searching for the first or the last error log message in the build log.

Log messages are the narrative threads that weave together the story of a CI/CD build process. While informative in isolation, each log message gains its true significance when interpreted within the preceding and succeeding entries continuum. This contextual framework is essential because the build process is inherently sequential and interconnected, where each step may affect or be affected by others. The cause of a particular outcome, be it success or failure, often cannot be discerned by examining a single log message in isolation. Rather, understanding the "story" of a build requires piecing together the sequence of events captured in the logs. This narrative approach enables developers and engineers to trace the execution flow, diagnose issues more effectively, and understand the dynamic interactions between different CI/CD pipeline components. However, it also increases the complexity of identifying the fatal error message, as the same log message can be fatal or not, depending on the context.

Providing a versatile definition of a fatal log message is challenging considering the large number of components printing into the build log, such as CI orchestrator, scripting, or tools such as test frameworks. Therefore, we propose to define the location of the root cause message in the build log instead of defining the properties of the log message itself. We propose the following definition:

The root cause message of a build failure is the first log message such that the failure of the build is inevitable.

The definition above allows for identifying the fatal log message's location in the build log. However, in practice, there is no easy way to extract the failure probability of the build for each log message. Thus, this definition is not designed as a practical method for pinpointing the exact root cause messages within the build log but rather aims to establish a conceptual framework. This topic will receive further examination in Chapter 6. It is worth emphasizing that it is not necessarily the first error message in the build log.

In summary, identifying the root cause message helps to understand the reason behind a failure, thereby facilitating its resolution, whether through manual intervention by an operator or via an automated mechanism. The following section delves into the different approaches for root cause analysis of CI/CD builds.

1.5 Root cause analysis

A build failure prompts the necessity for root cause analysis, which assists operators in identifying the fundamental reason for the failure to enable a quick fix. In practice, root cause analysis can be approached in various manners.

1.5.1 Labeling

One prevalent method involves the use of labeling to categorize the build. This process entails assigning one or more labels to a build to denote its failure type, such as "code failure," "instabilities," or "test failure." The granularity of the classification, which depends on the number of labels, differs depending on the classification approaches. For instance, a binary classification can classify code mistakes and infrastructure instabilities (Chapter 4). On the other hand, other approaches can leverage dozens of labels. For instance, approaches that rely on a knowledge base of regular expression often contain dozens of labels to match root causes in build logs. An analogous strategy involves labeling each line of the build log as normal or abnormal, enabling operators to concentrate their investigation on lines marked as abnormal. It is important to distinguish between these two labeling approaches: the former categorizes the entire build, while the latter focuses on the granularity of individual log lines. Labeling abnormal lines has the same objective as fault localization presented in the next section: reduce the search space for root cause messages from the entire logs to a subset of the log.

1.5.2 Fault localization

Fault localization methods are critically useful in automated root cause analysis of CI/CD failures, primarily because logs generated during CI/CD processes are often voluminous and complex, making manual inspection impractical and error-prone. These methods leverage algorithms and machine learning techniques to sift through the extensive build log, effectively identifying the location of the build log that is likely to contain the root cause message. We propose a definition of the root cause message in Section 1.4. By automating pinpointing the precise location within the code or deployment pipeline where a failure has occurred, fault localization tools aim to significantly reduce the time and effort required for diagnosis, enabling faster resolution of issues. Locating a failure in a build log is illustrated in Figure 1.3, where the red rectangle marks the root cause's location. Depending on the chosen method, fault localization techniques may pinpoint single or multiple lines as potential root causes. For these techniques to be effective, it is crucial that the number of lines identified as potential root causes remain significantly lower than the total number of lines in the log, thereby ensuring a substantial reduction in the volume of data requiring analysis. Ultimately, methods of fault

localization aim to facilitate the resolution of failures. In contrast, labeling strategies at the build level are more readily adapted for automated remediation, such as executing automated retries for specific labels. Fault localization methods can effectively reduce the amount of unnecessary information in the further analysis of the build log, whether this analysis is conducted through manual inspection or automated categorization. This is achieved by removing parts of the log that are not pertinent to understanding and fixing the failure. We propose a fault localization approach in Chapter 6. Labeling abnormal lines and fault localization approaches are similar. However, they differ as labeling abnormal lines can flag lines anywhere in the log, while fault localization is expected to flag an area of the log; therefore, lines must be contiguous.

1.5.3 Fix proposal

A more surprising yet effective approach to root cause analysis is the provision of corrective actions or fixes directly. In this scenario, rather than merely identifying the root cause, the algorithm proposes a specific fix, such as a code patch or an action (e.g., retry). This method represents a proactive stance towards resolving failures, offering developers immediate solutions rather than mere diagnostics. Indeed, root cause analysis is often not useful by itself: the main objective is facilitating the fix generation. Therefore, instead of relying on a two-step process, i.e., root cause analysis and fix generation, providing fixes is an effective way to support correcting build failures.

Each method presents a unique perspective on root cause analysis, highlighting the multi-faceted approaches for root cause analysis for CI/CD build failures. In the next section, we delve into the security of CI/CD systems.

1.6 Security of CI/CD

The security of Continuous Integration/Continuous Deployment (CI/CD) pipelines is crucial because these systems automate the software development process, from code integration to deployment. Given their central role in building, testing, and deploying applications, any vulnerabilities or breaches in a CI/CD pipeline can lead to significant security risks. These risks include unauthorized access to sensitive data, code tampering, and the potential deployment of malicious code into production environments. Moreover, since CI/CD pipelines often access production servers and sensitive credentials, a compromised pipeline can be a gateway for attackers to infiltrate an organization's infrastructure. Ensuring the security of CI/CD pipelines is therefore essential not only for protecting intellectual property and customer data but also for maintaining trust in the software development process and safeguarding the integrity of the delivered applications.

Figure 1.3: Illustration of fault localization in the build log.

1.7 Research questions

This thesis embarks on a comprehensive exploration across multiple research fields, intertwining the disciplines of software engineering, knowledge management, AI interpretability, and software security. As such, research questions span multiple fields.

- **RQ1**: What can we learn from a large corpus of CI/CD build logs, and in particular, can we identify build failure reasons?

  The first research question focuses on gleaning valuable knowledge from a large CI/CD build logs corpus to improve software development methodologies and efficiency. It primarily explores the feasibility of automatically identifying and classifying the causes of build failures. By analyzing the intricacies and recurring patterns present in build logs on a large scale, this research aims to deepen the comprehension of the processes and tools employed within CI/CD pipelines. Such an analysis can facilitate more efficient troubleshooting strategies, pinpoint areas of infrastructure bottlenecks or misconfiguration, and identify optimization opportunities.

- **RQ2**: How can CI/CD build failures be classified automatically?

  Automatic classification of CI/CD build failures refers to categorizing build failures into predefined categories defining code mistakes, configuration errors, or transient failures. This process typically requires the processing of build data to extract relevant features, such as error messages, code changes, or build environment details, which can then be used to train a model to recognize patterns associated with different types of failures. The ultimate goal is to enable automated identification of the nature of a failure without human intervention, facilitating quicker troubleshooting, automated fixes, or monitoring failure trends at scale.

- **RQ3**: How to define and locate root cause messages in CI/CD build logs?

  Labeling failed builds offers valuable insights into failure categorization, yet attaining detailed classification can be difficult due to the potential need for many labels. An alternative approach to assist operators in resolving issues involves extracting log messages from the build logs that aid in repairing the builds. The initial challenge lies in defining root cause messages, identifying them, and understanding their primary characteristics. The subsequent challenge involves automating the recognition of these log messages, as manual identification proves to be labor-intensive and impractical for scaling, especially when dealing with thousands of daily build failures.

- **RQ4**: Are logs sufficient to detect CI/CD compromise?

  The integrity, confidentiality, and availability of software are significantly influenced by the security of CI/CD pipelines, as these pipelines are instrumental in automating

the software development and deployment processes. A CI/CD pipeline breach can enable attackers to embed vulnerabilities or backdoors into the software, posing risks of data breaches, unauthorized access to sensitive information, and potentially allowing for further exploitation of the system's infrastructure. While logs from CI/CD processes offer crucial insights, their effectiveness in assuring the security of CI/CD systems raises questions. This prompts an investigation into the degree to which log data can offer assurances of security for CI/CD systems.

1.8 Thesis structure and contributions

This thesis comprises eight chapters, each focusing on research challenges in log analysis, knowledge graphs, network security, fault localization, and CI/CD security. This thesis leads to 3 publications:

- F. Moriconi, R. Troncy, A. Francillon, and J. Zouaoui, "Automated Identification of Flaky Builds using Knowledge Graphs," in International Conference Knowledge Engineering and Knowledge Management (EKAW), Poster Session, 2022.
- F. Moriconi, O. Levillain, A. Francillon, and R. Troncy, "X-Ray-TLS: Transparent Decryption of TLS Sessions by Extracting Session Keys from Memory," in 19th ACM ASIA Conference on Computer and Communications Security (ASIACCS), 2024
- F. Moriconi, A. I. Neergaard, L. Georget, S. Aubertin, and A. Francillon, "Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems," in 17th IEEE Workshop on Offensive Technologies, pp. 219–227, 2023.

The thesis is structured as follows.

Chapter 1 introduces the overall context of the thesis, outlines the research challenges and contributions, and provides a comprehensive overview of the thesis structure.

Chapter 2 offers a critical review of the existing literature relevant to the topics addressed in this thesis.

Chapter 3 discusses the methodologies for collecting and analyzing CI/CD logs from industrial and public datasets.

Chapter 4 examines the application of NLP techniques and Knowledge Graphs in classifying CI/CD build failures. This work leads to a publication [119].

Chapter 5 investigates strategies for the transparent and universal decryption of TLS-encrypted network traffic within CI/CD build environments. This work leads to a publication [117].

Chapter 6 delves into methodologies for identifying root cause messages in CI/CD build logs without relying on a pre-labeled dataset of root cause patterns, leveraging recurrent neural networks and an instrumentation engine.

Chapter 7 scrutinizes the security mechanisms of self-hosted CI/CD systems, focusing on achieving a long-term compromise of these systems without leaving any traces in the source code management systems. This work leads to a publication [118].

In Chapter 8, the thesis concludes by summarizing the research findings, presenting the main conclusions, and suggesting avenues for future work in both the short-term and long-term perspectives.

Chapter 2

Literature Review

This chapter engages in a comprehensive literature review, laying the groundwork for understanding the work developed in this thesis. It delves into critical areas such as software engineering, knowledge management, and computer security, setting the stage for a detailed exploration of CI/CD log collection, analysis, and handling failures within CI/CD pipelines. Furthermore, the chapter extends its focus to encompass knowledge graphs for enhancing log analysis, the principles of TLS inspection, and the emerging threats of malware targeting CI/CD systems.

2.1 Methodology

In this thesis, we explored different research fields, such as software engineering, knowledge management, and computer security. Therefore, we first describe the methodology we applied to all research fields. Then, we focus on every research field. To find and collect relevant publications for this work, we started with keyword searches on Google Scholar. Then, we identify the leading conferences and workshops where high-quality papers are published. Furthermore, we built a curated list of authors with multiple papers published in the field. We then made an exhaustive review of their publications. Finally, we leverage references and "cited-by" references (i.e., papers that cite the paper) to discover relevant work. Indeed, we identified this approach as highly relevant, especially for papers providing datasets. In the following sections, we describe our methodology for each research field.

2.1.1 Software engineering

We started by exploring literature about root cause analysis of CI/CD failures. We used the following keywords: root cause analysis CI/CD, root cause analysis continuous integration, failures CI/CD, classification of continuous integration failures, continuous integration test

failures, software failure root cause analysis, software defect prediction, and software fault localization. Considering many root cause analysis methods heavily rely on logs, we extended our search using log analysis keywords: *log analysis*, *log analysis root cause analysis*, *log analysis CI/CD*, *log analysis continuous integration*, and *log analysis anomaly detection*.

We identified conferences where significant papers are published:

- International Conference on Software Engineering (ICSE)
- International Conference on Automated Software Engineering (ASE)
- International Conference on Software Testing, Validation and Verification (ICST)
- International Symposium on Software Reliability Engineering (ISSRE)
- International Conference on Software Maintenance and Evolution (ICSME)
- Mining Software Repositories (MSR)
- International Symposium on Software Testing and Analysis (ISSTA)
- International Conference on Availability, Reliability and Security (ARES)
- International Conference on Software Quality, Reliability and Security (QRS)

Finally, we profited from an online repository of log analysis papers [1].

2.1.2 Knowledge management

Software engineering, fundamentally concerned with software development, operation, and maintenance, involves complex data management and knowledge processing. Therefore, knowledge management conferences provide useful insights into the latest strategies, tools, and methodologies for efficiently handling, storing, and retrieving knowledge. This is often highly relevant in software engineering topics, where the management of coding knowledge, documentation, user data, software artifacts such as logs, and project management practices are central to the discipline. Additionally, these conferences often showcase emerging trends, innovative technologies, and case studies, offering a rich source of current and practical information that can be applied to software engineering topics. We used the following keywords: natural language processing, text mining, log mining,

We identified conferences where significant papers are published:

[1] https://github.com/logpai/awesome-log-analysis

- International Conference on Knowledge Engineering and Knowledge Management (EKAW)
- International Semantic Web Conference (ISWC)
- Extended Semantic Web Conference (ESWC)

2.1.3 Computer security

In the context of knowledge management, data augmentation refers to the process of enhancing, enriching, or expanding the existing dataset to improve data quality and quantity, which can be crucial for better analysis, decision-making, and machine learning applications. However, in computer systems, information is often protected to provide confidentiality and integrity (e.g., encryption). Therefore, we investigated methods and solutions to extract secured information from software systems, such as network traffic. These approaches are typically presented in computer security conferences.

We identified conferences where significant papers are published:

- IEEE Symposium on Security and Privacy (S&P)
- ACM Conference on Computer and Communications Security (CCS)
- USENIX Security
- Network and Distributed System Security Symposium (NDSS)
- International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)
- ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024)
- Workshop on Offensive Technologies (WOOT)

In the following section, we started by exploring available CI/CD datasets.

2.2 Collection of CI/CD logs

2.2.1 Preliminaries

Datasets containing logs are valuable resources for supporting research that employs data-driven methodologies, such as automated root cause analysis, as they serve as abundant sources of empirical evidence. These datasets, typically consisting of system logs, program

logs, and user activity, encapsulate a wealth of information about the normal and anomalous functioning of systems. When employed in research, they enable the development and refinement of algorithms that can discern patterns, anomalies, and causal relationships within complex systems. This is particularly crucial in automated root cause analysis, where the ability to identify the underlying cause of a problem accurately depends heavily on the quality and comprehensiveness of the available data. Therefore, the availability and accessibility of detailed, varied log datasets are key factors that drive the advancement of research in this field, leading to more sophisticated and accurate diagnostic tools.

Unfortunately, public datasets of logs are rare. Datasets of software logs are rare primarily because of concerns about privacy and security. Software logs often contain sensitive information, including user data, system operations, and potential vulnerabilities, which companies are reluctant to share publicly. Sharing such data could lead to privacy breaches and expose systems to security risks. Additionally, logs can include proprietary information that companies consider competitive secrets. Even when anonymized, logs can be difficult to scrub thoroughly, maintaining the risk of revealing confidential information. Furthermore, the size and complexity of log files make them challenging to manage and share, especially for large-scale systems. These factors contribute to the scarcity of publicly available datasets of software logs. We first investigate datasets of systems logs, and then we focus on datasets of CI/CD logs. In the following sections, we identify the main papers that propose datasets of software logs.

2.2.2 System logs

Several papers have proposed datasets of system logs to support research on software analytics, anomaly detection, and log-based diagnostics.

Xu et al. [188] propose system logs from an HDFS cluster. HDFS (Hadoop Distributed File System) is a distributed file system designed to run on commodity hardware, providing high throughput access to application data and suitable for applications with large datasets. The dataset was created from a cluster running in a private cloud environment using a benchmark workload, and thus, there is no privacy information involved in the dataset. They manually labeled abnormal log lines (e.g., empty packet error message). The dataset is provided in a standard BSD license to support further research. Jingwen et al. [194] propose a similar dataset based on HDFS.

Loghub [195] encompasses 19 diverse, real-world log datasets. We present in Table 2.1 the datasets proposed by LogHub, along with their description, if the dataset is labeled, and the dataset size. These datasets are sourced from a broad spectrum of software systems, including but not limited to distributed systems, supercomputers [129], cloud orchestrators [108], oper-

ating systems, mobile platforms, server applications, and standalone software solutions. We observe that the Hadoop dataset is a job log with some level of similarity with CI/CD logs: it represents an execution of steps and has a status (success/failure). This paper delves into the statistical analysis of these datasets, sheds light on potential practical applications, and shares benchmark results derived from LogHub, such as the performance of log parsing approaches or log compression methods.

System logs and CI/CD logs serve different purposes and, therefore, contain different types of information. System logs are records generated by the operating system or applications running on a server or device, capturing events such as system errors, configuration changes, and other system-related activities. They are crucial for monitoring system health, diagnosing problems, and ensuring security. In contrast, CI/CD (Continuous Integration/Continuous Deployment) logs are specific to the software development process. They record the steps and outcomes of building, testing, and deploying code. These logs provide insight into the development pipeline, tracking changes, build successes or failures, testing results, and deployment status. While system logs focus on the operational aspects of the IT environment, CI/CD logs are centered around the software development lifecycle and the code deployment automation. In the next section, we focus on datasets of CI/CD logs.

2.2.3 CI/CD logs

Compared to system logs presented above, CI/CD datasets are rare. The low number of datasets of CI/CD logs compared to system logs can be attributed to several factors. First, CI/CD processes are relatively newer and more specialized compared to system logging, which is a fundamental aspect of all computing systems. System logs have been around since the early days of computing, resulting in a longer time frame to accumulate datasets. Second, CI/CD datasets are inherently more complex and varied, including many data types such as code repositories, build logs, deployment histories, etc. CI/CD orchestrators, such as Github Actions or Jenkins, do not follow common patterns for logging. This complexity makes it challenging to standardize and collect these datasets at scale. Additionally, the proprietary nature of many CI/CD workflows, which are often closely guarded for competitive reasons, limits the availability of public datasets. In contrast, system logs are more standardized and less sensitive, leading to a broader availability of these datasets. Finally, there's a stronger awareness and established practice of logging in systems for monitoring and troubleshooting, which has encouraged the accumulation of system log datasets over time. In contrast, the practice of analyzing CI/CD processes for data-driven insights is still developing, leading to fewer datasets being available in this domain.

Furthermore, even CI/CD build logs from open-source projects are not easily accessible. For instance, Travis removed their unauthenticated API to fetch logs [171], and GitHub Actions

Dataset	Description	Labeled	Raw Size
Distributed systems			
HDFS_v1	Hadoop distributed file system log	✓	1.47GB
HDFS_v2	Hadoop distributed file system log		16.06GB
HDFS_v3	Instrumented HDFS trace log (TraceBench)	✓	2.96GB
Hadoop	Hadoop mapreduce job log	✓	48.61MB
Spark	Spark job log		2.75GB
Zookeeper	ZooKeeper service log		9.95MB
OpenStack	OpenStack infrastructure log	✓	58.61MB
Super computers			
BGL	Blue Gene/L supercomputer log	✓	708.76MB
HPC	High performance cluster log		32.00MB
Thunderbird	Thunderbird supercomputer log	✓	29.60GB
Operating systems			
Windows	Windows event log		26.09GB
Linux	Linux system log		2.25MB
Mac	Mac OS log		16.09MB
Mobile systems			
Android_v1	Android framework log		183.37MB
Android_v2	Android framework log		3.38GB
HealthApp	Health app log		22.44MB
Server applications			
Apache	Apache web server error log		4.90MB
OpenSSH	OpenSSH server log		70.02MB
Standalone software			
Proxifier	Proxifier software log		2.42MB

Table 2.1: Enumeration of datasets presented by LogHub [195]. The label column indicates if abnormal lines are identified.

requires the user to be logged in to retrieve CI logs from public projects [64] with a strict rate limiting. To our knowledge, LogChunks [18] is a rare exception, proposing a dataset of 797 failed build logs scraped from Travis CI attached to open-source GitHub repositories. The root cause of the failure (i.e., a subpart of the log that could range from one to a few lines) is provided and cross-validated with source code owners. However, LogChunks only provides failed logs, which prevents the use of methods that require both failed and successful samples. Therefore, we attempted to retrieve both successful and failed logs from Travis CI to check if the dataset could be improved to build a large-scale dataset of CI/CD logs. In LogChunks, 85 repositories ranging over 29 programming languages have been scrapped. We tried to retrieve build logs from the same projects. However, only 4 out of the 85 repositories still allow scraping their build logs. Logs can become unavailable because access has been restrained by project owners, because they expired, or because the project does not use Travis CI anymore. We could scrape only 4,500 build logs, which is low compared to the size of system logs presented above.

Numerous studies highlight the widespread adoption and significance of CI/CD. For instance, Hilton et al. [86] noted as early as 2016 that 70% of the most popular projects and 40% of all GitHub projects employ continuous integration. A study examining tests conducted during CI at Google reveals that, on an average day, 800,000 builds and 150 million test runs are executed [113]. The TravisTorrent dataset encompasses logs from millions of Travis CI builds [12], facilitating investigations into the Travis CI platform. More recent Travis datasets, such as those in [18, 47, 48], provide additional metadata with the logs for further analysis. However, Travis is gradually losing popularity among developers, especially for open-source projects.

In conclusion, to our knowledge, there is no large-scale dataset of CI/CD build logs, either labeled or not, for the most used CI/CD orchestrators. Therefore, data-driven approaches for CI/CD, such as automated root cause analysis of build failures, cannot be directly trained and benchmarked on a common dataset. While multiple papers explore CI/CD orchestrators and their properties, they do not propose datasets of CI/CD logs. In the next section, we describe the literature on CI/CD orchestrators.

2.2.4 CI/CD orchestrators

Tingting et al. [24] explore the usage and impact of using GitHub Actions (GHA). They focus on the adoption of GHA for projects with more than 1,000 stars. They show that nearly 22% of projects use GHA. They extract usage insights by parsing workflow files only for Java projects. To identify usage patterns, they focus on sequence analysis of actions defined in workflow files. However, they ignore action versions and shell commands. Furthermore, they do not identify which commands are run by an action (i.e., they only identify actions used). Finally,

they estimate the impact of using GHA on software development efficiency (e.g., time to close issue).

Kinsman et al. [99] explore the use, the evolution over time, and the impact of GitHub Actions. Surprisingly, they found that only 0.7 % of repositories are using GitHub Actions. This low adoption can be explained as the study was done less than 2 years after the public opening of GitHub Actions. They categorize actions into one or more categories among 20 manually curated categories, such as Continuous Integration, Deployment, or Code Quality. They identify if actions are added, removed, or modified over time by extracting commits related to workflow files. They collected GitHub issues (i.e., bug tracking tickets) that discuss GitHub Actions. They found that only 15 % of the discussions were related to technical issues. Finally, they explore the impact of using GitHub Actions on project activity. They found that pull-requests are rejected more often and contain fewer commits when GHA is enabled. In conclusion, they mainly focus on the use and evolution of action steps in workflow.

Delicheh et al. [34] analyze programming languages Actions use. They scraped a dataset of 2,817 actions. Actions can be written in JavaScript (i.e., JavaScript Actions) or any language that runs in a Docker container (i.e., Docker Actions). Composite actions allow the reuse of other actions to reduce code duplication. They showed that JavaScript Actions represent 52.7 % of actions, Docker 36.2 %, and composite actions 11.1 %. They identify the dependencies of Actions, such as npm packages or docker images. For instance, 76 % of the JavaScript Actions included indirect dependencies at a depth of 4, with a median number of dependencies of 8. In conclusion, they showed that Actions often rely on many dependencies, up to hundreds. This significantly complicates the identification of the code that the actions will execute. Therefore, it raises concerns about the security of using Actions. This can be seen as a reason to use shell steps that allow full control of the code executed in CI/CD pipelines.

Saroar et al. [151] explore developers' perception of GitHub Actions through a survey analysis. They found developers prefer Actions with verified creators and more stars when choosing between similar Actions. Developers often switch to an alternative Action when facing bugs or a lack of documentation. They show that developers are developing new actions when no existing Action was available (56.8 %), existing actions are limited in functionality (25 %), existing actions are not performant enough (11.4 %), not sure how to find and reuse Actions (4.5 %), or existing actions were too complex (2.3 %). These results can help to explain why shell steps are used instead of action steps.

Bouzenia et al. [17] propose an empirical study of resource usage and optimization opportunities of GitHub Action workflows. They identify resource usage by category (e.g., Test, Build, Lint) and by trigger (e.g., push, pull-request, schedule). However, compared to the work presented in this paper, they analyzed at the job level, while our approach works at the step level. Analysis at the step level allows us to distinguish the different steps of a job: checkout,

lint, code analysis, compilation, push, etc. Furthermore, they categorized jobs based on their name (30 % of jobs were categorized as "other").

2.2.5 Summary

We delve into the importance of datasets containing CI/CD logs for research in automated root cause analysis and the challenges in their collection. We highlight the scarcity of public datasets due to privacy, security concerns, and the proprietary nature of software logs. We review existing literature on system logs, compare them with CI/CD logs, and discuss the rarity of CI/CD log datasets. In the following section, we focus on log analysis. Indeed, many root cause analysis methods rely on log analysis. Therefore, we first explore the literature on log analysis approaches to allow a better understanding of root cause analysis approaches.

2.3 Log analysis

Almost every software includes a logging capability (record events that occur in the system), a significant source of information for detecting issues or investigating the root cause of failures. Logs contain much more information than other approaches, such as numerical metrics (e.g., CPU load, response time), and are, therefore, very attractive. However, the volume of logs is often too large to be processed by human operators. For instance, a system can produce up to 50 GB of logs per hour [84], representing around 120 to 200 million lines of logs per hour. That excludes any exhaustive human processing. Naive methods for detecting anomalies in logs (like searching for the term 'error' or 'fail') do not work [188] because many lines can contain these terms without actually being an error. Furthermore, log messages are often composed of free text written in natural language by developers. They, thus, are subject to the diversity of vocabulary (e.g., present, detected, connected) and ambiguity (e.g., the server) of natural language. A fatal error in a subsystem is not necessarily a fatal error for the overall system (e.g., thanks to a retry mechanism). Finally, the log level is sometimes incorrectly set: some systems produce logs flagged as errors during normal operation, or real errors are logged as non-error levels (e.g., to avoid triggering alarms).

Logs are often highly imbalanced between normal and abnormal lines, e.g., 99,9% are normal lines in the HDFS dataset [188]. Therefore, there is a huge interest in automating identifying abnormal lines. However, classification problems become particularly challenging on highly imbalanced datasets because the model may become biased towards the majority class, leading to poor predictive performance on the minority class. This imbalance can result in a model that simply predicts the majority class for all inputs, achieving high accuracy overall but failing to correctly identify instances of the less frequent class, which are often of greater interest. Moreover, standard evaluation metrics may not accurately reflect the model's

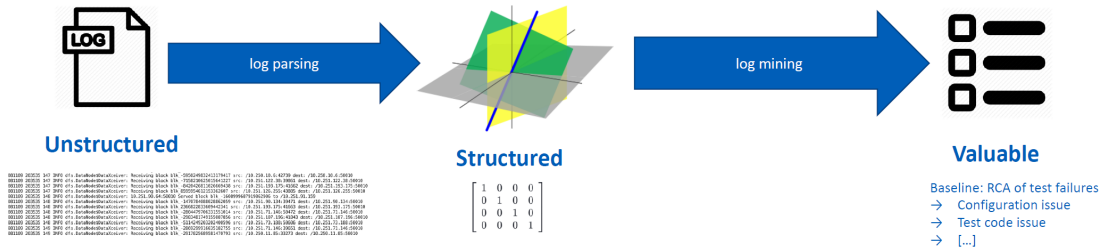

Figure 2.1: Log analysis steps

performance on the minority class, necessitating more nuanced metrics like precision, recall, and the F1-score. The scarcity of data for the minority class also hampers the model's ability to learn its characteristics, further complicating the training process.

Logs contain a large volume of heterogeneous information and, as a result, are difficult to process. Log analysis is usually divided into two challenges: parsing and mining phases. As illustrated in Figure 2.1, the first step (log parsing) is to transform log files (text files) to a structured format (e.g., list of events, matrix). Afterward, classification algorithms can be used to classify (supervised) or cluster (unsupervised) normal and abnormal logs (log mining).

2.3.1 Log parsing

The end goal of log analysis is often to do anomaly detection, failure localization, or failure prediction. To this end, most log mining algorithms require structured logs (sequence of events). A logline is composed of a log header and a log message. The log header comes from the logging library and follows the same format as a given log file. It often contains the timestamp of when the log message was generated, the severity level, and the part of the program from where the log is coming (e.g., module name). The log message is the result of the formatting of the log template with runtime information (Figure 2.3). Figure 2.2 shows an example of a log message with its log template (in bold). Parts of the log message coming from the log template are called constant parts (in blue), while parts of the log message coming from runtime information are called variable parts (in red). As log templates are written by developers during software development, they often do not follow standardized formats. Parsing a log message refers to extracting constant and variable parts of the log message. Achieving good parsing performance without prior knowledge of the log (e.g., source software, log message templates) is challenging. Furthermore, some logs (e.g., Continuous Integration build logs) may contain log messages from various software programs (e.g., shell, maven, helm), leading to heterogeneous log formats.

```
Received block blk_1592043549272047369 of
    size 67108864 from /10.251.105.189
Received block * of size * from *
```

Figure 2.2: Example of message template (in bold).

```
LOG("Received %d bytes", size)  (1)
Received 42 bytes               (2)
```

Figure 2.3: From log statement (1) to log message (2).

Challenges

Commercial software (e.g., Splunk [158]) offers automated log parsing, but only for a limited set of well-known software (e.g., Apache). Users must provide regular expressions to structure their logs for unsupported logs. However, maintaining an up-to-date database of regular expressions (i.e., regex) for log parsing is challenging in practice. As software becomes larger, the number of log templates increases; thus, it often becomes unmanageable to support all log templates. Software updates may imply adding, modifying, or removing log templates. Therefore, the regex database has to be maintained to remain efficient. Determining the set of log templates used in a software is not straightforward, even when the software's source code is available [188] - which is not always the case. Therefore, regexes are usually crafted using log messages generated during software execution, which implies that events must occur to be considered (log messages that rarely happen are unlikely to be included). Logs containing lines from different software are harder to parse, considering the increased number of log templates. Finally, considering the large volume of logs, regular expressions have to be highly specific to avoid wrong event identification.

Log parsers characterization

As proposed by Zhu et al. [196], we characterize parsers based on their operating methods (rule-based, statistical, and based on source code) and their working fashion (online or offline). Online log parsing approaches analyze and process log data as it is generated in real-time. In contrast, offline log parsing approaches analyze log data that has been previously stored and is not being processed in real-time.

Many methodologies (13 are listed in Table 2 [196]) were proposed to transform raw logs into structured form over the years. Log parsing methods can be divided into three categories based on their operating methods:

- Rule-based approaches: users must craft and maintain rules (e.g., regex) to structure logs. These approaches are hard to implement considering the high number of log templates and the fast development of software (e.g., new log template, modified log template).
- Statistical approaches: these approaches leverage the intuition that static parts of log templates should appear more often than variable parts.
- Source-code-based approaches: log messages are generated from log templates written in source code. As a result, log messages could be matched with log templates. However, source code is often not directly available, and using languages with polymorphism increases the complexity of log template extraction.

Zhu et al. [196] compare 13 log parsing approaches released between 2003 and 2018 on 16 heterogeneous datasets. They computed parsing accuracy and robustness (i.e., performance impact w.r.t size and source software) of the log parser, as well as computing efficiency. They conclude that the Drain [82] algorithm has the best accuracy.

Xu et al. [188] propose parsing source code to find log statements. While it works well for C and similar programming languages, object-oriented languages (with polymorphism) are far more complex to parse, as well as corner cases to generate log messages (e.g., complex string concatenation). Obviously, you need to have access to the source code of the running software, which is not always possible.

Unlike traditional methods based on regular expressions, He et al. [82] propose a tree-based approach to cluster log messages online. The logline is tokenized, and then log messages are classified according to the number of tokens and the tokens themselves. Tokens representing variable parts can be excluded by defining regular expressions to avoid branch explosion. These regexes are easy to maintain as they are for variable token format and not the log format (e.g., "blk_[0-9]+" for HDFS block ID). As mentioned in [196], log messages of variable length (e.g., printing a list) have a degraded accuracy.

Dai et al. [29] propose to parse logs by leveraging the occurrence frequency of n-gram. The main idea is that low-occurring n-grams represent variable parts of the log message. The proposed methodology has near-linear scalability and also works online (with negligible accuracy loss). However, methods based on occurrence frequency do not perform well if the variable part has few values. For instance, a variable that indicates different states (e.g., started, stopped) usually has a low number of different values.

Furthermore, log parsing algorithms tend to be highly over-fitted to the category of logs they were designed for. For instance, parsing accuracy (i.e., correct mapping between log messages and log templates) of the Drain algorithm ranges from 0.53 (Proxifier log) to 1

	Methods	Coverage	Preprocessing	Technique
Offline	SLCT [172]	Partial	No	Frequent pattern mining
	AEL [94]	All	Yes	Heuristics
	LKE [58]	All	Yes	Clustering
	LFA [123]	All	No	Frequent pattern mining
	LogSig [164]	All	No	Clustering
	IPLoM [110]	All	No	Iterative partitioning
	LogCluster [173]	Partial	No	Frequent pattern mining
	LogMine [75]	All	Yes	Clustering
	POP [81]	All	Yes	Iterative partitioning
	MoLFI [115]	All	Yes	Evolutionary algorithms
Online	SHISO [116]	All	No	Clustering
	LenMa [153]	All	No	Clustering
	Spell [43]	All	No	Longest common subsequence
	Drain [82]	All	Yes	Heuristics
	Logram [29]	All	Yes	Frequent pattern mining

Table 2.2: Summary of log parsing approaches.

(Apache log) [196]. Thus, it is often necessary to adapt the algorithm according to the log to be processed, which in practice can be long and complex.

Recent software (e.g., Kubernetes) tend to use structured logging to avoid the complexity of log parsing. In structured logging, log messages are often formatted in JSON. As a result, constant parts are the keys of the JSON message, while variable parts are the values. This makes log parsing immediate. Even in structured logging, log messages typically include a "message" field in natural language, which is better suited for human operators analyzing the logs.

2.3.2 Log mining

Structuring logs is insufficient to detect anomalies: log events should be processed to detect anomalies. A log is a text file composed of log lines (separated by cartridge return). Therefore, a log could be seen as an ordered sequence of log lines. However, it is worth noting that the sequence of log messages may not represent the sequence of actions done by the software (e.g., subsystems writing logs by burst). Furthermore, multiple systems may write messages in a unique log, leading to lines coming from independent systems. Log lines before and after a log line are referred to as the context of the log line.

He et al. [83] propose a survey that provides an in-depth look at research in automated log analysis, covering topics such as automating and aiding the creation of logging statements, compressing logs, converting logs into structured event templates, and using logs for anomaly detection, failure prediction, and diagnosis facilitation. Additionally, it reviews research that has released open-source tools and datasets. Concluding with a discussion on recent

advancements, the survey highlights several promising directions for future research in real-world and next-generation automated log analysis.

2.3.3 Semantic approaches

Some approaches take advantage of Natural Language Processing (NLP) algorithms. It looks like a promising path, as log messages are often composed of natural language written by developers for humans (operators or developers). This approach should allow the matching of similar log statements in terms of meaning (e.g., "Server is present" should be close to "Server is detected" and "Server detected: yes"). Furthermore, only a little preprocessing is required (in particular, it avoids the need for parsing logs), which greatly simplifies the practical use.

Bertero et al. [13] focus on detecting stress-related anomalies (system overloading). The goal is to classify whole logs. First, they preprocessed logs by replacing all non-alphanumeric characters with spaces. They trained Word2Vec on a dataset of 660 logs: 330 logs from normal (i.e., not stressed) systems and 330 from manually-stressed systems. They used the Continuous Bag Of Words (CBOW) approach for training, and the embedding space is dimension 20. They showed that the dimension of the embedding space has little impact on performance (Figure 6 in the paper). Word2Vec maps a token to a vector. As a logline contains multiple tokens, they explored two strategies to represent a full log from its tokens: barycenter and TF-IDF[2] (scikit-learn implementation) approaches. The authors note that the aggregation technique has little impact on performance. We describe the TF-IDF approach in Section 4.2.2. They tried Random Forest, Naive Bayes, and neural networks for log mining. The authors mentioned that their work aims to assess the strategy of using NLP to represent logs. Therefore, they used Random Forest Classifier, Multi-Layer Perceptron Classifier, and Gaussian NB from the scikit-learn toolkit with default parameters (i.e., without trying to optimize parameters). Finally, they obtained an Area Under Curve (AUC) of more than 95% using Random Forest and neural networks classifiers, while naive Bayes performed significantly worse. Considering the dataset has been crafted using fault injections (i.e., a finite and determined set of errors) and that the classification is binary, this task should be considered simple.

Aussel et al. [6] propose classifying log lines between normal and abnormal lines. First, they remove stop words except for a set of words that make sense in the context of software engineering: "no", "not", "nor", "on", "off", and "any". Then, they explored two methods to represent log lines: Bag of Words (BoW) and n-gram (with $n = 2$ and $n = 3$). Then, they used bisecting k-means and LDA to characterize log lines before using a classification algorithm. The k-means approach assigns a single label to each line, while for LDA, they assign a vector of 50 topics per line. Finally, they used the Random Forest algorithm with a sliding window of 20 lines. They measure the performance of their methodology on 2 datasets: an industrial

[2]Term Frequency - Inverse Document Frequency

aeronautical system log (non-public) and a public HDFS dataset. On the industrial dataset, which is highly imbalanced (1 failure line for 23,000 lines), they obtained an F-score equal to 0.937 with LDA and bi-gram. It is worth noting that they identified abnormal lines manually (i.e., using regex). Thus, the recall score does not reflect the real number of errors in the log. Furthermore, they did not propose a clear definition of an abnormal line. On the HDFS dataset, they obtained an F-score equal to 0.992. However, they did not mention the ratio of classes and how the abnormal lines were labeled.

Semantic approaches are attractive for their ease of implementation, especially on unstructured logs. While a strong trend exists to write structured logs (e.g., JSON), many software still produce unstructured logs. Furthermore, structured logs often contain a log message written in natural language, thus making natural language processing still relevant.

Du et al. introduce DeepLog [44], a deep neural network model grounded in Long Short-Term Memory (LSTM) architecture, designed to interpret system logs as natural language sequences. DeepLog autonomously learns patterns from normal operational logs, identifying anomalies when deviations from these learned patterns occur. The authors also showcase DeepLog's capability for incremental online updates, allowing the model to adapt to evolving log patterns. Furthermore, DeepLog's ability to construct workflows from system logs is underlined, offering users a powerful tool for diagnosing and analyzing the root causes of detected anomalies. They substantiate their claims with extensive experimental evaluations, demonstrating DeepLog's superiority over traditional data mining methods in log-based anomaly detection. In the following section, we explore the use of Knowledge Graphs.

2.3.4 Knowledge Graphs

The classical log analysis presented above relies on extracting log events from log files. Therefore, all events have the same representation, e.g., a log message is often represented by its log template. Then, supervised or unsupervised log mining algorithms are used to identify anomalies based on the sequence of log events. However, large systems often have heterogeneous subsystems that can be linked together. For instance, a log message from system A can refer to a property of system B (e.g., IP address). Such relations are not easily discovered and leveraged by analysis of log events.

Knowledge graphs, by their inherent structure, facilitate data representation as a network of entities and their interrelations, thus providing a more intuitive and comprehensive perspective of log data. By encapsulating the intricate relationships between various components and events within a system, knowledge graphs pave the way for more insightful and proactive log data management, root cause analysis, and failure prediction. This section delves into the methodologies and benefits of employing knowledge graphs in log analysis, elucidating their

role in transforming raw log data into actionable insights, such as root cause analysis.

Principles

Knowledge graphs are advanced data structures representing information in a graph format, primarily focusing on the interconnections between entities and their attributes. At their core, these graphs consist of nodes representing entities (like people, places, or things) and edges that denote the relationships between these entities. This structure is highly effective for representing complex, interconnected data in a way that is both intuitive and machine-readable. One of the key features of knowledge graphs is their ability to incorporate various types of relationships and properties, which makes them highly versatile for different applications, such as semantic search or recommendation systems. The graph's ability to link related concepts allows for more sophisticated querying and data retrieval, enabling users to uncover insights that might be difficult to discern from traditional data analysis methods. Furthermore, knowledge graphs often utilize ontology frameworks, which provide a structured and formal representation of knowledge within a specific domain. Ontologies help in defining the types of entities and the possible types of relationships that can exist in the graph, thereby establishing a consistent and standardized framework for data representation and interpretation. A critical aspect of knowledge graphs is knowledge graph embedding, which is a method used to represent nodes and edges in continuous vector spaces. This technique facilitates the application of machine learning algorithms to knowledge graphs. By transforming the graph's entities and relationships into numerical form, embedding methods make it easier to compute and analyze the graph's data, enabling tasks such as link prediction, entity resolution, and recommendation. There are several approaches to knowledge graph embedding, such as TransE, DistMult, and ComplEx, each with its own way of modeling relationships and entities. These methods vary in complexity and suitability for different types of graphs and applications, and choosing the right method depends on the specific requirements and characteristics of the knowledge graph in question. In the following sections, we focus on using knowledge graphs for log analysis and root cause analysis. We distinguish methods that target the identification of abnormal patterns (Section 2.3.4) from methods that aim to provide a level of explicability of their prediction (Section 2.3.4).

Log analysis

Ekelhart et al. [52] propose Semantic LOG ExtRaction Templating (SLOGERT) for automated Knowledge Graph construction from raw heterogeneous logs. The main contributions are: (i) a novel paradigm for semantic log analytics that leverages knowledge-graphs to link and integrate heterogeneous log data; (ii) a framework to generate RDF from arbitrary unstructured log data through automatically generated extraction and graph modeling templates; (iii) a set

of base mappings, extraction templates, and a high-level general conceptualization of the log domain derived from an existing standard as well as vocabularies for describing extraction templates; (iv) a prototypical implementation of the pro- posed approach 3, including detailed documentation to facilitate its reuse and (v) an evaluation based on a realistic, multi-day log dataset.

Xie et al. [187] introduce a graph-based approach for log anomaly detection, termed LogGD. This approach converts log sequences into graph representations, employing the Graph Transformer Neural Network. This network uniquely integrates the graph structure with the semantics of its nodes to enhance log-based anomaly detection. They assess LogGD's effectiveness using four prominent public log datasets of system logs. They findings demonstrate that LogGD surpasses current leading quantitative and sequential approaches in anomaly detection, exhibiting robust performance across various window size configurations. These outcomes substantiate LogGD's efficacy in detecting anomalies in log data.

He et al. [85] introduce AdvGraLog, a novel approach utilizing a Generative Adversarial Network (GAN) model that leverages log graph representations to identify anomalies. Log anomalies are identified when the discriminator's reconstruction error is substantial. This method involves creating log graphs and using a Graph Neural Network (GNN) for a detailed graph representation. Their approach transforms standard negative logs into adversarial samples using a GAN generator. The discriminator, which is an AutoEncoder, detects anomalies by evaluating the reconstruction error against a set threshold. Adversarial training is employed to enhance the quality of adversarial samples and improve the discriminator's ability to detect anomalies. The experimental results, conducted on real-world datasets of system logs, highlight the superiority of AdvGraLog over conventional methods.

Li et al. [106] propose the GLAD framework (Graph-based Log Anomaly Detection), which is tailored to unearth relational anomalies within system logs. GLAD integrates log semantics, as well as relational and sequential patterns, into a comprehensive anomaly detection approach. It starts with a field extraction module using prompt-based few-shot learning to pinpoint critical fields in log data. Following this, GLAD forms dynamic log graphs within sliding windows, linking these fields and parsed log events. In these graphs, events and fields are nodes interconnected by their relationships as edges. GLAD then applies a temporal-attentive graph edge anomaly detection model to spot anomalous connections in these graphs. This model combines a Graph Neural Network (GNN) with transformers to assimilate content, structure, and temporal dynamics. Tested on three datasets on system logs, their method proves its efficacy in detecting anomalies, as evidenced by various relational patterns.

In this master thesis [134], Lucas Payne proposes to use knowledge graphs for log anomaly detection. His study develops a complete system that leverages graph neural networks (GNNs) and KGC for anomaly detection in log files. This system comprises two primary modules:

the initial module uses predefined templates to construct a KG from log data representing normal operations. The second module utilizes KG embedding models, augmented with GNN layers, on the formed KG. It then applies KGC to evaluate the likelihood of newly encountered information being anomalous, using a binary classification approach. He tested the approach on two publicly available datasets, assessing its performance using established KGC metrics.

Root Cause Analysis

Li et al. [107] introduce Logs2Graphs, an unsupervised graph-based approach for log anomaly detection. The method transforms event logs into attributed, directed, and weighted graphs, which are then analyzed using graph neural networks for graph-level anomaly detection. Therefore, logs need to be parsed to log events, which can be complex in practice. They present a new graph neural network architecture, termed One-Class Digraph Inception Convolutional Networks (OCDiGCN), specifically designed to detect graph-level anomalies in such graphs. OCDiGCN combines graph representation with anomaly detection, enabling the learning of a representation particularly effective for identifying anomalies, thereby enhancing detection accuracy. A notable feature of OCDiGCN is its ability to pinpoint a small set of critical nodes that significantly influence its predictions. These nodes provide explanatory insights, aiding in root cause analysis following anomaly detection. Our evaluation, conducted on five benchmark datasets, demonstrates that Logs2Graphs matches or surpasses existing state-of-the-art methods in log anomaly detection on system log datasets.

Tailhardat et al. [162] focus on enhancing incident management in Information and Communications Technology (ICT) systems through the use of knowledge graphs and graph embeddings. They address the complexity and uncertainty in causal reasoning for incident management in complex ICT systems. In the context of their work, incident management refers to infrastructure problems or large-scale issues. It, therefore, differs slightly from CI/CD build failures. The authors propose using knowledge graphs to explicitly represent incident contexts, aiding support teams in responding to complex incidents. Graphs are based on a domain-specific ontology called NORIA-O [161], proposed by the authors in a previous paper. They explore challenges in incident management using RDF knowledge graphs, as well as multiclass classifiers with graph embeddings for incident categorization and model-based anomaly detection. The paper includes experiments and results using a dataset from real industrial settings. It concludes with insights into the approach's effectiveness and suggestions for future work in this area, emphasizing the potential of knowledge graphs in simplifying the diagnostic phase of incident management.

Sui et al. [160] introduce LogKG. This novel framework is designed for diagnosing failures by constructing knowledge graphs (KGs) from log entries. LogKG works by extracting entities and their interrelations from logs, thereby enabling comprehensive mining of multi-field

information through the knowledge graph. To optimize information encapsulated in these KGs, they propose a failure-oriented log representation (FOLR) method to identify failure-related patterns. Leveraging the OPTICS clustering algorithm, LogKG aggregates historical failure instances and labels typical failure scenarios, facilitating the training of a robust failure diagnosis model capable of pinpointing root causes. They demonstrate LogKG's superiority over existing methodologies through evaluations conducted on both real-world and public log datasets. However, both datasets are system logs, not CI/CD logs. Furthermore, they deployed LogKG within a leading global Internet Service Provider (ISP), underscoring its efficacy and practicality in real-world applications.

We have explored the use of knowledge graphs for log analysis or root cause analysis. In the next section, we focus on log enrichment, which ultimately allows knowledge graphs to be enriched with data relevant to the prediction task.

2.3.5 Log enrichment

Log fusion, also known as log enrichment, might be helpful to enhance root cause analysis methods by providing a more comprehensive and integrated view of system events. This technique involves linking log events from various sources and possibly enriching them with additional contextual information, which aids in creating a more detailed and accurate picture of system activities and interactions. This enriched log data allows for more effective identification and understanding of anomalies, patterns, and trends that could indicate underlying issues. By integrating logs from multiple components, log fusion enables a holistic analysis, facilitating the discovery of correlations and causal relationships that might be overlooked when examining logs in isolation. This comprehensive perspective is essential for accurately diagnosing system problems, ultimately leading to more effective troubleshooting and resolving complex issues. Therefore, we explore methods that enable log fusion. Interestingly, such methods are often published in security conferences thanks to their ability to identify patterns indicating compromising or malicious behavior. We postulate that root cause analysis problems can be seen as behavior identification (malicious or not), and therefore, we explore existing work in this field.

In the context of today's enterprise environments, endpoint monitoring solutions play a crucial role in enhancing attack detection and investigation capabilities. These solutions are instrumental in continuously recording system-level activities through audit logs, thereby providing an in-depth view of security incidents. They encounter, however, a significant challenge: a semantic gap exists between the low-level audit events and the high-level system behaviors, complicating the task of security analysts to recognize behaviors of interest and detect potential threats. Traditionally, to bridge this gap, the predominant approach involves matching streams of audit logs against a rule-based knowledge base that describes specific

behaviors. This process, however, heavily depends on expert knowledge, making it less accessible and more complex.

Addressing this issue, Zeng et al. [193] introduce WATSON, an innovative automated approach designed to simplify the abstraction of behaviors by inferring and aggregating the semantics of audit events. WATSON allows uncovering of the semantics of events through their usage context within audit logs. It extracts behaviors by linking connected system operations and then synthesizes the event semantics to represent them. Furthermore, to significantly reduce the analysis workload, WATSON clusters semantically similar behaviors and identifies the key representatives for further investigation by analysts. The effectiveness of WATSON is underscored in its evaluation, where it demonstrates high accuracy in behavior abstraction and the potential to reduce the analysis workload by two orders of magnitude in the context of attack investigation.

Yu et al. [190] introduce ALchemist. This approach leverages the high-level semantics provided by built-in application logs and combines them with the low-level, fine-grained information found in audit logs. Notably, these two types of logs share many common elements, which ALchemist exploits through a log fusion technique. Central to ALchemist is a relational reasoning engine called Datalog, which enhances the technique's ability to infer new relations, such as the task structure of execution in various applications, including in environments characterized by complex asynchronous execution models. This capability is particularly effective in discerning high-level dependencies between log events. The researchers' evaluation, which encompassed 15 popular applications like Firefox, Chromium, and OpenOffice, as well as 14 Advanced Persistent Threat (APT) attacks documented in the literature, reveals ALchemist's high efficiency. Notably, it achieves these results without requiring software instrumentation. The technique gives very good results in partitioning execution into autonomous tasks, thereby minimizing false dependencies and creating precise attack provenance graphs. The overhead incurred is minimal. Furthermore, in comparative analyses, ALchemist demonstrates superior performance over NoDoze [78] and OmegaLog [79], two other state-of-the-art techniques that do not require instrumentation. This advancement represents a significant leap forward in the domain of attack forensics, offering a more refined and effective tool for combating sophisticated threats.

2.3.6 Summary

We outline the difficulty of log analysis due to the volume, diversity, and ambiguity of logs. Conventional methods like searching for specific error terms are ineffective because of the natural language and varied vocabulary used in logs. Additionally, logs are often imbalanced, with a vast majority representing normal operations, complicating anomaly detection. Log analysis is divided into parsing and mining. Parsing involves transforming logs into a struc-

tured format for further analysis. This process is challenging due to the lack of standardization in log formats and the dynamic nature of software development. Several approaches to log parsing are mentioned, such as rule-based and statistical methods. Each has its advantages and limitations, and no single method is universally effective. We also explore log mining, which involves processing structured logs to detect anomalies. This process is complicated because log sequences may not accurately represent the sequence of actions in the software. Various methodologies and tools for log mining are discussed, including statistical approaches and Natural Language Processing (NLP). We then explore the use of Knowledge Graph for processing logs. Despite the proliferation of research papers detailing the use of knowledge graphs for enhancing the interpretability, accessibility, and actionable insights from system logs in various domains, there appears to be a conspicuous absence of academic work focusing specifically on CI/CD logs. Finally, we introduce the concept of log enrichment, which integrates logs from multiple sources to provide a more comprehensive view of user or system behaviors, and we postulate it as useful to support root cause analysis. In the following section, we explore failure patterns in CI/CD environments.

2.4 Failures in CI/CD pipelines

2.4.1 Principles

Understanding and classifying these build failures is essential for rapid troubleshooting, improving the development process, and ensuring a stable and efficient CI/CD pipeline. Each type of failure requires a different approach to resolve, and accurately identifying the failure category can significantly expedite the process of fixing builds and maintaining continuous delivery. We described failure categories for CI/CD pipelines in Section 1.4. In the following sections, we delve into existing work on CI/CD.

2.4.2 Empirical analysis of CI/CD logs

This section delves into the empirical analysis of Continuous Integration and Continuous Deployment systems, focusing on their operational dynamics, efficiency, and impact on software development practices.

Build failures In the realm of both industrial and open-source software development, Continuous Integration (CI) practices have been widely adopted, offering notable improvements in the software development lifecycle. However, the occurrence of errors during CI builds emerges as a substantial challenge, undermining development efficiency and incurring significant costs due to the increasing time required for error resolution. Rausch et al. [142] conducts

a comprehensive analysis of build failures within CI environments, establishing a linkage between repository commits and corresponding CI build data. Through an examination of 14 open-source Java projects, the research identifies 14 prevalent error categories, with test failures being the most dominant, accounting for more than 80% of errors in some projects. Additionally, the study uncovers issues related to noisy build data, such as transient Git interaction errors and infrastructure flakiness. An exploration into the factors influencing build outcomes reveals that, while process metrics significantly affect the build success in half of the projects, the most critical determinant across all projects is the stability of recent build history. Specifically, for a majority of the projects, over 50% of failed builds were preceded by another build failure, highlighting the profound impact of recent build performance, evidenced by the fail ratio of the last ten builds, on subsequent build outcomes.

Bad practices Zampetti et al. [191] explore the adverse effects and challenges of CI adoption despite its acclaimed benefits for enhancing software quality and reliability. Employing a mixed-method approach that included semi-structured interviews with 13 experts and an analysis of over 2,300 Stack Overflow posts, they have identified 79 CI "bad smells" categorized into seven distinct areas concerning the management and process of CI pipelines. Further examination through a survey of 26 professional developers assessed the significance of these bad practices, juxtaposing the findings with existing literature. Notably, while some findings, such as the inefficient use of branches, reinforce prior studies, others reveal previously unaddressed issues, like misapplications related to static analysis tools or the excessive use of shell scripts, and challenge prevailing beliefs, for instance, the advisability against nightly builds. The implications of this catalog of CI bad smells extend to various stakeholders: recommending practitioners to prioritize specific, portable tools and transparency in build failures; suggesting educators instill a comprehensive CI culture beyond mere technological instruction, including what practices to avoid; and urging researchers to focus on developing tools for failure analysis and the detection of CI bad smells.

Repair time Despite CI's widespread adoption, it has been observed that the process of integrating changes frequently necessitates considerable time and effort to address emerging errors, consequently halting the progress of development tasks such as the introduction of new features and bug resolution. Ivens et al. [154] delve into the variables affecting the duration required to rectify build failures within the CI framework, focusing particularly on the roles of developer activity, project characteristics, and the complexity of builds. By analyzing data gathered from 18 industrial projects affiliated with a software company, encompassing 13 metrics derived from existing literature on build failure analysis, this study employs association rules (i.e., a data mining technique) to explore the nexus between the aforementioned factors and the time to correct build failures. The findings elucidate significant correlations,

underscoring that more seasoned developers tend to resolve build failures more swiftly and that failures identified in the initial phases of a project are corrected faster. Moreover, it was noted that build failures involving a higher count of lines and modified files are associated with extended correction times. Consequently, this research illuminates the determinants influencing build failure correction time in CI, offering insights that could enable software development teams to refine their CI methodologies and mitigate the repercussions of build failures on their developmental endeavors.

Build flakiness Durieux et al. [48] examine the phenomena of restarted and flaky builds alongside their repercussions on the development workflow. They discover that developers opt to restart at least 1.72% of builds, totaling 56,522 restarted instances within the Travis CI dataset analyzed. This practice is more prevalent in mature and complex projects, where builds initially fail due to factors such as test failures, network issues, or Travis CI limitations like execution timeouts. The analysis reveals that restarted builds significantly affect the development process; notably, in 54.42% of cases, developers analyze and restart a build within an hour of its initial execution, indicating that they frequently interrupt their workflow to resolve CI issues. Furthermore, these restarts triple the time required to merge pull requests, extending the median merging duration from 16 hours to 48 hours, thereby highlighting the substantial impact of restarted builds on the efficiency of development practices.

2.4.3 Predicting CI/CD failures

Kerzazi et al. [98] explore the consequences of build breakdowns: they examine 3,214 builds generated within a large software firm over a six-month period. Their findings reveal a significant incidence of build failures, amounting to 17.9%. Additionally, they calculated the cost associated with these failures to exceed 336.18 man-hours. To gain deeper insights into these issues, they interviewed 28 software engineers from the company. These discussions shed light on the typical scenarios leading to build failures and their impact on team collaboration and coordination. Their quantitative analysis further explored the primary factors influencing build breakdowns. They discovered a correlation between build failures and several variables: the number of contributors working concurrently on branches, the nature of tasks undertaken on a branch, and the roles of the stakeholders involved in the builds, such as developers versus integrators.

Santolucito et al. [150] introduce VeriCI, a tool designed for localizing CI configuration errors directly at the code level before pushing the code. VeriCI helps developers to ensure the correctness of CI/CD configurations without starting a real CI/CD build, which often takes a significant time (commit, push, wait for the build to finish or fail). This tool operates as a static analysis instrument before initiating the build request on the CI server. VeriCI leverages

the commit history and the corresponding build histories present in CI environments. These histories serve a dual purpose: predicting build errors and localizing them effectively. VeriCI harnesses these build histories as a labeled dataset for automatically generating customized rules that describe accurate CI configurations. This is achieved using supervised machine learning techniques. To enhance the precision in identifying the root causes of build failures, VeriCI employs a neural network. This network is trained to filter out constraints less likely associated with the root cause of the failure. The performance of VeriCI has been evaluated using real-world data sourced from GitHub. The results show that VeriCI achieves a 91% accuracy rate in predicting build failures and correctly pinpointing the root cause in 75% of cases. Furthermore, a between-subjects user study involving 20 software developers has demonstrated that VeriCI significantly aids users in identifying and rectifying errors in CI configurations.

Continuous Integration and Continuous Deployment enable developers to build software projects automatically and frequently. However, they face challenges such as extended build duration and recurrent build failures, collectively called build performance issues. This aspect of CI can deter developers from pursuing further development tasks. Although prior research has examined build durations and failures separately, scant attention has been paid to the potential interplay between minimizing build times and reducing failures. Specifically, existing literature lacks clarity on how build performance is affected by project context, the impact of efforts to decrease build times on the frequency of build failures, and whether addressing failures leads to longer or shorter build times. Recognizing the importance for developers to achieve both timely and successful CI builds, Ghaleb et al. [59] explore through experimental and survey methods practices that might have dual or inverse effects on build durations and failures. By refining the TravisTorrent [12] dataset to focus on active projects, collecting data on 924,616 CI builds from 588 GitHub projects linked with Travis CI, and surveying developers involved in these projects for their insights, this research identifies significant correlations between project characteristics and both build durations and failures. It also uncovers a complex relationship between efforts to rectify build failures and their consequences on build times, often leading to longer durations without ensuring successful builds. Moreover, the study reveals that improving a project's build performance depends on its existing build times and failure rates and that, over time, projects may prioritize one performance measure over the other, especially when it is not feasible to optimize both simultaneously. The findings, bolstered by survey feedback, offer developers guidance on practices for maintaining efficient and successful CI builds while urging researchers to consider the potential dual or inverse implications of their recommendations on CI build outcomes.

Chen et al. [23] propose a novel, history-aware prediction method named BuildFast. The approach leverages multiple failure-specific features derived from analyzing build logs and changed files in closely related historical builds. It introduces an adaptive prediction model

that alternates between two models based on the outcome of the preceding build. BuildFast's efficacy was evaluated in a practical online scenario, predicting builds in chronological order while assessing the benefits of correct predictions against the costs of incorrect ones. Through experimentation across 20 projects, BuildFast demonstrated a 47.5% improvement in F1-score for predicting failed builds over existing methods, showcasing its potential to enhance efficiency and reduce costs in CI environments significantly.

Saidani et al. introduce DL-CIBuild [149], a novel approach employing Long Short-Term Memory (LSTM)-based Recurrent Neural Networks (RNN) for the prediction of CI build outcomes. This method leverages historical CI build data to learn both long and short-term dependencies, requiring sophisticated feature engineering to extract informative features for accurate prediction. Additionally, they employ a tailored Genetic Algorithm (GA) for hyper-parameter optimization of the LSTM model. Through extensive evaluation using a benchmark comprising 91,330 CI builds from 10 large, long-lived software projects utilizing the Travis CI build system, they demonstrate the LSTM-based model's superior performance over traditional Machine Learning (ML) models in both online and cross-project prediction scenarios. Their findings highlight DL-CIBuild's reduced sensitivity to training set size and its robustness against concept drift, further showcasing the Genetic Algorithm's efficacy in hyper-parameter optimization compared to several baseline Hyper-Parameter Optimization (HPO) methods.

2.4.4 Classifying CI/CD failures

In the evolving field of agile software testing, the integration of machine learning for automated root cause analysis represents a significant advancement. Kahles et al. [96] present an approach wherein machine learning techniques are utilized to classify CI/CD build logs. The primary focus is on feature extraction from these logs, a critical step in understanding and automating the analysis process. The initial phase of the study involves clustering unlabeled data. This step, although yielding some correlations between various clusters and failure root causes, is hampered by the ambiguity present in numerous clusters. This observation underscores the necessity for labeled data to enhance the analysis. Subsequently, further interviews with testing engineers facilitate the establishment of five definitive ground-truth categories. This refinement plays a significant role in the next phase of the study. It enables the training of sophisticated artificial neural networks, which are employed for two key purposes: classification of the data and its preprocessing for enhanced clustering. The effectiveness of these neural networks is evident in the impressive accuracy rate of 88.9% achieved by the methodology. What sets this study apart is its dual approach: not only does it leverage the expertise of human professionals in the testing field for initial data interpretation, but it also successfully integrates this expert knowledge into machine learning algorithms for root cause analysis. This methodology does not merely represent a solution for the current challenges in agile software testing environments but also establishes a foundational framework for future

research and applications. It exemplifies the potential of combining human expert insights with advanced machine learning techniques to create robust, accurate, and efficient systems for root cause analysis in dynamic and agile contexts. This work is mainly extracted from a Master thesis [11] by the principal author.

Afonso Soares wrote a Master thesis [114] on predicting build flakiness by classifying CI/CD log files. The thesis explores the feasibility of using machine learning and natural language processing techniques for identifying flaky behavior in build processes through text classification of log files, suggesting integration into CI/CD pipelines for automatic detection and restart of flaky builds. The study initiated data collection from the GitHub Actions CI platform, gathering over 37k builds from nearly 17k repositories. The dataset includes information on when a build was retried. An empirical analysis of this data, alongside a comparison with a prior study on Travis CI, confirmed the persistence of flakiness at similar rates in the GitHub ecosystem. A dataset comprising 2,567 successful and 1,608 unsuccessful restarts was then used for classification tasks. During the experimentation phase, various subsets from the Travis CI and GitHub Actions datasets were evaluated under different scenarios with supervised learning algorithms, achieving accuracies over 70% across diverse conditions, though preprocessing techniques like stop-word removal and stemming were found ineffective. Attempts to transfer knowledge between CI systems (i.e., train on build logs from one CI system while predicting flakiness from build logs from another CI system) were unsuccessful, but the potential remains unexplored for future research. The study concludes that applying standard text information retrieval methods to detect flakiness in CI/CD log files is viable and promising for further advancements.

Bonic et al. [15] introduce a framework to ease root cause analysis of CI/CD failures. This framework employs a modular architecture that encompasses the collection, identification, analysis, and presentation of data. It is designed to automatically structure vast and varied datasets, augment this data with additional insights from the continuous integration system, and incorporate both default and customizable labels to aid in pinpointing the causes of failures. The framework allows easier data normalization, which is typically required for machine learning. Additionally, it drastically reduces the time required for manual debugging, as manually searching through the generated artifacts within a feasible timeframe is often impossible. The efficacy of this method was validated through the measurement of manual root cause analysis times across multiple jobs, utilizing publicly available data from the Kubernetes and OpenShift projects. This not only demonstrates the practical benefits of the framework but also ensures the reproducibility of the research by the wider academic community. In conclusion, the framework augments CI/CD logs with data from various sources to target a homogeneous format and allow the search for user-defined patterns to assign labels to build logs.

2.4.5 Locating root cause message in CI/CD build logs

Locating root cause messages (indifferently named fault localization) involves identifying specific log entries, error messages, or system alerts that directly indicate the underlying issue or anomaly leading to a system's malfunction or degraded performance. This process requires sifting through potentially vast amounts of data to pinpoint the exact messages that reveal the cause of a problem. On the other hand, the classification of failures as explored in Section 2.4.4 refers to the broader task of categorizing system failures into predefined types or classes based on their characteristics, symptoms, or effects on the system. This classification helps in understanding the nature of failures, facilitating quicker diagnosis, and informing appropriate responses. While locating root cause messages is a focused approach aimed at uncovering the direct cause of a single failure incident, the classification of failures provides a higher-level overview, enabling the identification of patterns or trends in system behavior over time. Nevertheless, identifying root cause messages is crucial, particularly when constructing or maintaining a set of classification labels poses challenges, such as the need for numerous labels. Moreover, employing fault localization techniques as a preliminary step before applying classification methods can help minimize noise data, thereby enhancing the performance of classification algorithms.

The task of fault localization within CI/CD build logs must be distinctly understood and not mixed up with tasks that might appear similar but are fundamentally different. To begin with, detecting abnormal lines within these logs should not be mistaken for identifying root cause messages. This distinction is crucial, particularly when multiple log messages are marked as abnormal. Furthermore, specific methodologies, for example, BLUiR [148], facilitate precisely determining the root cause's location within the source code (e.g., function name) from bug reports. However, bug reports are fundamentally different from CI/CD build logs as they are written in natural language. Therefore, such methods are not transparently applicable to CI/CD failures.

Existing techniques for troubleshooting CI failures fall short in automatically pinpointing the root causes due to two significant limitations. First, these techniques predominantly focus on identifying faults within the source code, overlooking the build scripts, which can also trigger CI failures. Second, the current tools are designed to address test failures, neglecting the instances where a build failure might be the underlying issue. Addressing these gaps, Hassan et al. introduce UniLoc [77], a unified technique for localizing faults in both source code and build scripts based on CI failure logs. UniLoc utilizes an information retrieval (IR) strategy without presupposing the failure's specific location or nature, whether in source code, build scripts, or whether it pertains to a test failure. By treating source code and build scripts as searchable documents and considering build logs as search queries, UniLoc goes beyond the conventional application of off-the-shelf IR techniques. It employs domain-specific heuristics

to refine search queries, search spaces, and ranking formulas, enhancing the accuracy of fault localization. Evaluating 700 CI failure fixes across 72 open-source projects built with Gradle demonstrates UniLoc's superior performance. Authors support it marks a significant advancement over existing state-of-the-art IR-based tools like BLUiR and Locus, underscoring UniLoc's potential to significantly improve the efficiency and accuracy with which developers diagnose root causes of CI failures.

2.4.6 Fixing CI/CD failures

Hassan et al. [76] present initial steps towards developing an automatic build repair solution in the CI environment, addressing both source code and build script issues. They started with an empirical investigation of software build failures and corresponding repair patterns. Drawing insights from this study, they devised a method capable of autonomously rectifying build script errors. Future expansions of this method aim to integrate repair mechanisms for both source code and build scripts. Additionally, they propose to evaluate the effectiveness of their automated solutions through user studies and comparisons between our generated fixes and actual repairs.

2.4.7 Reproducible CI/CD artifacts

Unreproducible binaries refer to the issue where the same source code yields different binary outputs in separate build processes. This variability hinders the ability to verify the integrity and origin of binaries, making it challenging to determine whether differences are due to legitimate code changes or malicious alterations. Similarly, in the context of CI/CD, unreproducibility occurs when identical source code leads to different build outcomes or artifacts each time it is built in the CI/CD pipeline. This inconsistency can occur due to various factors, such as differing build environments, dependencies (e.g., network services), or timing issues. Achieving reproducible CI/CD builds is important for reliability and security, as it guarantees consistent behavior across various environments and deployments, helps prevent unpredictable build outcomes (i.e., flaky builds), and ensures that the pipelines are easier to debug and validate.

He et al. [80] address the challenge of unreproducible builds in software engineering, a problem that undermines software credibility by generating different artifacts from the same build process. They highlight that while significant attention has been devoted to automatically identifying the causes of such unreproducibility, the domain of automatic rectification has been largely overlooked. To bridge this gap, they introduce ConstBin, an automated tool designed to produce consistent artifacts by detecting and rectifying unreproducible commands within the build process. ConstBin operates through an extensible set of rules to replace prob-

lematic commands with their corrective operations, marking a pioneering step towards fixing inconsistencies during build processes. The effectiveness of ConstBin was assessed on 348 open-source packages known for their unreproducible builds, demonstrating a success rate of 83.91% in achieving consistent artifacts. This research not only underscores the feasibility of automating the correction of unreproducible builds but also positions ConstBin as the first tool of its kind to address this issue successfully.

Ren et al. propose RepFix [144] for automated patch generation to fix unreproducible builds. By employing system-level dynamic tracing, RepFix captures detailed system call traces and user-space function call information, integrating both kernel and user-space probes to identify the execution points of build commands precisely. Simultaneously, it introduces a similarity-based mechanism for retrieving relevant patches and generates fixes by applying the edit operations found in existing patches, benefiting from the extensive archive of patches accumulated through reproducible build practices. The effectiveness of RepFix is demonstrated through extensive testing on a dataset of 116 real-world packages, successfully rectifying unreproducible build issues in 64 packages.

2.5 TLS inspection

TLS provides security guarantees that are essential to secure communications. However, in some contexts, traffic must be inspected for legitimate reasons, such as ensuring compliance, security, or optimization (e.g., cache). Therefore, multiple approaches to allow inspection of encrypted traffic were developed.

2.5.1 Principles

Different approaches for TLS inspection exist, ranging from network interception, target program cooperation (e.g., key extraction), or program instrumentation. Man-In-The-Middle (MITM) places a proxy between the client and server. It will relay traffic and impersonate servers from a client perspective. Therefore, clients must trust fake certificates emitted by the proxy. Another approach is to instrument (e.g., function hook, library replacement) target program to directly retrieve plaintext traffic or disable security checks (e.g., certificate validation). While this approach often works, it is time-consuming and challenging to implement (e.g., statically linked binaries, large binaries) as it is program-specific. Furthermore, this is feasible for dynamic libraries but a lot more difficult for static libraries because there is no easy way to intercept function calls, and it can become very difficult when Link Time Optimization (LTO) is used. Cooperative approaches were developed to enable programs to log TLS session keys for debugging. The *de facto* standard SSLKEYLOGFILE environment variable allows dumping session keys to a text file. However, support for this feature is limited and

often disabled, as it opens new security vulnerabilities. We detail the benefits and limitations of each approach in Chapter 5. The following section presents literature on TLS inspection approaches based on memory analysis.

2.5.2 Memory analysis for TLS inspection

Dolan-Gavitt et al. [42] proposed a framework based on PANDA for memory forensics. They applied their work for TLS 1.2 session decryption. Each target program must find a tap point that reads or writes the TLS master secret. To this end, they used an instrumented TLS server that saves the master secret; then, they used a string-based search to identify the part of the program that wrote the secret. Therefore, their method requires that the target program allows connection to an arbitrary server (i.e., instrumented server). They tried their approach on OpenSSL `s_client`, Chrome, and Firefox, among others. For all programs in the experiment, they found the tap point that wrote the master key. However, it is unclear if programs should contain a unique tap point to write TLS keys. Large software like Chrome or Firefox includes multiple subsystems to establish TLS connections: fetching pages, checking updates, account synchronization, downloading safe-browsing lists, etc. The authors did not mention if they succeeded in decrypting all TLS traffic or only a part of it. Furthermore, their approach requires the target program to run in a hypervisor: they concede a 5x slowdown over native execution. While this overhead is fine for some specific analyses, it is not acceptable for deployment on production systems.

Taubmann et al. [166] propose TLSkex for extracting TLS keys for processes running inside a virtual machine. They leverage Virtual Machine Introspection (VMI) techniques to snapshot guest memory, and they further reduce the search space using heuristics: limit to memory pages that are writable and entropy-based lookup. Furthermore, they use virtual network interfaces to slow down (i.e., block for a short period) TLS packets to allow memory snapshots to happen. Their method works on TLS 1.2. However, in 2023, more than 90% of the traffic to the top content delivery network provider Cloudflare is encrypted using TLS 1.3 or QUIC [26]. As TLS 1.3 prevents downgrade attacks, their method can only be used on 10% of Cloudflare's traffic. Furthermore, TLS 1.3 uses 4 keys for handshake and application traffic security instead of 1 master key for TLS 1.2. The authors support their approach as useful for decrypting TLS sessions of malicious programs. However, malware tends to act differently when it detects it is running in a virtual machine, e.g., it does not run its malicious payload or contact their C&C servers [10]. In addition, they experiment with their method only on small client programs (curl, wget, OpenSSL s_client) and one server program (Apache2). Large programs such as web browsers or web-based apps (e.g., Electron-based) may be significantly more challenging, considering their large memory footprint. Finally, the method has not been tested with simultaneous handshakes. However, parallel TLS connections are pervasive with such programs. As TLSkex requires virtualization, large-scale deployment would significantly

impact performance since virtualization implies a 5 times slowdown [42].

Nubeva [91] is a commercial product that allows automated decryption of TLS traffic in cloud-native environments. To this end, an agent must be installed on each virtual machine running TLS clients. They claim to support TLS 1.2 and TLS 1.3. At the start of the TLS session, the agent extracts the TLS session keys from the process memory. Their approach relies on a signature database to locate keys in process memory. The signature database contains memory patterns, such as memory content located before or after session secrets, for known programs. Therefore, a signature must exist for the target program. If no signature is found, TLS sessions cannot be decrypted. It is unclear whether the signature applies to the TLS library or the whole target software. To generate new signatures, Nubeva would need access to the target program for inspection, which is not always desirable because of confidentiality or intellectual property constraints. Furthermore, it is unclear if they require the target program to start a TLS connection to a complicit server to generate the signature (which is impossible with some programs, e.g., hard-coded URLs). Finally, there is no public insight into the manual work required to add support for a new TLS client. As Nuveba's approach is not generic (i.e., it does not work on unknown software), the efforts required to use it at scale are significantly increased.

DroidKex [165] provides a method for fast data extraction from process memory applied to Android applications. They demonstrate their ability to extract TLS ephemeral keys without prior knowledge of memory structure. However, Android applications tend to have a smaller memory footprint than desktop apps. Furthermore, the TLS libraries ecosystem is reduced compared to libraries available for desktop operating systems. Therefore, this method does not seem directly applicable to large programs like browsers.

An ideal method would work on TLS 1.2, TLS 1.3, and QUIC. Furthermore, it should not require the target program to run in a virtual machine or program modification. Finally, the method should be generic to work on many software, including large programs such as web browsers.

Method	Fully generic	Require hypervisor
Dolan-Gavitt et al. [42]	no	yes
TLSkex [166]	yes	yes
Nubeva [91]	no	no
X-Ray-TLS	**yes**	**no**

Table 2.3: Comparison of methods that extract TLS keys from process memory.

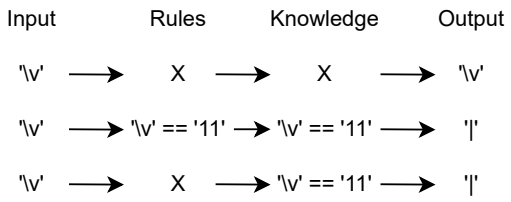

Figure 2.4: Illustration of the learning process and perpetuation of knowledge in a compiler. After learning (step 2), the compiler can apply the knowledge without explicit rule (step 3).

2.5.3 Summary

We explore existing work for TLS inspection, a process necessary in certain contexts for compliance, security, or optimization. We briefly explore interception methods, such as Man-In-The-Middle (MITM) proxies, program instrumentation, or cooperation with target programs to log TLS session keys, before focusing on methods based on memory analysis. A deeper analysis of existing methods is presented in Chapter 5. Memory analysis techniques for TLS inspection rely on extracting the encryption key from program memory without the collaboration of the target program. However, these methods face limitations such as reliance on hypervisors, which introduces significant performance overhead, compatibility issues with newer protocols like TLS 1.3 and QUIC, or their ability to be generic of the target programs. Commercial and research solutions are discussed, each with its own trade-offs regarding generality, the requirement for hypervisors, and the ability to handle versatile applications.

2.6 Malware targeting CI/CD systems

Continuous Integration and Continuous Deployment pipelines are crucial for automating software development processes but are susceptible to several attack vectors. Furthermore, CI/CD systems can be self-hosted systems (i.e., the CI/CD system is used to build a new version of the CI/CD system), thus opening new areas of attack vectors, especially for long-term persistence. In the following sections, we explore the security of self-hosted architecture and common attack patterns for CI/CD systems.

2.6.1 Self-hosted architecture security

In 1984, Ken Thompson introduced a paper [169] discussing the level of trust that should be placed in software, especially for self-hosted architectures. The crux of Thompson's argument is illustrated through a thought experiment, where he describes how a compiler could be modified to include a backdoor in a program it compiles, such as a login system, without leaving any trace in the source code of either the compiler or the compiled program. This

modification to the compiler could also be made to self-perpetuate, meaning the compiler would insert the backdoor into any future versions of itself, spreading the vulnerability without detection. Figure 2.4 illustrates this property. The first step is to teach the compiler that `\v` is equal to ASCII character 11. Then, even after removing the rule from the source code, `\v` will be modified to ASCII character 11 as the rule is now part of the compiler. Furthermore, the rule will persist when the compiler is compiled again (e.g., compiler update). Gratzer et al. [72] show that knowledge acquired by quines (i.e., self-replicating software) is difficult to detect and remove. Thompson extends this idea to a more profound level by suggesting that any software could potentially be compromised at any stage of its development or execution process. This includes not just compilers but operating systems, libraries, and even hardware. The essence of his reflection is on the deep-rooted trust we place in the unseen and unverified mechanisms that underlie our digital tools. He provocatively concludes that you can't trust code that you did not totally create yourself.

Compilers are an interesting target because of their ability to alter source code just before compilation. Bauer et al. [152] took advantage of compiler bugs in GCC to create a backdoor. Furthermore, two real-world attacks leveraged this concept. In 2009, the `Win32.Induc.A` virus [56] infected Delphi compilers to add malicious code to applications built using the compiler. In 2015, Xcodeghost [186] was one of the most successful attacks on the Apple App Store. It consisted of a modified Xcode compiler that makes the applications being spied on.

Wheeler [178] proposes a countermeasure to detect attacks described by Thompson through diverse double-compiling. It can be combined with other emerging solutions such as bootstrappable [20] and reproducible [21] builds (i.e., methods ensuring consistency between binaries and source codes) to improve trust in software [156].

Self-hosted architectures open new attack vectors for long-term persistence. However, before long-term compromise, the system must be infected initially. To this end, we explore literature about possible paths for initial infection of CI/CD systems, such as remote code execution.

2.6.2 CI/CD security

Backdoors [168] often refer to a long-term compromise of targeted systems. Therefore, the malicious code should persist on updates. There are different ways to get initial access to the system in CI environments. Attacks against CI systems leverage their dependencies to many components, such as code dependencies or tooling used in pipelines. Improper configuration of CI [100] can lead to vulnerabilities [140].

Common vulnerabilities are listed by the OWASP foundation [132]. Notable examples include hypocrite commits [185], dependency confusion [14], or supply chain attacks [101] (e.g., attacks on upstream packages). OSC&R [109] framework describes common techniques for

supply chain attacks, such as compromising legitimate artifacts or publishing and advertising malicious ones. Williams [183] shows that attackers tend to focus on popular projects largely used by the developer's community, even if they are often better protected. In addition, malicious code can come directly from project maintainers [5]. Therefore, malware in CI can come from malicious code in dependency repository [137] or exploits against pipelines [177]. In addition, bypassing auto-merge rules [146] or required reviews [60] can help to merge malicious code to the main branch.

Docker is widely used to provide a consistent execution environment in different systems. However, it raises new questions about security [19, 27, 111]. Indeed, Docker containers share the same kernel as the host system. Furthermore, container images can be made by the community. While images can be analyzed to detect malicious or vulnerable code, analysis often relies on a database of malicious content. This does not allow the detection of unknown (zero-day) attacks. Therefore, using containers for CI could open up new security vulnerabilities, such as using a compromised base Docker image.

Regarding supply chain attacks, attempts have been made to detect and mitigate them, such as forensic [128] or safeguard levels [157] artifacts. Regarding Docker, Petazzoni [135] warned about using Docker-in-Docker as it requires privileged containers, making it easier to attack the host system. Zero Trust Container Architecture [103] encourages always considering the code within the container as potentially malicious, asking for a higher level of security.

2.6.3 Summary

We explore the work proposed by Ken Thompson about the security of self-hosted architecture. Furthermore, we show that CI/CD systems have a large attack surface, ranging from dependency confusion, malicious container images, or code management vulnerabilities.

2.7 Summary

In the state-of-the-art chapter of this thesis, we delve into the existing body of work spanning multiple research domains that are pertinent to the investigations conducted herein. Our exploration started by examining public datasets of Continuous Integration/Continuous Deployment logs, where we identified a notable scarcity. We highlight that the datasets available are insufficient for conducting large-scale machine learning analyses due to various limitations. Subsequently, our attention shifts to log analysis methods, a critical component given its significance in root cause analysis methodologies. In this context, we probe into literature that employs semantic techniques, particularly those utilizing natural language processing (NLP), as another approach for log analysis. This survey examines Knowledge Graphs as a novel approach for enhancing log analysis and facilitating more effective root

cause identification. Further, our investigation encompasses the phenomena of failures within CI/CD processes. Here, we cover a spectrum of research from empirical studies to the prediction, classification, localization, and automated fix of failures, offering a comprehensive overview of current methodologies and their applications. Additionally, the thesis explores the domain of model behavior analysis, shedding light on the current methodologies and their implications for CI/CD processes. We also scrutinize methods for Transport Layer Security (TLS) inspection, a critical aspect of ensuring secure communication within CI/CD pipelines. Lastly, we turn our focus to the realm of CI/CD security, surveying existing research to understand the landscape of security practices, challenges, and advancements within CI/CD environments. This exploration of state-of-the-art research aims to establish a foundation for the novel contributions of this thesis, situating it within the broader context of current academic and practical endeavors in these fields.

Chapter 3

Collection of CI/CD builds

This chapter delves into the collection and analysis of CI/CD runs, showcasing an industrial dataset of Jenkins builds alongside a public dataset of GitHub Actions runs. The focus then shifts to a detailed analysis of the GitHub Actions dataset, where we explore shell command steps' usage and execution times. This analysis aims to comprehensively understand the workflows and efficiency in CI/CD practices, highlighting the intricacies of automation and optimization in software development environments. In the following section, we describe how Jenkins and Github Actions work.

3.1 Background

This section examines the main characteristics of Jenkins and GitHub Actions. Although there are similarities between the two, as both offer CI/CD capabilities, there are notable differences in specific configurations. Additionally, the terminology employed in each platform may vary despite referring to the same concepts. The subsequent section will delve into the properties of Jenkins.

3.1.1 Jenkins

Jenkins is a CI/CD orchestrator used to automate various stages of the software development process, including building, testing, and deploying applications. In Jenkins, a "job" is a basic unit of work representing a single task in the automation process, such as compiling code or running tests. A "pipeline" is a collection of jobs connected to define the workflow of the automation process, allowing for more complex and sequential tasks to be executed as part of the build process. A "folder" in Jenkins organizes and manages related jobs and pipelines, logically structuring and categorizing the CI/CD processes.

The tool enables the definition of continuous delivery pipelines via a `Jenkinsfile`, a text

file containing the pipeline configuration, which is stored in the version control system. This aligns with the Infrastructure as Code (IaC) paradigm, allowing the pipeline configuration to be version-controlled along with the application code. Groovy, used in Jenkinsfiles, is a dynamic language that runs on the Java Virtual Machine (JVM) and integrates seamlessly with Java. Its syntax is similar to Java but less verbose, making it more concise and readable, which is better for scripting. Groovy supports dynamic typing, which is useful for handling various data types dynamically during pipeline execution. It is particularly effective for creating domain-specific languages (DSLs), thanks to its flexible syntax and features like closures and operator overriding, enabling the expressive pipeline syntax used in Jenkins. Additionally, Groovy's integration with Java allows Java libraries and frameworks to be used. In the following section, we describe GitHub Actions.

3.1.2 Github Actions

GitHub Actions (GHA) is a full-featured CI/CD orchestrator proposed by GitHub that aims to compete with other major CI/CD systems such as Gitlab, Travis CI, Jenkins, Circle CI, or Drone. GHA is available for both private and public projects on GitHub. GHA is based on runs, workflows, jobs, and steps. A run is an execution of a workflow. A workflow, also referred to as a CI/CD pipeline, is composed of jobs. Each job is started in a fresh environment (i.e., a dedicated virtual machine). Workflows can run jobs in sequence or in parallel. Each job contains a sequence of steps (action or shell code). GitHub provides cloud runners (i.e. virtual machines that will execute jobs) based on Linux, Windows, and MacOS. Cloud runners are managed by GitHub and are billed per minute. Free accounts have 2,000 free minutes of workflow execution each month, while paid accounts can choose between different plans to meet their needs. Therefore, there is a strong interest in optimizing CI/CD execution time. Project owners can also set up self-hosted runners for advanced use cases. In the remainder of the section, we describe GHA workflow properties.

Workflows

A GHA workflow is a set of jobs that will be triggered on specific conditions. A code repository can have multiple workflows. For instance, a workflow for code quality (e.g. triggered on any code change) and a workflow for releasing (e.g. triggered when a developer adds a new git tag). Workflows are defined using YAML files stored in the code repository (in `.github/workflows`). Workflows define jobs that are started in parallel unless they are configured to depend on other jobs. By enabling fail-fast, when a job fails, all concurrent jobs are stopped to save resources. Indeed, when a job fails, the run is considered failed, unless configured otherwise. Jobs consist of steps that are run in order. Output (i.e. stdout, stderr) from steps is saved in the run log. The Github runner enriches the run log. For instance, it prints in the log various metadata

such as the runner version or the container image that populated the run environment. It also prefixes log lines with a timestamp indicating when the log line was emitted. Run logs are mainly useful when runs are failing: developers often investigate the run log to find the reason for the failure. In this chapter, we will analyze run logs to gain insights into the execution times of steps used in workflows.

Steps

Steps can be either action or shell steps. Unlike jobs, steps are executed in the order they are defined and in the same runner environment. When a step fails, the job is aborted, unless configured otherwise. In the remainder of the section, we further explain the shell and action steps.

Shell Shell steps offer great flexibility by allowing the use of scripting languages generally well-known by developers. The default shell interpreter depends on the runner environment (e.g. bash for Linux-based runners, PowerShell for Windows-based runners). Scripting languages allow developers to implement custom CI/CD strategies similarly to how they do on their computers. For instance, in the JavaScript ecosystem, installing dependencies and running tests can be done using the shell command `npm install && npm test`). GHA runners allow the injection into shell code of contextual information using variable interpolation, e.g. git reference that triggered the run, repository secrets, and environment variables. Shell command returns codes that will enable GHA to decide if the step has succeeded or failed. In conclusion, step shells are well-known by developers to set up CI/CD pipelines, but they do not offer off-the-shelf features for common CI/CD tasks.

Action GHA encourages the use of actions. Actions are off-the-shelf steps for common CI/CD tasks. This helps to decrease the development efforts required to use GitHub Actions. For instance, `checkout` action allows to pull repository code in the job environment. By default, only a single commit is fetched for the reference that triggered the workflow. Actions can be customized using a `with` block in the YAML workflow definition file. For the `checkout` action, developers can configure the action to pull git submodules. Available parameters are usually defined in the documentation of the action. Actions can be stored in any GitHub repository, including the repository where the action will be used. Versioning is done through usual git references (e.g. branch, tag, commit hash). Therefore, the action version can refer to code that can be modified (e.g. git tag). Unlike shell steps, it might be challenging to ensure the integrity of the code that will be run during workflow execution when using action from other repositories. Under the hood, actions can rely on JavaScript code, custom Docker images, or other actions (composite action). However, this is not directly visible to end-

Figure 3.1: Extract of GitHub Action log file. Each line is prefixed with the time of execution.

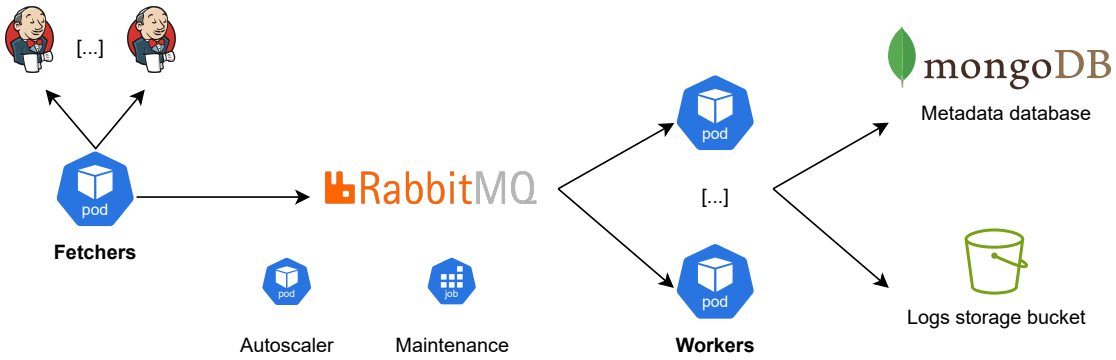

Figure 3.2: High-level overview of the architecture of the Jenkins Scanner.

users. Therefore, actions are roughly black boxes that perform a specific task. To facilitate the discovery of actions, GHA provides a marketplace. Action owners must register in the marketplace for their actions to be available. As of November 2023, there were 20,782 actions available in the marketplace. However, only 809 actions are coming from verified creators. As actions can access sensitive data such as CI/CD secrets, developers prefer actions from verified creators [151]. Therefore, actions might not be available for less common needs. In this context, developers can fall back to shell steps.

Runs and Logs

When workflows are triggered, the jobs are executed by GHA runners. GHA runners collect all the standard output produced as the result of executing the steps in log files. Interwoven with the standard output of steps, GHA runners put dedicated information such as which job and step is running, and which version of the environment has been set up.

3.2 Industrial dataset of Jenkins builds

This thesis work was done in collaboration with Amadeus, a world-class software company. Considering the lack of available public dataset of CI/CD builds, as stated in Section 2.2, we started by building a dataset thanks to CI/CD system of our industrial partners. Amadeus has about 10 million CI/CD builds every year, with many build failures.

3.2.1 Scraping CI/CD builds

We developed a tool named "Jenkins Scanner," programmed in Python, to extract CI/CD build data from Jenkins. The architecture of the scraper consists of three main elements: fetchers, a queueing system, and workers. Autoscaler and maintenance jobs help to ensure automated scaling of the number of workers and cleanup of old data. Figure 3.2 depicts an overview of the architecture. The fetcher's role is to identify and collect builds for scraping by accessing the `/rssAll` endpoint on the Jenkins master servers, which provides information on the latest builds executed on the server. Builds are then forwarded to the queueing system, built upon RabbitMQ, a well-established and open-source solution. The processing of builds is carried out by a dynamic number of workers that dequeue tasks from RabbitMQ. The number of workers is autoscaled based on queue size to keep the time to import low. The time to import is between the end of the build and when the scanner processes the build. We aim to keep it small to enable real-time CI/CD performance analysis. These workers interact with the Jenkins master's API to gather build data. Build logs are stored in an S3 bucket, and metadata are stored in a MongoDB database. This approach enables efficient management of millions of CI/CD builds. Furthermore, it allows leveraging MongoDB's search capabilities to filter builds based on specific criteria. Due to the industrial dataset's proprietary nature, we cannot disclose specific details like failure patterns or data examples for this dataset. The following section details how we automatically label a part of failed CI/CD builds.

3.2.2 Automatic labeling of instabilities failures

Labeled data is needed for training a supervised learning method, as well as for performance validation in unsupervised settings. We propose a method to label a part of historical builds automatically. Historical builds refer to builds with at least a subsequent build, i.e., builds except the latest builds of jobs. A failed build receives a label based on the code changes between it and the subsequent build, but only if the subsequent build succeeds. Therefore, our method applies to any build n satisfying these criteria:

$$\forall n \text{ such as } build_n \text{ failed and } build_{n+1} \text{ is successful}$$

If there is no code change between these builds (i.e., the build $n+1$ was started with the same codebase as the build n), we consider that the root cause is related to the infrastructure (non-deterministic); otherwise, we consider that the root cause is related to the code repository (deterministic). In other words, if a broken build is restarted without any modification and passes, it has failed because of a non-deterministic failure. Reliably detecting code changes between builds may be challenging. While Jenkins provides code changes between builds,

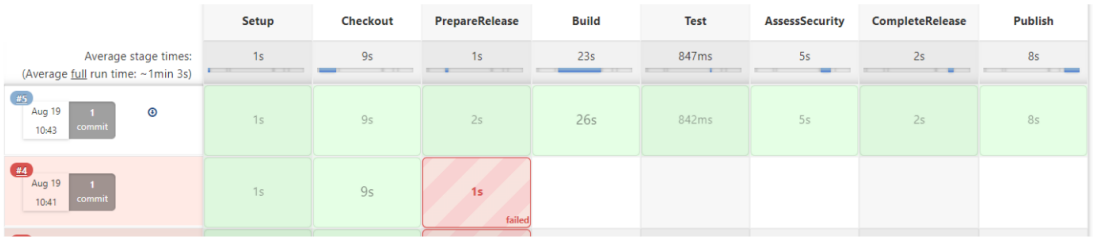

Figure 3.3: Example of a failed build (4) followed by a successful build (5). The root cause of the build 4 can be identified by analyzing the code change between the two builds.

during our experiment, it returned the wrong results in a few cases. It notably happened when git history was rewritten and force-pushed. Therefore, we retrieve git commit hash where the builds are based on, and compare them to detect code changes. It does not provide granularity at the file level (i.e., which files were modified), but it helps to detect code changes. If the git commit hashes are unavailable, builds are not considered for the labeled process. Pipeline configuration files can be injected at build time without committing to the code repository, impacting our ability to detect changes. However, this approach's security implications encourage enabling this approach only to a tiny subset of users. As a result, we believe it does not significantly impact the correctness of labels assigned to failed builds. In practice, this method helps to label around 10% of historical failed builds. We made the hypothesis that these builds are representative of all failed builds.

3.3 Public dataset of GitHub Actions runs

Many CI/CD orchestrators are available: GitHub Actions, Gitlab, CircleCI, Travis, Drone, Jenkins, among others. GitHub Actions (GHA) was launched in 2019 and has gained significant interest from open-source projects since many of them use GitHub to host their source code and GitHub Actions is simplified by the extensive integration between services. In addition, GitHub offers a free tier (2,000 minutes of CI time per month) suitable for a wide range of uses. Therefore, we focused our study on GitHub Actions.

GitHub Actions rely on workflows. A workflow is composed of jobs that can be run in sequence or in parallel. Each job contains a sequence of steps. A step is either an action or a shell code. Actions aim to simplify the use of GHA by providing off-the-shelf steps for common CI/CD tasks. For instance, Actions exist for repository checkout, installing a specific Python version, or building and pushing a docker image to a registry. However, developers can still write shell code using shell steps. It allows developers to execute commands as CI/CD steps. This can address a need not covered by actions or ease the migration of existing CI/CD pipelines. Therefore, there are two ways to define a job step: using a pre-defined Action or using shell code. The usage of action steps has been investigated in the literature [24, 34, 151]. However,

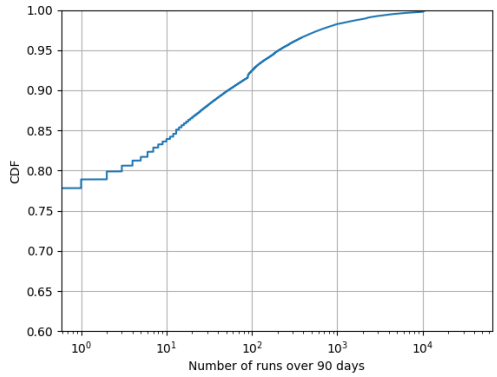

Figure 3.4: Cumulative Distribution Function (CDF) that represents the number of GHA runs for the 239,106 repositories with more than 100 stars. The x-axis is log-scaled. The y-axis starts from 0.6. Using our repository selection criterion (≥ 30 runs), we selected 12 % of repositories for analysis.

to the best of our knowledge, there is no large-scale analysis of GitHub Actions workflows that also includes shell steps in addition to actions.

Our goal is to dive into the use of shell commands in workflows and analyze their execution time. As a consequence, we cannot rely on the YAML workflow definition files, as it is commonly done by the other studies of the GHA ecosystem [24,33,99], because they do not contain any runtime information. Additionally, GHA's API only provides the execution time at the job level, and we explained previously that shell commands are used at the finer-grained step level. The only place where the information necessary for our study is located is in the run logs of the repositories, where lines are timestamped, allowing us to reconstruct the execution time of steps.

In the following sections, we explain how we select a set of repositories (see 3.3.1), how we extract their run logs (see 3.3.2), and how we analyze this data to answer our research questions (see 3.4).

3.3.1 Selection of repositories

Considering the large number of public repositories in GitHub, we have selected only a subset of repositories relevant to the study. To discover repositories, we leverage GitHub Search [28]. It provides a search engine to discover public GitHub repositories based on criteria (e.g. number of stars, license, topics). It can search for GitHub repositories with more than 10 stars only. Using the search engine, we have selected repositories with more than 100 stars in terms of popularity. We obtained a list of 239,106 repositories. For each repository with more than 100 stars, we have extracted the number of GHA runs (the sum of runs of all workflows) over

the latest 90 days using GHA's API. We successfully scraped 239,106 (99.3 %) repositories. We implemented a retry mechanism to mitigate the impact of transient errors (e.g. HTTP 502 error code). For 0.7 % of the repositories returned by GitHub Search, we could not check for the use of Github Actions because of scraping errors: HTTP 404 (0.53 %), HTTP 410 (0.10 %), HTTP 403 (0.06 %), HTTP 502 (0.01 %) and HTTP 451 (0.00 %). This extraction was done between the 1st and the 5th of October, 2023. The distribution of the number of runs for repositories is depicted in Figure 3.4. We selected repositories with more (greater or equal) than 30 runs in the last 90 days. We identified 28,683 (12 %) repositories. We excluded 361 fork repositories. We aim to exclude toy usage of GHA by selecting repositories with more than one run every three days (on average). It is worth mentioning that some workflow can have less than 30 runs over the 90-day period, e.g. releasing workflow. However, we made the selection at the repository level. If the repository has more than 30 runs over 90 days, we scrape runs of all workflows, even if they have only a few runs over the period. Therefore, we identified 28,322 repositories with over 30 runs in the last 90 days. The list of repositories is not updated after the initial construction. In other terms, we did not search for new repositories that could satisfy the run activity criterion after the initial discovery. We identify a *workflow* by its repository and its path in `.github/workflows`. Therefore, if two workflow files in two repositories are identical, we treat them as different workflows. In comparison, Bouzenia et al. [17] studied 950 repositories.

We scraped all workflows present in the selected repositories. However, we did not scrape all runs for each workflow. Indeed, some workflows have a very large number of runs (e.g. dozens of thousands in 90 days). Therefore, we scraped only the 5 most recent runs of each workflow. Downloading 5 runs per workflow allows us to compute average values (e.g. step execution time) for the same workflow. We limited run logs ZIP archive download to 20MB. If the archive is larger than this threshold, the download is aborted and the run is ignored. Large log archives are rare (<1 % of downloads were aborted because the archive was too large) and significantly impact the run processing pipeline performance. Therefore, we choose to ignore very large logs.

3.3.2 Downloading runs

We target to discover new runs as fast as possible. Indeed, a part of the information we scraped can be deleted after the run is finished. Run logs are kept 90 days by default, but repository owners can reduce the retention to 1 day. When a run is triggered by a code change (e.g., pull request, commit), the source commit can be deleted (e.g., pull-request is merged with a squash commit). Therefore, we target to scrape runs as fast as possible after the run is finished. The run processing pipeline is illustrated in Figure 3.5. The processing path (i.e., number of workers) can be adapted to keep processing time low. We only scrape runs with `conclusion` equal to success, failure, or timed_out. Indeed, runs with different `conclusion`

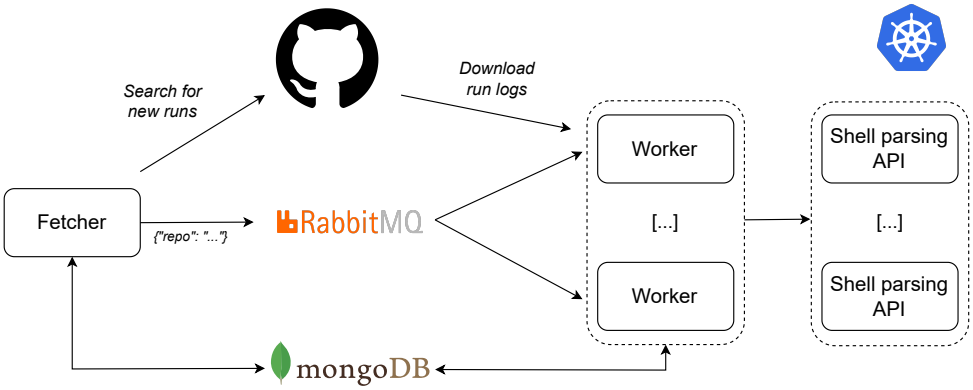

Figure 3.5: Overview of the run scraping process. New runs are discovered by fetcher component. When a repository contains new runs to process, repository processing is triggered by publishing a message to RabbitMQ queue. Workers are deployed on Kubernetes and automatically scaled to ensure runs are processed quickly.

do not have a log, thus making them useless for the scope of the study. GitHub imposes a rate-limit of 5,000 API call per hour per account. As recommended by GitHub [62], we use conditional requests to detect whether a repository contains new runs. It helps to reduce resource usage (i.e., by returning an empty response with HTTP code 304) when no new run is available. Furthermore, conditional requests do not count against the rate limit. Conditional requests rely on `Etag` header. We discovered that `Etag` values are related to the token used in the request: different tokens will lead to different `Etag` for the same resource. For each run, we download run metadata and run logs. GitHub allows downloading run logs as a ZIP archive. The archive contains logs for jobs and steps. Step logs are redundant with job logs (i.e., step logs are present in job logs), We postulate that GitHub uses this file structure to ease the processing of logs. However, as the ZIP algorithm compresses files independently, we extract and re-compress log files in a tar archive with gzip compression.

3.3.3 Dataset metrics

We collected GitHub Actions runs to answer the research question detailed in Section 4.1. We identified 28,322 code repositories with more than 30 runs in 90 days, excluding forks. These repositories contain 151k workflows. There are 20 different programming languages. The number of repositories, the number of workflows, and the number of runs analyzed (i.e., log was available) are depicted for each programming language in Figure 3.1. The identification of the main programming language used in the repository is provided by GitHub. We attempted to retrieve 5 runs per workflow. However, we were not able to retrieve exactly 5 runs per workflow. Some workflows are rarely used, thus, there are fewer than 5 runs in the last 90

Table 3.1: Breakdown of the programming languages of repositories considered for the study. Languages are sorted by the number of repositories.

Language	# Repositories		# Workflows		# Runs	
Python	5.52k	19.50%	22.34k	19.21%	97.31k	18.95%
TypeScript	4.44k	15.66%	18.90k	16.25%	84.19k	16.40%
Go	2.95k	10.41%	15.77k	13.56%	70.38k	13.71%
JavaScript	2.78k	9.80%	9.78k	8.41%	44.2k	8.61%
C++	2.14k	7.55%	9.22k	7.93%	40.98k	7.98%
Java	2.07k	7.32%	8.34k	7.17%	36.66k	7.14%
Rust	2.02k	7.14%	7.97k	6.86%	34.9k	6.80%
C	1.25k	4.41%	5.17k	4.45%	23.3k	4.54%
PHP	1.2k	4.22%	4.56k	3.92%	20.12k	3.92%
C#	1.1k	3.88%	3.79k	3.26%	15.88k	3.09%
Shell	735	2.60%	2.71k	2.33%	11.95k	2.33%
Ruby	639	2.26%	2.3k	1.98%	10.45k	2.04%
Kotlin	634	2.24%	2.27k	1.95%	9.72k	1.89%
Dart	319	1.13%	1.38k	1.18%	5.93k	1.15%
Swift	250	0.88%	250	0.69%	3.29k	0.64%
Elixir	126	0.44%	126	0.30%	1.56k	0.30%
Objective-C	60	0.21%	60	0.24%	1.21k	0.24%
Nix	56	0.20%	56	0.15%	56	0.15%
Groovy	36	0.13%	36	0.12%	36	0.12%
Smalltalk	6	0.02%	6	0.01%	6	0.01%
Total	28,322		116,259		513,492	

days. Therefore, run logs have expired and cannot be downloaded. In practice, we retrieved on average 4.41 runs per workflow (std: 1.28). We were able to retrieve at least one run log for 116k workflows (77%). We collected 513,492 run logs. The total uncompressed size is 640 GB. We focused only on workflows with at least one run log: 116k workflows over 151k workflows. For the following metrics, we consider one run per workflow. Indeed, it allows to maximize the diversity as runs from the same workflows are likely to share the same jobs and steps. Therefore, from 116,259 runs (i.e., 1 per workflow), we extracted 348,909 jobs. There are, on average, 3,3 jobs per workflow (median: 1.0, std: 10). From jobs, we extracted 2,327,747 steps. There are, on average, 6,7 steps per job (median: 5.0, std: 5.4).

Dataset Availability

The dataset and code enabling the reproduction of all experiments presented in this chapter are available at https://purl.org/github-actions-shell-study. This dataset is the biggest one of GitHub Actions containing logs which is currently available.

3.4 Analyzing GitHub Actions runs

3.4.1 Introduction

This chapter explores the use and properties of actions and shell steps. Furthermore, we analyze run logs from workflows of 25k projects to gain insights about runtime performance, such as step execution time. In this work, we aim to answer to the following research questions:

RQ3.1: *What is the breakdown between actions and shell steps?* Developers can choose between using actions and shell steps but what is the breakdown between these two options? Is the breakdown significantly modified between projects with different properties, such as programming language? What are the most used actions? We also analyze shell steps to extract the commands being used together with their arguments. We aim to answer to the following questions: What are the most used commands? What are the arguments? Can we classify command behaviors using command names and arguments?

RQ3.2: *What is the execution time breakdown of workflows?* After exploring the breakdown between steps, we focus on execution time analysis and we analyze run logs to gain insights into the execution time at the step level. What is the breakdown between action and shell in terms of execution time? Is the breakdown modified across ecosystems? What are the high-level categories that take most of the pipeline execution time?

RQ3.3: *What is the variability of the execution time of a step?* We finally explore the variability of the execution time of the most used steps among 111k workflows over 25k projects corresponding to the subset of the top 10 programming languages used in more than 1000 repositories. How to determine if two steps are identical since steps can have the same purpose without being byte-by-byte identical? What is the variability of execution times for actions and shell steps? Does step volatility analysis enable targeted optimization efforts?

In addition to the run metadata provided by the GitHub API, we parse the job logs. Initially, we extracted structural information from the logs using specific regular expressions. These included details such as the container image employed for execution, downloaded GitHub Actions, and the shell and GitHub Action steps.

Concerning the shell step, we retrieved the commands embedded in the shell. For instance, given a shell step: `cd /home/;git clone myrepo;`, we captured commands `cd` and `git`. Additionally, we identified command annotations such as `GIT-CLONE` and categorized those commands under `FILE_SYSTEM` and `VERSION_CONTROL`. To achieve this, we utilized an existing shell parser [46] designed for analyzing shell scripts within Dockerfiles. This parser provides an Abstract Syntax Tree (AST) representation of the shell which allows us to precisely analyze the shell without relying on unprecise regular expressions. The parser extracts: (i) all bash commands, (ii) annotations for 91 types of commands, and (iii) 39 categories for the

types of commands.

Our parsing success rate for shell steps stands at 99.0 %. Parsing issues occurred due to parser exceptions (0.96 %) or invalid requests (0.04 %). Remarkably, the parser adeptly annotated 70.34 % of all commands. This extensive coverage ensures a comprehensive understanding of the shell steps used in GitHub workflows.

3.4.2 Categorizing shell commands

In order to get a better understanding of the shell commands that developers are using inside their workflows, we categorized the most common commands.

To do so, for each shell command that is supported by the shell parser, we assign a category by an iterative and discovery process. Then we look at the different scenarios that those commands support for example, `npm` has the scenario `test` and is called `npm test` and we assign a category for those scenarios when it is relevant. In total, we created 39 categories for 82 commands and 88 additional scenarios. The annotations cover 70.34 % of the commands found in the GHA dataset and therefore are categorized into at least one category. The annotations and categories of the commands are available in our repository.

We did not categorize Actions with the same taxonomy since they operate at a different scope. Shell commands are mostly a single action while Actions combine multiple commands to form an abstraction. Therefore, it is difficult to create a comprehensive taxonomy that groups different levels of abstractions.

For clarity, we only run the analysis on the 5 most represented programming languages. However, the public dataset contains data for all programming languages presented in Section 3.3.3.

3.4.3 Breakdown between actions and shell steps

In this initial research question, we delve into the examination of shell and action steps.

The primary outcomes of this research question are showcased in 3.2, illustrating the utilization of shell steps versus GitHub Actions across programming languages. Shell steps are overall more popular than action steps with 56 % of usage. This trend is particularly pronounced in Rust, C, and C++, where over 60 % of the steps employ shell. Conversely, Go stands out with the lowest shell usage at 46.75 %. However, we did not identify large variability in terms of the median number of commands or actions between languages that could explain this difference. Therefore, our hypothesis is that the prevalence of shell steps correlates with the maturity of the GitHub Actions ecosystem for specific languages. For instance, C++ lacks popular GitHub

actions tailored to its language, while Go boasts top GitHub Actions that cater uniquely to its needs, indicating a potential gap for C++ developers.

Our exploration into shell and action steps persists as we delve into the most frequently employed steps per language. 3.3 showcases the top 10 popular GitHub Actions, while 3.4 highlights the top 10 annotated shell commands. Not surprisingly, the most preferred actions revolve around checkout, cache, and artifact management.

Interestingly, there's a notable absence of actions responsible for executing tasks like building or testing. The majority of actions are linked to continuous integration tasks, such as setting up an environment and managing artifacts. Indeed, almost 20 % of workflows feature actions dedicated to uploading artifacts. The establishment of a specific environment is present in nearly 50 % of instances across all languages, except for C++, which lacks a dedicated setup.

However, the top actions alone don't offer a comprehensive view of developer practices within GitHub Actions. A clearer picture emerges from the top 10 shell commands, revealing a variety of language-specific commands like `pytest`, `go test`, and `cmake`. Additionally, `yarn` enjoys more popularity in TypeScript workflows compared to JavaScript ones, and while `tar` is widely used across all languages, it still lags behind `echo` in terms of popularity.

These commands are dedicated to executing tasks that developers commonly perform during development, such as installing, building, and testing. We assume that developers are familiar with these commands and prefer using them directly instead of relying on external actions.

Based on these observations, researchers aiming for a clear picture of the activity performed by developers in GitHub Actions should consider analyzing the shells instead of solely focusing on the action steps, as is currently the practice.

Answer to RQ3.1. We note that shell usage constitutes 56 % of the steps in GitHub Workflows. This finding is intriguing, given that the research community predominantly concentrates on GitHub Actions to scrutinize developer practices in continuous integration. Additionally, distinctions arise in the types of actions carried out with GitHub Actions as opposed to shell steps. Actions are tailored for tasks associated with continuous integration, whereas shell steps predominantly pertain to commonplace development tasks.

3.4.4 Execution time of workflows

In this second research inquiry, our focus lies in analyzing the predominant steps where workflows allocate most of their time. The objective is to pinpoint the commands and actions with the highest execution intensity. We anticipate observing actions and commands associated

Table 3.2: Usage of shell steps among programming languages. Medians reflect the number of unique actions (resp. commands) used per workflow.

Language	% Shell	Median # Actions	Median # Commands
Rust	66.4 %	3	4
C	62.51 %	2	4
C++	61.15 %	2	4
PHP	57.66 %	3	2
Python	56.27 %	3	3
TypeScript	51.58 %	3	2
C#	49.68 %	3	2
JavaScript	49.42 %	3	1
Java	48.07 %	3	2
Go	46.75 %	3	2
All languages	56.01 %	3	2

Table 3.3: Top 10 of the most used actions in workflows grouped by programming language. The percentage indicates the ratio of workflows where the action is used over the total number of workflows. The cells in bold are the elements that are unique in the top 10.

#	Python		TypeScript		Go		JavaScript		C++	
1	checkout	88 %	checkout	85 %	checkout	87 %	checkout	84 %	checkout	84 %
2	setup-python	52 %	setup-node	54 %	**setup-go**	50 %	setup-node	45 %	upload-artifact	45 %
3	upload-artifact	16 %	cache	16 %	upload-artifact	14 %	upload-artifact	13 %	cache	13 %
4	cache	12 %	upload-artifact	13 %	cache	11 %	cache	10 %	setup-python	10 %
5	codeql/init	6 %	**pnpm/action-setup**	10 %	**setup-buildx**	9 %	codeql/init	8 %	deploy-pages	8 %
6	codeql/analyze	6 %	codeql/init	6 %	**docker/login**	9 %	codeql/analyze	8 %	upload-pages-artifa	
7	**codecov**	6 %	codeql/analyze	6 %	**golangci-lin**t	8 %	deploy-pages	7 %	download-artifact	7 %
8	setup-node	5 %	deploy-pages	5 %	codeql/init	7 %	upload-pages-artifact	7 %	codeql/init	7 %
9	deploy-pages	5 %	upload-pages-artifact	5 %	codeql/analyze	7 %	**jekyll-build-pages**	4 %	codeql/analyze	4 %
10	upload-pages-artifact	5 %	codeql/autobuild	4 %	**setup-qemu**	7 %	codeql/autobuild	4 %	setup-node	4 %

Table 3.4: Top 10 of the most used shell commands in workflows grouped by programming language. The percentage indicates the ratio of workflows where the command is used over the total number of workflows.

#	Python		TypeScript		Go		JavaScript		C++	
1	PIP-INSTALL	48 %	ECHO	25 %	ECHO	28 %	ECHO	22 %	ECHO	32 %
2	**PYTHON-MODULE**	33 %	YARN-RUN-SCRIPT	19 %	MAKE	25 %	NPM-INSTALL	18 %	APT-INSTALL	27 %
3	ECHO	26 %	**YARN-INSTALL**	15 %	**GO-TEST**	10 %	NPM-RUN	18 %	CD	27 %
4	PYTHON	19 %	NPM-RUN	14 %	CD	9 %	NPM-CI	15 %	**CMAKE**	27 %
5	**PYTEST**	11 %	NPM-INSTALL	13 %	**CURL**	8 %	NPM	13 %	MKDIR	23 %
6	CD	10 %	NPM-CI	12 %	MKDIR	7 %	TAR	9 %	APT-UPDATE	20 %
7	APT-INSTALL	10 %	CD	8 %	APT-INSTALL	6 %	YARN-RUN-SCRIPT	9 %	**CMAKE-BULD**	18 %
8	MAKE	9 %	NPM-RUN-BUILD	8 %	**GO-BUILD**	6 %	NPM-RUN-BUILD	8 %	MAKE	16 %
9	TAR	7 %	NPM	7 %	**GIT**	6 %	**TAR-VERBOSE**	8 %	PIP-INSTALL	15 %
10	APT-UPDATE	7 %	TAR	6 %	TAR	6 %	**TAR-COMPRESS**	7 %	TAR	11 %

with building and testing processes, yet we lack insights into other steps that might impede continuous integration.

3.6 illustrates the cumulative distribution of shell commands. It reveals that the majority of steps involve only a few commands, allowing us to disregard shell steps with more than one command and therefore improving the accuracy of our measurements.

3.5 outlines the proportion of time GitHub dedicates to executing shell steps versus GitHub Action steps. Surprisingly, the proportion surpasses our expectations. We anticipated a proportion similar to what we presented in RQ3.1, around 50 %, but instead, the time distribution ranges from 70.7 % for JavaScript to 87.6 % for Python. Globally, 82.6 % of the time is consumed by shell steps. This indicates a major disproportion of the time spent in the commands and therefore confirms our analysis in RQ3.1 that the tasks performed by GitHub Action steps are different from shell steps.

To understand the difference we delve into specific commands with the two last columns of 3.5. Those columns indicate the proportion of time spent in each shell command category (see 3.4.2 for extraction methodology). It shows that shell steps predominantly allocate time to testing and building processes. 3.6 presents the 10 most time-consuming commands for each language. We observe some variation depending on the programming language. In Python, the CI primarily spends time on testing and installing dependencies, while for Go, it focuses on building and testing. It is interesting to note that commands from other ecosystems enter the top 10, for example, we can see that C++ has `Python` and `npm` in rank 3 and rank 10 respectively, and Go also has `npm` in rank 8. We observe less diversity of ecosystem for `Python`, `TypeScript` and `JavaScript`

In general, these observations indicate that developers continue to opt for shell steps when undertaking tasks such as testing, building, or installing, rather than relying on GitHub Actions. As highlighted in RQ3.1, the majority of GitHub actions are associated with tasks like configuring the environment, setting up the ecosystem, or managing artifacts. These activities appear to be more geared toward the CI infrastructure, while the commands executed in shell steps are more generic and could be performed by developers directly.

Answer to RQ3.2. The key finding from this research question underscores that a significant 82.6 % of the time is dedicated to shell steps. Across various programming languages, we did not discern substantial variations. When examining the steps consuming the most time, we observed that commands associated with building, testing, and installing take precedence. This implies that these tasks are not executed through GitHub Actions; instead, developers opt for a shell step to carry them out.

Table 3.5: Presents the usage of shell steps among programming languages according to the amount of time spent and the time spent in each category of shell commands.

#	Language	% Shell	Shell Category	% Time
1	Python	87.6%	BUILD TOOL	18.9%
2	TypeScript	75.8%	BUILD	18.0%
3	Go	76.3%	DEP MANAGER	15.5%
4	JavaScript	70.7%	TEST	7.4%
5	C++	86.5%	EXECUTE	6.6%
6	Java	82.9%	DEV TOOL	4.8%
7	Rust	78.7%	INSTALL	4.2%
8	C	84.1%	DOCKER	2.3%
9	PHP	74.3%	CODE COVERAGE	0.7%
10	C#	78.3%	FILE SYSTEM	0.2%
	All languages	82.6%		

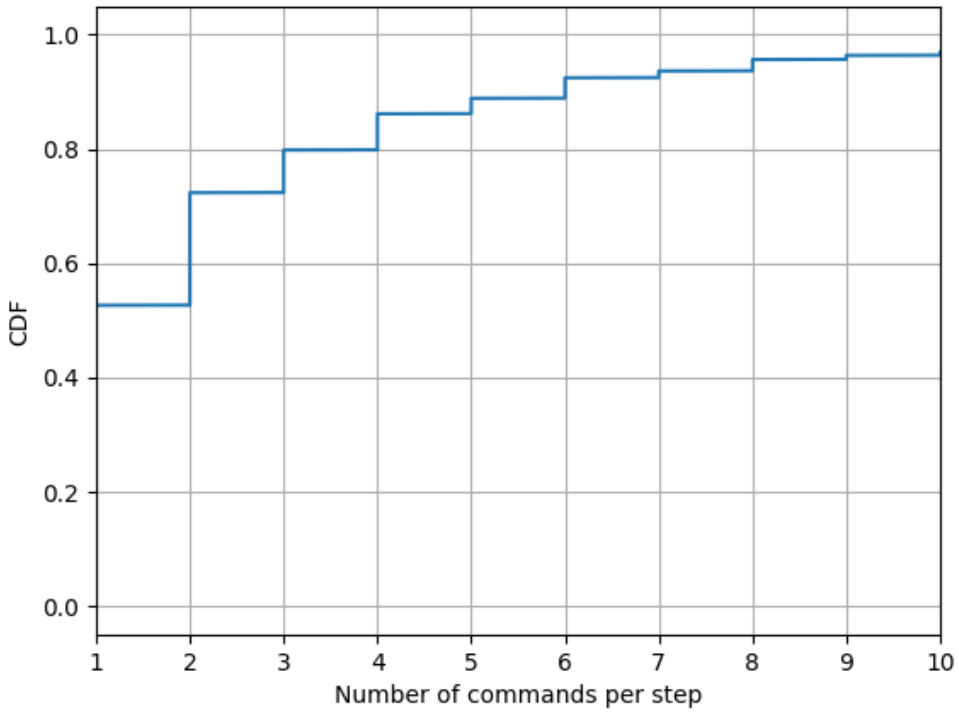

Figure 3.6: Cumulative Distribution Function (CDF) that represents the number of commands in shell steps for all programming languages. We observe that 53 % of shell steps have one command and 97 % have 10 steps or less.

Table 3.6: Top 10 of most time-consuming shell commands per language. We compute the time as the proportion of time spent inside each GitHub Action Run.

#	Python		TypeScript		Go		JavaScript		C++	
1	Other	20.7 %	Other	16.0 %	MAKE	13.9 %	Other	10.6 %	Other	17.1 %
2	PYTEST	4.1 %	YARN-RUN-SCRIPT	8.2 %	Other	10.1 %	NPM-RUN	7.6 %	CMAKE-BULD	7.9 %
3	PIP-INSTALL	3.9 %	NPM-RUN	5.0 %	GO-TEST	4.5 %	NPM-INSTALL	5.8 %	MAKE	2.8 %
4	MAKE	2.7 %	YARN-INSTALL	4.2 %	GO-BUILD	1.3 %	NPM-CI	3.7 %	PYTHON	1.6 %
5	PYTHON	1.5 %	NPM-CI	3.0 %	ECHO	0.7 %	NPM	3.3 %	CMAKE	1.3 %
6	DOCKER-BUILD	0.9 %	NPM-INSTALL	2.4 %	DOCKER-BUILD	0.4 %	YARN-RUN-SCRIPT	3.3 %	CTEST	0.7 %
7	COVERAGE	0.4 %	NPM	1.7 %	MAKE-TEST	0.3 %	YARN-INSTALL	2.5 %	PIP-INSTALL	0.5 %
8	ECHO	0.3 %	NPM-RUN-BUILD	1.3 %	NPM-INSTALL	0.3 %	NODE	1.2 %	DOCKER-BUILD	0.3 %
9	PYTHON-MODULE	0.3 %	MAKE	0.8 %	GO-MOD	0.2 %	MAKE	1.1 %	ECHO	0.3 %
10	MAKE-TEST	0.1 %	NODE	0.5 %	GO-VET	0.2 %	ECHO	0.9 %	NPM-INSTALL	0.2 %

3.4.5 Steps' execution time variability

In this final research question, we study the variability of the execution time of the steps. We explore the variability as we postulate that steps with highly variable execution times will likely be candidates for optimization. We explore the execution times of steps among all runs we analyzed. We computed the standard deviation of execution times at the step level. To ensure we measure statistically significant results, we consider steps with at least 100 executions. Furthermore, we filter out steps that appear in less than 2 % of workflows in order to focus the study on popular ones.

We compare the execution times of the same step among workflows across repositories. Therefore, we have to define the equality property between steps. For actions, we consider steps identical if both actions have the same repository, the same name, and, if any, the same folder. We consider equality does not require the same action version. Indeed, we do not expect the purpose of the action to change between versions, even for major versions. For instance, steps with `actions/checkout@v3.1` and `actions/checkout@v4` are considered identical, but `github/codeql- action/autobuild@v2` and `github/codeql-action/analyze@v2` are not.

For shell steps, the equality property is more challenging. Unlike actions, shell step codes tend to be highly tailored to the workflow they were designed for. Therefore, it is unlikely that shell step codes from different workflows will be byte-by-byte identical for commands, arguments, and environment variables. Furthermore, we extracted execution duration at the step level. Therefore, we can only identify the execution of commands when there is only one command per step. When multiple commands are used in the same step, it is challenging to identify parts of the log for each command. Indeed, commands might not output anything or have the same logging format. Even if hypothesizing that different commands have different logging formats, detecting the change of the log format in the step log is still challenging at scale. However, for steps, the runner adds log messages to indicate the beginning of a step. Therefore, we can reliably determine the execution duration of a step. As a result, we do the variability analysis only on shell steps with one command. However, the same command might have very different behavior. For instance, `pip install` and `pip uninstall` are the same commands but with different arguments. The latter command will probably be significantly faster. Therefore, commands should be compared without consideration of their arguments. However, a byte-by-byte comparison of arguments is not satisfactory. Indeed, arguments can impact the execution time, but it mainly depends on the argument. For instance, enabling verbose logging (usually using `-v`) will have a negligible impact on execution time. To distinguish behaviors, we leverage command annotations given by the shell parser. Therefore, we consider shell steps to be equal only if both have only one command and if they have the same command annotation. In the following sections, we discuss results for actions and annotations.

Actions

We extract the top 10 of actions with the highest standard deviation of execution time. Results are shown in Table 3.7. We observe that `docker/build-push-action` has a significantly higher standard deviation than the following actions in the top volatile actions with 2127 seconds (about 35 minutes). As the name suggests, the action allows to build and push to a registry Docker images. Building container images is recent compared to building software. Indeed, Docker project was released in 2013. Therefore, we postulate that best practices for building Docker images are less widely spread and well-known by the developer community. Furthermore, Docker build time can be significantly impacted by configuration, such as the use of layer caching or the use of a shared base image. Therefore, we postulate that actions with higher variability depict practices where knowledge is sparse in the community. However, not all actions in the top 10 can be explained with the same reasoning. For instance, `github/codeql-action/autobuild` or `actions/download-artifact` have very few configuration parameters. Therefore, we hypothesize that high-volatility actions depend on factors other than configuration parameters. Factors include repository properties, such as code size, or infrastructure properties, such as available network bandwidth for the runner. On the other hand, actions with the lowest standard deviation of execution time are not good candidates for optimization strategies. Indeed, optimizing such actions will allow only a small execution time gain.

Interestingly, we still observe a relatively high deviation for setup actions such as `actions/setup-java` or `actions/setup-node`, with a deviation of 3min and 1.5min respectively. Those actions are provided by GitHub and we were expecting to see stable execution time. This could indicate high fluctuation in the GitHub Action infrastructure.

Command annotations

We study the execution time variability of commands. We identified commands by their annotations to better represent the command behavior. Results are shown in Table 3.8. The most volatile annotations are the Python test framework `pytest`, followed by build tools. We were expecting those fluctuations since the execution time of those commands is highly dependent on the project. For instance, the execution time of tests will likely depend on the number of tests. The main insights given by volatility indicate that repositories use different strategies for the same command behavior. For instance, a workflow that takes a long time for `pytest` can rework its test strategy (e.g., redundant tests hunting). In less volatile commands, system utilities are predominant.

Based on these results it is difficult to identify potential step optimization candidates. The execution time can depend on a number of factors outside that are difficult to study at the

Table 3.7: Standard deviation (std) of execution time for actions with more than 100 executions appearing in more than 2 % of the workflows of the top 10 programming languages. The first 10 rows represent higher std, and the last 10 represent lower std.

Action	Standard Deviation (s)
docker/build-push-action	2127.85
github/codeql-action/autobuild	880.81
actions/jekyll-build-pages	621.59
github/codeql-action/analyze	421.28
actions/setup-java	163.37
actions/download-artifact	136.48
actions/setup-node	99.46
actions/cache	79.15
actions/upload-artifact	68.87
actions/deploy-pages	49.61
⋮	
shivammathur/setup-php	32.27
actions/setup-python	30.97
actions/setup-go	24.1
codecov/codecov-action	21.17
actions/checkout	15.99
pnpm/action-setup	15.77
docker/setup-buildx-action	7.42
docker/setup-qemu-action	2.81
docker/login-action	1.02
actions/upload-pages-artifact	0.01

Table 3.8: Standard deviation (std) of execution time for annotations with more than 100 executions appearing in more than 2 % of the workflows of the top 10 programming languages. The first 10 rows represent higher std, and the last 10 represent lower std.

Annotation	Standard Deviation (s)
PYTEST	1092.21
NODE	875.48
NPM	780.01
MAKE	732.0
PYTHON	722.09
CMAKE-BULD	651.03
YARN-RUN-SCRIPT	518.43
NPM-RUN	493.88
MV	348.91
NPM-CI	230.81
⋮	
CAT	4.07
EXIT	3.44
GIT-GC	1.83
APT-UPDATE	1.75
TRUE	1.51
CHMOD	0.8
SED	0.44
MKDIR	0.4
JQ	0.35
TOUCH	0.1

macro scale (unlike the action steps). We recommend performing this variability of the execution time of the shell steps at the project level to detect anomalies.

Answer to RQ3.3. We show that execution times of the same step can vary significantly between workflows, indeed we observe a variation of 35 minutes for GitHub Action steps and 16 minutes for shell steps. We postulate that high execution time variability can imply optimization opportunities but project-specific analysis is required for variation for the shell steps.

3.4.6 Threats to validity

In this section, we discuss the threats to validity according to the taxonomy of [184].

Internal validity We measure the execution time of steps based on run logs. Most of the runs (99.25 % [24]) use cloud runners provided by GitHub. Cloud runners are a shared platform (multi-tenancy). Runners are expected to be the same for all users. However, GitHub might apply limitations such as CPU throttling for some runs (e.g., runs that highly impact the infrastructure). This can affect the reliability of the running time we measure for the steps. We mitigate the problem by computing the median execution time over multiple runs. We used a shell parser to extract commands, annotations, and categories for shell steps. Considering the high number of shell steps to analyze (3.4M), it is difficult to assess the quality of the parsing of all shell code. To mitigate the risk of incorrect parsing, we did sanity checks. For instance, when the shell parser extracts exactly one command from the shell code, we extract the number of shell separators using a regular expression (i.e., no shell separator is expected when only one command is identified).

External validity A potential threat is that the study period is non-representative due to abnormal activity on GitHub Actions. We reduce this risk by collecting data during a relatively large period of time: 90 days. During that period, we monitored the GitHub announcements to ensure that nothing abnormal was happening.

We also only focus on projects with more than 100 stars from GitHub. However, to ensure that our results still generalize as much as possible, we selected a large and diverse set of projects in various languages, across a wide range of projects on GitHub. However, all of our projects are open source, so we cannot make claims about how our results might generalize to proprietary projects.

3.5 Summary

Continuous integration and deployment (CI/CD) has gained traction in recent years in both the open-source community and industry. To support a seamless CI/CD experience, GitHub launched GitHub Actions (GHA) in 2019. In 2023, nearly 12 % of projects with more than 100 stars regularly use GitHub Actions. A notable feature of GHA is the concept of *actions* that are reusable building blocks for CI/CD workflows. Several studies have delved into the use of actions in CI/CD pipelines. However, despite the existence of *actions*, CI/CD workflows also use shell commands. While being the most traditional way (and sometimes the only way in some CI/CD infrastructure), there is, to the best of our knowledge, no existing study of how developers use such commands in Github Actions workflows. In this work, we fill this knowledge gap by performing a comprehensive analysis of the use of shell commands inside CI/CD workflows of GHA. We investigate to what extent shell commands are used, how much of the execution time they account for, and how volatile this execution time is on a new dataset we release to the community composed of 116k workflows spread into 28k repositories across

20 programming languages. We find out that even in GHA where reusable actions are available, 56 % of workflows' steps are shell commands, and they account for 82 % of the execution time. Moreover, we find out that some shell commands have a very volatile execution time, raising questions about optimization opportunities in their configuration.

Chapter 4

Classification of CI/CD build failures

Previously, Section 1.4 discussed the essential characteristics of CI/CD failures. This chapter is dedicated to distinguishing between deterministic and non-deterministic failures. We examine two distinct methodologies for categorizing these failures. Further, in Section 4.3, we investigate the application of Natural Language Processing (NLP) techniques for analyzing and categorizing build logs. Section 4.4 introduces an ontology to define CI/CD ecosystem knowledge. Utilizing this ontology alongside Knowledge Graphs, we enhance the process of classifying build failures. Finally, we demonstrate how identifying failure causes can contribute to the automatic fix of CI/CD failures under certain conditions.

4.1 Introduction

Categorizing build failures is an effective method to support operators in troubleshooting failures. Furthermore, implementing automated systems for classifying these failures plays a significant role in identifying failure patterns at scale or initiating automatic corrective measures. To carry out such classification, it is imperative to delineate distinct classes. The criteria for defining these classes vary widely, from a broad categorization to a more detailed, fine-grained approach. For instance, common failure class sets include:

1. The nature of failures, distinguishing between deterministic and non-deterministic types.
2. The location of the error within the code, categorized into broad areas such as source code, test code, or configuration code.
3. The specific paths or files involved, like *JenkinsFile*, *setup.cfg*, or *src/api.py*, providing a precise identification of the failure points.
4. The team best qualified to resolve the build failure

5. The network component responsible for the build failure, such as the source code management system, the artifact management system, or the dependency proxy.

In this chapter, we focus on distinguishing deterministic from non-deterministic failures. Identifying non-deterministic CI/CD failures is crucial because it ensures the software development process's stability, reliability, and efficiency. Non-deterministic failures, also known as flaky builds, can lead to significant delays in software release cycles, increased costs, and reduced developer productivity due to the time spent troubleshooting and rerunning tests. Moreover, these failures can erode trust in the CI/CD process, making it difficult for teams to rely on automated builds and deployments to make software quality and readiness decisions.

Failures of a deterministic nature will be designated as the *REPO* category, whereas failures of a non-deterministic nature will be classified as the *EXTERNAL* category. The rationale behind these designations stems from the anticipated location of corrective actions: either within the software's code repository or external to it. For example, deterministic failures typically necessitate modifications within the code repository, such as correcting syntax errors. On the other hand, non-deterministic failures frequently pertain to aspects of the CI/CD ecosystem that exist beyond the code repository, including CI/CD worker configurations or unpredictable errors. It's important to recognize that non-deterministic failures may also be resolved through changes to the code in the source repository, just as deterministic failures can sometimes be addressed without altering the repository's code. Hence, these terms facilitate easy reference to the types of failures and the general location for interventions in most instances while not rigidly specifying the origins of these issues. The following section explores the strategies to measure the performance of classification tasks.

4.2 Background

4.2.1 Evaluation metrics

Developing metrics to evaluate machine learning models is essential for researchers and practitioners to pinpoint the most suitable and efficient models for particular tasks. To prevent the model from being evaluated on data encountered during training, the dataset is divided into training and testing sets. The training set fits the model, allowing it to learn the underlying patterns and relationships in the data. In contrast, the testing set, which the model has not seen during training, is used to assess its predictive performance and generalization capability. This separation helps to detect overfitting, where the model performs well on the training data but poorly on new, unseen data, ensuring that the model's performance is robust and reliable across different datasets.

Dividing data into training and testing sets may result in bias if these subsets fail to represent

the total population adequately. Bias occurs when the division leads to an overrepresentation or underrepresentation of certain patterns, features, or categories in one of the subsets, potentially causing the model to exhibit high performance on the training set but poor performance on the test set, or the reverse. Moreover, continuously refining the model to enhance its performance on a particular test dataset can impair its generalization capabilities. Such circumstances can lead to an inaccurate assessment of the model's performance and capacity to generalize to novel, unseen data.

To avoid this limitation, cross-validation is important for measuring the performance of classification tasks because it helps ensure that the model is generalizable and not overfitting to a particular subset of the data. By partitioning the data into multiple subsets and using these subsets iteratively as training and testing sets, k-fold cross-validation provides a more robust estimate of the model's performance on unseen data. This process helps identify the model's ability to generalize predictions across different data samples, leading to more reliable and valid performance metrics. However, cross-validation requires the whole dataset to be labeled, which is not always possible when manually labeling only a dataset sample.

The performance of an algorithm is typically presented using a confusion matrix. A confusion matrix is a table used in classification analysis to visualize the performance of an algorithm by showing the true and false positives and negatives for each class. It shows the number of correct and incorrect predictions made by the model compared to the actual classifications. The matrix is typically a 2x2 grid for binary classification (an example is shown in Table 4.1), with rows representing the actual classes and columns representing the predicted classes. The matrix's main diagonal shows the number of true positives and true negatives, i.e., the instances correctly identified by the model. At the same time, the off-diagonal elements represent the false positives and false negatives, i.e., the instances incorrectly identified.

Table 4.1: Confusion Matrix for Binary Classification.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Metrics were introduced to make comparing models easier than comparing confusion matrices. The model's performances are typically compared through accuracy, precision, recall, specificity, and F1 score. The definitions of these metrics are given below:

1. **Accuracy:** It measures the proportion of correct predictions (both true positives and true negatives) among the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{4.1}$$

2. **Precision:** Also known as positive predictive value, it measures the proportion of true positive predictions in the total predicted positives.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.2}$$

3. **Recall (Sensitivity or True Positive Rate):** It measures the proportion of actual positives that are correctly identified.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.3}$$

4. **Specificity (True Negative Rate):** It measures the proportion of actual negatives that are correctly identified.

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{4.4}$$

5. **F1 Score:** The F1 score combines precision and recall into a single metric by taking their harmonic mean, offering a balanced measure of a model's performance, especially in uneven class distributions.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.5}$$

6. **Receiver Operating Characteristic (ROC) Curve:** A graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It is created by plotting the true positive rate (Recall) against the false positive rate (1 - Specificity) at various threshold settings.

7. **Area Under the ROC Curve (AUC-ROC):** A single scalar value summarizing the performance of a classifier across all thresholds. The AUC represents the probability that a random positive example is ranked higher than a random negative example. A value of 1 represents a perfect classifier, while 0.5 represents a worthless classifier.

In classification tasks, the key evaluation metrics include accuracy, which measures the overall correctness of predictions; precision and recall, which assess the model's ability to identify positive cases correctly and its performance on actually positive cases, respectively; F1 score, which combines precision and recall into a single metric for balance; and the area under the ROC curve (AUC-ROC), which evaluates the model's ability to distinguish between classes at various threshold levels. These metrics provide a comprehensive view of model performance, providing the basis for comparing models.

The following section presents the main natural language processing techniques that will be

used for log classification.

4.2.2 Analyzing textual data

Techniques such as TF-IDF and Random Forest exemplify the advanced methodologies used to analyze and derive meaningful insights from textual data sets across various domains.

The TF-IDF (Term Frequency-Inverse Document Frequency) method is a statistical measure used to evaluate the importance of a word in a document relative to a collection or corpus of documents. It is based on two key concepts: Term Frequency (TF), which measures how frequently a term appears in a document, and Inverse Document Frequency (IDF), which assesses the rarity of a term across all documents in the corpus. The TF component emphasizes words that are frequent in a specific document, while the IDF component downweights words that are common across many documents, thereby highlighting words that are unique to particular documents. The combination of these two measures, TF-IDF, results in a weight for each word in each document, with higher weights assigned to terms that are important or characteristic of the document, thus enabling the identification of relevant terms and the comparison of documents within the corpus for tasks such as search, information retrieval, and document clustering.

Random Forest is a machine learning algorithm that operates by constructing many decision trees at training time and outputting the predicted class by taking the mode of the classes (i.e., the class with the most estimators) for classification tasks or the mean prediction for regression tasks. It is an ensemble method that combines the predictions of several base estimators built with a given learning algorithm to improve generalizability and robustness over a single estimator. In the case of Random Forest, the base estimators are decision trees. Random Forest introduces randomness when building each tree; it randomly selects a subset of features at each split in the training process, which helps make the model more diverse and hence reduces the risk of overfitting the training data. This approach allows Random Forest to provide high accuracy in various applications, including classification and regression tasks.

The following section explores the classification of CI/CD logs using natural language processing techniques.

4.3 Natural language processing

4.3.1 Method

We propose to categorize build failures based on natural language processing methods. As stated, the prediction task is binary: we aim to predict whether the root cause is deterministic

or non-deterministic (i.e., *REPO* or *EXTERNAL*). This model is referred to as the *EXTERNALITY* model. The model name illustrates the objective of the model: estimate the extent to which the root cause originates outside the code repository. The prediction model uses TF-IDF to represent the build log and Random Forest to predict the root cause category. TF-IDF is part of *Bag of Words* (BoW) approaches, meaning the order of the words inside the document is lost. As discussed in Section 1.4, log messages are the narrative threads that weave together the story of a CI/CD build process. Given this, the utilization of TF-IDF might initially seem counterintuitive. Nonetheless, TF-IDF remains a straightforward yet frequently effective method. The presence or absence of specific words within build logs can aid in their classification. Consequently, we postulate that employing TF-IDF for log analysis can yield robust classification outcomes. In the following section, we benchmark the methods on an industrial dataset.

4.3.2 Evaluation

We benchmark the algorithm on a dataset to evaluate the effectiveness of TF-IDF and Random Forest in classifying build logs with *EXTERNALITY* model. We use the industrial dataset defined in Section 3.2. To ensure relevant classification metrics, we construct the dataset to be balanced between the two classes (*REPO*, *EXTERNAL*). Therefore, both classes are equally represented. As the original dataset contained more logs with *REPO*, we used sub-sampling to create a balanced dataset. We used k-fold cross-validation with five splits ($k = 5$) to compute performance metrics. The vocabulary size is equal to 5000.

The model should be retrained regularly to allow the model to learn new failure patterns over time. In Amadeus, we implemented a weekly retraining job. Results presented here are the ones of the model named *2024-01-27T01:17:33.170189*. Table 4.2 presents precision, recall, and F1-score metrics. Figure 4.1 presents the confusion matrix. The approach enables the classification of deterministic versus non-deterministic failures with an F1-score of 0.81 for the *EXTERNAL* class and 0.76 for the *REPO* class, significantly outperforming a random classifier as the dataset is balanced.

	Precision	Recall	F1-Score	Support
EXTERNAL	0.80	0.82	0.81	7565
REPO	0.78	0.74	0.76	6234
Accuracy			0.79	13799
Macro avg	0.79	0.78	0.78	13799
Weighted avg	0.79	0.79	0.79	13799

Table 4.2: Classification report for EXTERNALITY model.

We measure the impact of varying the hyperparameters on the F1-score. We vary vocabulary size, number of estimators, max depth of estimators, minimum document frequency, and maximum document frequency. We did not observe significant performance variation. Ta-

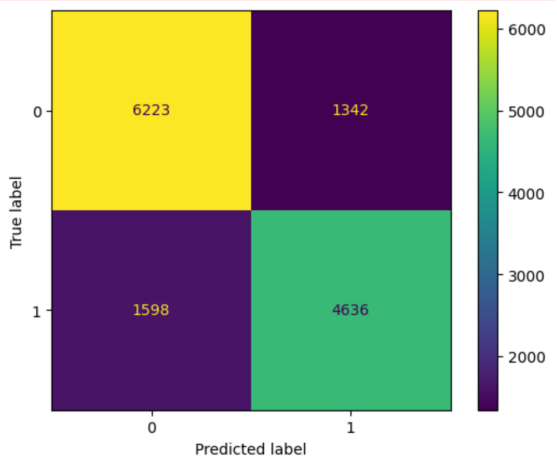

Figure 4.1: Confusion matrix for EXTERNALITY model. There are four values: True Positive (TP) and True Negative (TN) on the diagonal, and False Positive (FP) and False Negative (FN) on the anti-diagonal.

ble 4.3 presents the F1-score when minimum and maximum document frequencies are varied. The F1-Score is not significantly impacted. Results were surprising as we expected a minimum document frequency to help ignore build-specific tokens, such as random identifiers. Similarly, we expect a max document frequency to help ignore tokens that appear in all build logs (both successful and failed logs), such as logging made by the orchestrator.

Table 4.3 presents the F1-score when varying vocabulary size and filtering of log lines. We tried to keep only error lines instead of all lines to reduce the vocabulary size and noise from non-errored lines. Error lines refers to lines with any of the terms `error`, `fatal`, `exception`, `fail`, `abort`, `problem`, or `issue` (case-insensitive). We observe this has a minimal impact on the F1 score.

We manually investigated incorrectly classified logs. We discovered that the following patterns tend to lead to wrong predictions. First, some root cause messages appear identically in both classes, i.e., `EXTERNAL` and `REPO`. For instance, the root cause message `Docker login failed` appears in both classes. This is surprising because this root cause seems to be related to bad user credentials. By leveraging monitoring infrastructure, we observed it appeared when the docker registry was under heavy load. This is because the server-side Docker login workflow has a timeout for checking the validity of credentials. If the time-out is reached, it returns a bad login error. This is a questionable choice as the error message suggests the issue is on the client side, and the HTTP response code is 403. A server-related failure, such as `Internal Server Error`, would be more adapted.

Second, CI/CD workflows can transparently inhibit the impact of non-zero exit codes. For instance, `command -arg || true` will not return a non-zero exit code, even if the command

Minimal DF	Maximal DF	Avg. F1-Score
0.0	0.8	0.753
0.0	0.9	0.758
0.0	1.0	0.758
0.1	0.8	0.757
0.1	0.9	0.757
0.1	1.0	0.761
0.2	0.8	0.753
0.2	0.9	0.756
0.2	1.0	0.758

Table 4.3: Impact on the F1-score when varying minimal document frequency between 0.0 and 0.2, and maximal document frequency between 0.8 and 1.0. All other hyperparameters remain identical (vocabulary size equal to 1000). We observe that it does not significantly impact the F1 score.

Vocabulary size	Log lines	Avg. F1-Score
1000	all	0.758
5000	all	0.757
1000	error lines only	0.747
5000	error lines only	0.748

Table 4.4: Impact on the F1-score when varying vocabulary size and log lines. Filtering tokens based on document frequency was not used. We observe that it does not significantly impact the F1 score.

returns a non-zero exit code. However, the `stdout` and `stderr` will still be logged into the build log. Therefore, we observed in the build log error messages that are usually considered fatal (as they come from a command returning a non-zero exit code). Still, in the context of this build, they will not impact the build outcome. Furthermore, while some CI/CD systems, such as GitHub Actions, automatically log the shell code executed in the pipeline, this is not the case for Jenkins orchestrator. As such, there is no trivial way to detect whether the exit code is inhibited or not from the build logs. This can be detected by analyzing the workflow definition file. However, it implies relying on other data sources rather than solely on build logs.

Employing a Natural Language Processing (NLP) method for classifying build logs has yielded encouraging outcomes. Nevertheless, build logs represent merely a fraction of the data accessible from CI/CD builds, including elements like build triggers and workflow steps. Within this context, leveraging Knowledge Graphs could prove to be more effective.

4.4 Knowledge Graphs

Knowledge graphs are data structures that represent information in a network of interconnected entities and their relationships, often visualized as nodes (entities) and edges (relationships). They integrate, manage, and retrieve structured and unstructured data, enabling more effective data analysis, linking, and semantic querying. Knowledge graph embeddings transform the entities and relationships in a knowledge graph into continuous vector spaces. This representation facilitates machine learning tasks by enabling algorithms to process and analyze the relationships and patterns within the graph more efficiently. Embeddings capture the semantic relationships between entities in a low-dimensional space, making it easier to perform tasks like link prediction, entity resolution, and recommendation. We described knowledge graphs in Section 2.3.4. Furthermore, we investigated their use for log analysis (Section 2.3.4) and root cause analysis (Section 2.3.4).

This work focuses on the binary classification of builds between REPO and EXTERNAL classes, as described in Section 4.3. We hypothesize that Knowledge Graphs are best suited for representing builds in the continuous integration process, considering their ability to easily integrate heterogeneous information, such as series of events of variable length and categorical data typical of the traces produced by CI systems. A knowledge graph allows the decomposition of the process into elements (e.g., a build machine, a build trigger) represented as nodes in the graph. We postulate that processing the graph will allow the identification of failure patterns that would be much more difficult to detect independently. The next section defines how to organize CI/CD builds knowledge using an ontology.

4.4.1 Ontology

An ontology is a structured framework that categorizes and defines the fundamental types, properties, and relationships of entities within a particular knowledge domain. It is used to model the nature of reality in that domain, allowing for the consistent representation and shared understanding of concepts. Ontologies are often used in artificial intelligence, semantic web technologies, and information science to enable data interoperability, integration, and knowledge management by providing a common vocabulary and set of assumptions about how to interpret the data within a specific context.

We design an ontology to represent builds in continuous integration systems. The ontology is available at https://purl.org/cibuilds. The ontology contains 14 classes, 13 object properties, and 10 data properties. Classes and properties are defined below:

- **Classes:** Build, BuildMachine, BuildStage, BuildTrigger, CodeRepo, Command, ConfigUpdate, Job, RootCause, SourceCodeUpdate, SourceFile, TestCodeUpdate, Timer, User.
- **Object Properties:** fatalStage, hasCodeRepo, hasCommand, hasFile, hasRootCause, hasStage, isExecutionOf, nextStage, previousStage, runsOn, triggeredBy.
- **Data Properties:** brokenSince, hasArgument, hasBuildDate, hasBuildUrl, hasExitCode, hasRelativePath, hasTimeToFix, isSuccess, isUsingRemoteServices, isInfrastructureInstabilities.

Figure 4.2 illustrates the representation of a CI/CD build in the knowledge graph. The represented build has two stages (Compile and Upload). It was triggered because of a code change in a source file (*src/main.py*), and it has failed because the stage Upload has failed. After defining an ontology to organize builds knowledge, the next section presents how to build a graph representation of a CI/CD build.

4.4.2 Building build graphs

An ontology defines how the structure of the knowledge graph, ensuring consistency when representing CI/CD builds. We experimented with Knowledge Graphs on industrial CI/CD builds, as for the NLP approach detailed in Section 4.3. To populate the knowledge graph with real-world data, we leverage the REST API of the CI/CD orchestrator. The API provides information such as why the build was triggered, on which build machine it ran, and the different steps of the build. We leverage the Jenkins Scanner defined in Section 3.2.1 to scrape CI/CD builds. The scraper generates the representation of builds in Turtle format and stores them in text files (one file per build).

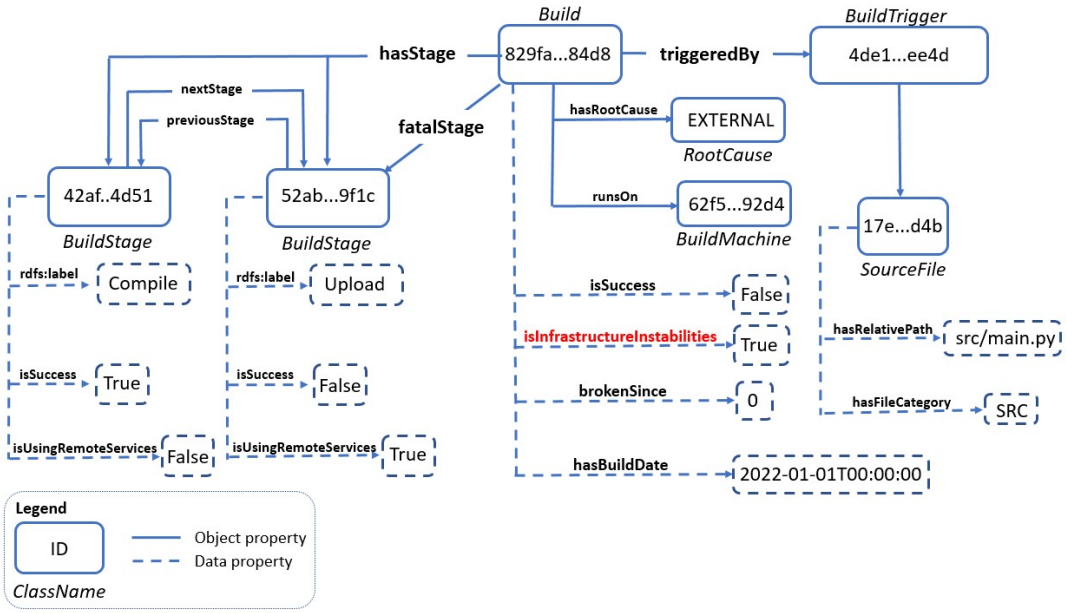

Figure 4.2: Excerpt of the Knowledge Graph representing a specific Build with its stages and states. The red arrow corresponds to the edge we expect to predict.

CI/CD orchestrator API provides detailed information about the build itself but lacks information about the environment. In the next section, we explore how to enrich the graph with simple heuristics or by extracting and linking data from other systems.

4.4.3 Enriching graphs using data augmentation

Orchestrator API provides information about the build. However, there is much more information available: insights from the build log, the state of the environment (e.g., known outages), analyzing network connections between the build and its environment (e.g., API calls, dependencies download), or other metrics created by other systems (e.g., infrastructure load, command metrics, failure analyzers). It is also possible to enrich available data using simple heuristics. This information heterogeneity from multiple systems encourages a data integration and enrichment approach, best conducted using knowledge graphs.

Using a heuristic, we generate the value of the data property `isUsingRemoteServices`. If the `rdfs:label` value of the build stage (i.e., the stage name) contains at least one of the words *publish, deploy, download, push, remote, upload, checkout* (case-insensitive), then we consider that this stage is using remote services. Most builds have stages that rely on external dependencies to work properly: download dependencies from a remote server, use web APIs, push compilation artifacts to remote storage, etc. Due to infrastructure instabilities, broken builds often result from failure to use remote services. Therefore, it is relevant to document

whether a stage uses remote services.

When a build is triggered because of a code change, the graph comprises which files were modified by the commit that triggered the build. An example is illustrated in Figure 4.2, where *src/main.py* was modified. However, the CI/CD orchestrator provides relative paths of modified files but does not provide information on the category of files. For instance, identify whether it is a source, test, or config file. We postulate the category of modified files can impact the build outcome and, as such, are relevant for root cause analysis. Therefore, we augment the graph using a heuristic to generate the *hasFileCategory* data property. The category is identified using the following methodology: checking file path against these patterns: if a path matches the test pattern, it's categorized as "TESTS"; if it matches the config pattern, it's "CONFIG"; otherwise, it's categorized as "SRC." Patterns are regular expressions for the Python *re* module. Test pattern is *.*test(s)?.** . Config pattern is files with extensions *cnf,conf,cfg,cf,ini,xml,toml,yaml,yml,json,dat,properties,hocon,txt,md,lock,sh,bash*, files named *config,conf,jenkinsfile*, and dotfiles (i.e., files started with a dot). This method allows the graph to be augmented with file categories automatically.

Network traffic analysis can help to precisely identify the remote resources in use as well as transport failure (e.g., TCP timeout) or application failure (e.g., HTTP status code >= 400). However, nowadays, most of the traffic is encrypted using TLS, thus preventing passively monitoring HTTP requests. We discuss possible solutions in Chapter 5.

Integrating service status is also interesting to model in the graph whether remote services are available during the build execution. While a broken remote service can lead to build failure, services can have a partial outage: they fail to process only a small part of total requests. Therefore, service status can be green even when the root cause is related to this service. Therefore, we did not integrate service status into the graph.

Given that knowledge graphs represent builds, the next section will focus on classifying graphs to provide the classification of build failures.

4.4.4 Predicting links

We propose to use Knowledge Graph Embedding (KGE) methods on the large and heterogeneous graph about CI builds presented in Section 4.4.1. KGE aims to represent entities as vectors that can be compared with a distance function and similarity metrics. In our experiments, we tried TransE [16], DistMult [189], HolE [126] and TorusE [51]. TransE, as conceptualized by Bordes et al., posits a geometric interpretation of relational data, representing entities and relations in a shared embedding space where relations translate entities close to each other if they are interconnected. DistMult, introduced by Yang et al., simplifies relational modeling through a bilinear approach, utilizing diagonal matrices to encapsulate symmetric

interactions between entities, thus facilitating an efficient and interpretable representation of relationships. HolE, proposed by Nickel et al., amalgamates the dimensionality reduction of canonical correlation analysis with the rich representational capacity of tensor factorization, employing circular correlation to generate composite representations of entity pairs. Lastly, TorusE, formulated by Ebisu and Ichise, advances the geometric paradigm by mapping embeddings onto a toroidal space, thereby addressing the cyclical nature of some relations and enhancing the model's ability to capture recurrent relational patterns.

We did not use literal values to create the embeddings. For instance, TransE models relations as a translation from head to tail entities.

$$\mathbf{e}_h + \mathbf{e}_r \approx \mathbf{e}_t$$

Reworking the expression above and applying the p-norm leads to the TransE interaction function:

$$f(h,r,t) = -\|\mathbf{e}_h + \mathbf{e}_r - \mathbf{e}_t\|_p$$

The objective of the learning phase is to maximize the TransE interaction function. When the algorithm is trained, similar entities should have similar vectors. In the context of this work, the head entity is a `Build`, the relation is `isInfrastructureInstabilities`, and the tail entity is a `Boolean`.

This classification problem is equivalent to a link prediction problem for the `isInfrastructureInstabilities` edge (in red in Figure 4.2) between a Build object and a boolean value. The `isInfrastructureInstabilities` property is a binary representation of `hasRootCause` property: this is equal to true if `hasRootCause` is connected to `EXTERNAL` root clause node, or equal to false if connected to another node. It aims to simplify (reduce from multi-class to binary) the prediction task. In the following section, we evaluate the performance of the approach.

4.4.5 Evaluation

We use a dataset of 829,638 triples representing 8,586 builds where we know the real root cause (*EXTERNAL* or *REPO*). This is an average of 96.6 triples per build. We labeled them following the methodology described in Section 3.2.2. The dataset was split into training (90%)

Embedding method	Embedding dimension	balanced accuracy score	F1 score
TransE	50	0.875	0.75
TransE	100	0.898	0.795
TransE	300	0.839	0.679
DistMult	50	0.799	0.597
DistMult	100	0.895	0.79
DistMult	300	0.875	0.75
HolE	50	0.615	0.231
HolE	100	0.867	0.733
HolE	300	0.772	0.545
TorusE	50	0.837	0.674
TorusE	100	0.908	0.816
TorusE	**300**	**0.948**	**0.897**

Table 4.5: Accuracy and F1 scores for the prediction of the failed build due to instabilities while varying the embedding dimension and embedding algorithms. The highest F1-score is achieved for TorusE with an embedding dimension of 300.

and testing (10%) sets. For the embedding generation, we used the PyKeen framework. Our code is available at https://purl.org/cibuilds/code-v1 to ease reproducibility.

Table 4.5 presents the evaluation results. We used an artificially balanced dataset to validate the performance, where we varied the embedding sizes and embedding methods. We limit the hyperparameter optimization to embedding size only to reduce search space. We used PyTorch's ExponentialLR learning rate (PyKeen's default). Low dimensional spaces result in poorer classification accuracy. However, with an embedding dimension of 50, we outperform the NLP approach presented in Section 4.3 for three over four methods with a more straightforward and easier-to-explain algorithm. The highest classification accuracy (0.948) and F1-score (0.897) are achieved for the TorusE embedding method with an embedding dimension of 300. The next section explores how to fix non-deterministic failures automatically.

4.5 Leveraging failure classification for automated fix

In the context of CI/CD, non-deterministic build failures refer to errors or failures that occur unpredictably and inconsistently, often due to factors like timing issues, resource contention, or external dependencies. These failures are inconsistent across multiple runs of the same build, meaning the build could fail once and then succeed later without any changes in the code or configuration. Restarting the build can fix these issues because it allows the build to run under slightly different conditions, potentially avoiding the transient problem that caused the failure in the first place.

There is a tradeoff between (i) not restarting any failed build (ii) restarting all failed builds. The

first approach requires operators to troubleshoot any broken builds, which takes the operator's time, while the last approach increases the load on the CI/CD infrastructure. Classifying builds between deterministic and non-deterministic failures allows to restart only non-deterministic failures.

We implemented a tool, referred to as *Jenkins autofix*, that works as follows:

1. For a failed build, ensure the build can be restarted
2. Predict if the failure is deterministic or non-deterministic
3. If the failure is non-deterministic, restart the build, else do nothing.

The tool was enabled only for a small part of the CI/CD builds (roughly 0.1%). Furthermore, some builds are not idempotent and, as a result, should not be restarted, e.g., releasing builds. Therefore, we require teams managing CI/CD pipelines to opt in and follow a procedure to enable the tool on their pipelines (e.g., giving appropriate permissions to a robotic account).

For the period of 1st to 30th March 2024, for CI/CD jobs that opted in the tool, 410 builds were predicted as non-deterministic failures and, as such, were automatically restarted. For automatically triggered builds, 176 (43.78%) were successful, otherwise, 226 (56.22%) were failed. In other terms, 43.78% of non-deterministic failures were fixed by the tool. Considering a troubleshooting time of 10 minutes, the tool saved 37.6 hours of work during the period.

We manually investigated why restarted builds are still failing. We identified the following reasons. First, this is because of a wrong prediction about the root cause of the failure: the algorithm predicted a non-deterministic failure, but it was a deterministic failure. As such, the restarted build will fail. Second, the root cause of the failure was temporary and, as such, limited in time. However, when the build was restarted (i.e., immediately after the failure), the temporary outage was finished. Common examples include the outage of a remote service (e.g., artifact storage system, source code management system).

In this section, we discussed the automated identification of non-deterministic failures. The next section presents another approach that leverages an existing regular expressions database for finer-grained classification.

4.6 Findings closest existing failure labels

The Build Failure Analyzer plugin for Jenkins is a tool designed to assist in diagnosing and resolving build failures in the Jenkins continuous integration (CI) system. When a build fails, this plugin analyzes the build log for known failure causes, which are defined by patterns in the

log output. Administrators can configure the plugin with rules that describe common failure scenarios and their solutions, allowing it to identify these issues when they occur automatically. Once a known failure cause is detected, the plugin can display detailed information about the issue, including potential fixes, directly in the Jenkins interface. This helps developers and operations teams to quickly understand why a build failed and how to resolve the issue, thereby reducing the time and effort spent on troubleshooting and increasing overall productivity and efficiency in the CI process.

However, we measured in the industrial context that only 5% of failed builds are labeled thanks to this plugin. Indeed, despite some effort to populate the base of regular expression, only a small part of failed builds are identified. Furthermore, labels tend to be generic. For instance, a failure label is `Failed to clone git repository`. It does not indicate the root cause, such as server timeout, bad credentials, or merge conflicts (the three examples presented here are based on real logs flagged with the label above). Therefore, despite its weaknesses, this plugin provides a database of regular expressions and root cause labels, although generic. However, regular expressions are on-off matching: they do not provide a `similarity score` to assess whether a root cause is close to specific labels. For instance, consider the regular expression `.*java.io.FileNotFoundException.*` developed to identify issues related to missing files for Java programs. The expression will match the log message emitted from a Java program, e.g., `java.io.FileNotFoundException: example.txt (No such file or directory)`. However, the expression will not match the log message from a Python program, e.g., `FileNotFoundError: [Errno 2] No such file or directory: 'example.txt'`. However, both log messages, emitted from Java and Python programs, refer to a similar underlying issue: a missing file. Therefore, we postulate a similarity score can be relevant. When no regular expression matches for a given log, the closest label will be presented to the developer instead of lacking any hint about the root cause (i.e., no match message). We define the closest label as the most helpful to identify the failure's root cause. This can be extended to identifying a list of closest labels instead of one label.

We built a dataset of failed builds where exactly one failure pattern has matched. As such, we excluded build logs without matching (95%) or build logs matched by more than one regular expression (< 0.1%). This dataset will be used for training. We reused the approach based on Natural Language Processing techniques, i.e., TF-IDF and Random Forest, defined in Section 4.3 to classify build logs. The results are shown in Table 4.6. Average accuracy score refers to the average of accuracy scores using a k-fold cross-validation with $k = 5$. All settings have a high classification accuracy score (> 98%). Evaluation with error lines only refers to the same methodology as Section 4.3.2. This indicates that the algorithm based on TF-IDF and Random Forest successfully recognizes failure patterns, as the regular expressions do. However, this does not imply the approach is helpful at this stage. Indeed, classifying failure patterns similarly to what regular expressions do is not helpful for the end user: directly using

Vocabulary size	Log lines	Avg. accuracy score
1000	full log	0.989
5000	full log	0.989
1000	error lines	0.992
5000	error lines	0.991

Table 4.6: Impact on the Accuracy when varying vocabulary size and log lines. We observe that the accuracy score remains high for all settings.

regular expressions will be easier and more accurate. We target to find the closest label when none of the regular expressions match.

However, evaluating the approach is challenging. Identifying the most helpful label for the failure's root cause, i.e., the closest label, is challenging and subjective. As such, cross-validation would be required between developers that annotate build logs. Setting up experiments involving developers can be challenging primarily due to the significant time commitment required from participants who are often busy professionals. Additionally, recruiting the right developers with the necessary skills and experience can be difficult, as these individuals are in high demand in the company. This scarcity makes it tough to find available participants who meet the specific criteria needed for the experiment. As such, we propose the evaluation of the approach as future work.

This approach has similarities to the approach presented in Chapter 6, where a downstream task that is not directly helpful for end users is used to train an algorithm. In the following section, we conclude this chapter.

4.7 Conclusion

In conclusion, this chapter has provided an in-depth examination of CI/CD build failure classification, highlighting the crucial distinction between deterministic and non-deterministic failures to enhance the CI/CD pipeline's reliability and efficiency. We explored a Natural Language Processing (NLP) technique, TF-IDF and Random Forest, to establish a foundational approach for categorizing build logs. We obtained an accuracy of 78%. Then, we explored the use of Knowledge Graphs. Integrating Knowledge Graphs and ontologies represented a significant advancement, offering a more effective method for comprehending and classifying build failures. We discussed possible strategies to enrich build representations with expert knowledge, such as using regular expressions on the steps' names. By encapsulating the intricate relationships within CI/CD processes into a graph structure, we achieved a 94% accuracy.

While the NLP method is more straightforward to implement and thus more accessible for

real-world applications, adopting Knowledge Graphs necessitates developing an ontology to organize build-related knowledge systematically. Data collection for NLP is limited to the collection of build logs. At the same time, the approach using Knowledge Graphs requires deeper integration with Application Programming Interfaces (APIs) provided by CI/CD systems, thus making the development of the solution harder. Knowledge Graphs, with their capacity to assimilate diverse data sources, open new opportunities in root cause analysis, enabling the incorporation of additional build data. For example, they allow for including insights from network traffic (Chapter 5). Precisely identifying the data sources that are most relevant for the classification task remains a future challenge, although the accuracy achieved so far opens opportunities for real-world deployments.

4.8 Summary

In this chapter, the examination focuses on two methodologies for segregating CI/CD failures into deterministic and non-deterministic groups. It was demonstrated that employing a straightforward yet effective NLP technique, TF-IDF combined with Random Forest enables the classification of failures with a 78% accuracy rate on a balanced test dataset. Since build logs constitute only a fraction of the available data on CI/CD builds, the utilization of knowledge graphs was investigated. An ontology designed for the Jenkins ecosystem was developed. Then, CI/CD builds were integrated into knowledge graphs. Subsequently, knowledge graph embeddings were employed to predict links between entities, effectively providing root cause analysis of build failures. This strategy attained a 94% accuracy rate, surpassing the approach's performance relying solely on build logs.

Chapter 5

Inspecting TLS traffic

This thesis explores approaches for automated root cause analysis of CI/CD failures. In Chapter 4, we present an approach based on Knowledge Graphs for classifying build failures. Knowledge Graphs ease the integration of heterogeneous data sources, such as build properties or insights from network traffic. Therefore, this chapter explores strategies for inspecting network traffic, especially encrypted traffic.

5.1 Introduction

Modern encryption protocols provide confidentiality and integrity guarantees essential for commercial and private communications over untrusted channels (e.g., the Internet). In 2022, 97% of pages loaded by Google Chrome in the USA were over HTTPS [71]. In addition to web traffic (HTTPS), many application protocols have adopted encryption, for example, emails (IMAP/SMTP) using STARTTLS or implicit TLS, VoIP, DNS over HTTPS, Virtual Private Networks (VPN), and QUIC. While some protocols use custom encryption, most rely on Transport Layer Security (TLS). Traffic encryption has become pervasive, greatly enhancing communications security; however, this also raises new challenges for network monitoring and intrusion detection. This creates, for example, a blind spot for Intrusion Detection Systems (IDS) between the internet and internal company systems [166]. Multiple solutions have been proposed to allow network analysis of encrypted traffic. However, they are often tricky to use in practice as they require target cooperation or break the TLS security model, which is often unacceptable. Therefore, network administrators must use context-specific solutions that are often costly to implement and maintain or even impossible to deploy at scale. In this work, we focus on the decryption of TLS sessions in a generic and non-invasive way to allow easy real-world usage, e.g., in corporate networks. We propose a generic mechanism for key extraction from the process memory to solve this problem. Our approach relies on eBPF, an in-kernel feature enabling the safe execution of privileged code, to identify processes that start

a TLS connection, taking a process memory snapshot before and after the TLS key exchange. We then recover secret keys from the difference between snapshots.

In this chapter, we make the following scientific contributions:

- We highlight that previous approaches to TLS traffic inspection are undesirable in many settings.
- We propose a generic approach to capture key material in memory with minimal intrusiveness, no static signatures, and low overhead.
- We implement X-Ray-TLS, a tool that reliably collects TLS key material on a Linux host that only relies on existing kernel facilities, allowing easy, large-scale deployment.
- We evaluate X-Ray-TLS on different setups: major TLS libraries, CLI tools, and a web browser.
- We identify several use cases where X-Ray-TLS helps to improve security (e.g., CI/CD pipelines)

The remainder of this chapter is organized as follows. Section 5.2 describes how TLS protocol works, synthesizing existing approaches and related work on TLS interception. Section 5.3 presents X-Ray-TLS' approach. In Section 5.4, we evaluate our approach on an extensive dataset of TLS clients and describe some of its limitations. In Section 5.5, we detail real-world use cases of X-Ray-TLS and compare existing interception solutions. Section 5.7 concludes on TLS interception approaches and describes potential improvements.

5.2 Background

Transport Layer Security (TLS) is a widely used client-server protocol that provides confidentiality, server authentication, integrity protection, and, optionally, client authentication. TLS is often used to secure communications over untrusted networks such as the Internet. TLS is used to provide secure HTTP (HTTPS), virtual private network security (e.g., OpenVPN), or industrial protocols (e.g., DNP3), among others. TLS 1.3 is also used to secure QUIC [92], a network transport protocol announced by Google in 2013, which is based on UDP. It provides layer 4 and layer 6 features as an all-in-one replacement for TCP and TLS. Therefore, it aims to provide faster session establishment by reducing required round-trips.

TLS is a transport security protocol that provides security guarantees only during transit. Therefore, the local security on each channel side (i.e., client and server) is outside its scope. Therefore, accessing encrypted content while having complete control of either client or server

is not considered a security vulnerability of TLS. TLS 1.0 was first defined in 1999 as an upgrade of SSLv3 (Secure Sockets Layer). Since then, TLS 1.1 (2006), TLS 1.2 (2008), and TLS 1.3 (2018) have been defined. As of 2022, only TLS 1.2 and TLS 1.3 should be used. Older versions are deprecated and not supported by major web browsers (e.g., Firefox, Chrome).

A TLS session starts with a handshake. The TLS handshake aims to agree on a shared secret between the client and the server over an untrusted channel. First, the client sends ① a ClientHello message advertising the supported TLS version, a random number (client random), and a list of supported cipher suites. The server answers ② with a ServerHello packet containing the chosen TLS version, a random number (server random), and the chosen cipher suite. The following packets are encrypted using handshake-specific keys agreed upon during the initial ClientHello/ServerHello exchange using the ECDHE algorithm. The client will verify that the server certificate is signed by a trusted (i.e., in client trust store) Certificate Authority. Furthermore, the certificate contains a Subject field. It represents server names protected by the TLS certificate (e.g., example.com, www.example.com, *.apps.examples.com). The certificate is valid only if the requested hostname matches the certificate's Subject. It is up to certificate authorities to ensure that a certificate requester owns all CNs matched by a certificate before issuing the certificate. Server certification validation ensures the client connects to a legitimate server for the requested hostname. Then, the client sends a final Finished message to indicate that the handshake is finished and that future traffic must be encrypted using application traffic secrets. In conclusion, TLS provides confidentiality, integrity, and server authentication guarantees in transit over an untrusted channel. Handshake for TLS 1.3 with only server-side authentication is depicted in Figure 5.1. Beyond the TLS 1.3 handshake we just described, various variants of TLS secure communication exist: TLS 1.2, session resumption, middlebox compatibility, or QUIC, which reuses TLS 1.3 message flow. In the following section, we describe the challenges when inspecting TLS traffic.

5.3 X-Ray-TLS' approach

X-Ray-TLS allows inspection of TLS sessions made by local programs. We applied our approach to TLS clients. No fundamental limitation prevents the application of the same approach on TLS server programs. However, in the context of the use cases detailed in Section 5.5.2, we commonly do not have access to the TLS server (e.g., remote server over the Internet). Therefore, we focused on extracting TLS session keys from TLS libraries used in client mode. For TLS 1.3, inspection targets only application traffic (i.e., not handshake traffic). Our approach works as follows. When a TLS handshake starts, the process initiating the connection is frozen (i.e., SIGSTOP), its memory is dumped, and the process is released (i.e., SIGCONT). We only dump writable memory regions as TLS session keys cannot be stored in read-only memory areas. We hypothesize that TLS clients are not re-mapping the memory region containing TLS

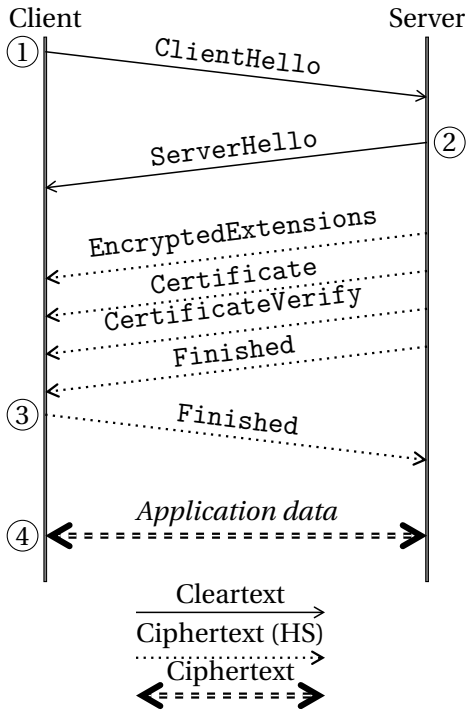

Figure 5.1: Illustration of TLS 1.3 handshake. TLS 1.3 requires only one round-trip for cold-start compared to TLS 1.2 which requires two round-trips.

secrets from read-write to read-only during the handshake. When experimenting on common TLS libraries (as described in Section 5.4.1), we did not identify a TLS library that remaps the memory region containing TLS keys during runtime, thus verifying the hypothesis. Dumping target process memory requires root privileges on the machine running the target TLS client program. While root privileges would allow instrumenting the target program (e.g., to retrieve plaintext using function hook), X-Ray-TLS allows to achieve the same result with significantly less setup effort. When the TLS handshake is completed (i.e., first `ApplicationData` TLS record), the source process is frozen, memory is dumped, and the process is released. The two memory dumps are used to generate a memory difference (i.e., content added between the first and second memory dumps), which is expected to contain the TLS secret keys. Looking for session keys in the memory difference instead of the full memory dump significantly reduces the search space (usually from GB to KB). We end up with several key candidates as various data is written between the beginning and end of the handshake, such as TLS certificates and internal state updates. Finally, we do an optimized exhaustive key search on key candidates.

We leverage eBPF to detect TLS connections and, therefore, can identify arbitrary TLS sessions without requiring intrusive instrumentation, like program instrumentation or loading a custom Linux kernel module. X-Ray-TLS requires eBPF features added in Linux kernel 4.4 (released in 2016), so it can be used on systems with kernel 4.4 and above. eBPF lets us map

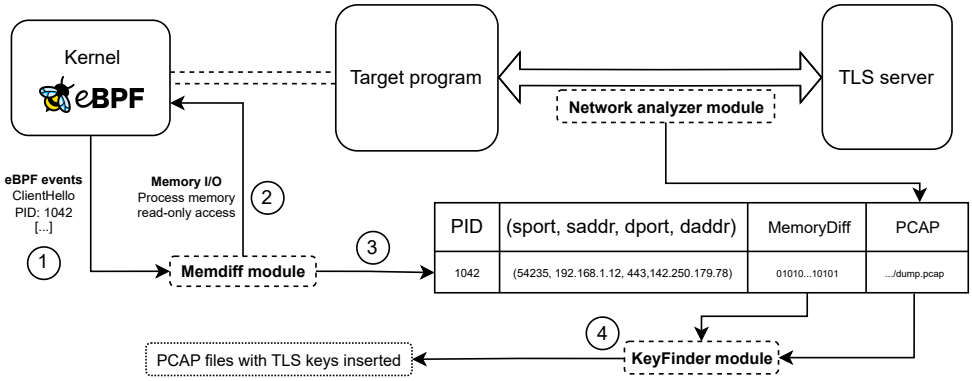

Figure 5.2: Overview of X-Ray-TLS architecture. `Memdiff` module computes the memory changes between the beginning and end of the handshake. `Network Analyzer` dumps network traffic to PCAP files. `KeyFinder` looks for secrets in the memory diff. Session decryption works as follows: (1) detection of the TLS session; (2) target process memory is dumped twice; (3) memory changes between dumps are saved; (4) key candidates are generated using heuristics, and then an exhaustive search is performed.

network sockets (IPs/ports tuple) with the PID. Tshark is Wireshark's command line interface (CLI). It implements dissectors for TLS (from TLS 1.0 to TLS 1.3), among many other protocols. We leverage tshark for network traffic monitoring (i.e., dumping traffic to PCAP files) and brute force key candidates without re-implementing the TLS stack. Finally, we leverage Docker to provide an all-in-one, easy-to-use tool that can be used in corporate and research environments.

We detail in Section 5.3.1 how we detect TLS sessions and map them to the source program. Then, we detail in Section 5.3.2 how to generate key candidates from memory snapshots. Finally, Section 5.3.3 details how to test key candidates and find session keys.

5.3.1 TLS Session Detection

To intercept TLS sessions, we first need to reliably detect when TLS sessions are started and identify the source program. To this end, we leverage eBPF to safely run custom code in a privileged context (e.g., kernel space). Therefore, our method does not require a kernel module or patch. We use BCC [138] to compile and load eBPF code and set up a shared ring buffer between eBPF code and a managing user-space program. Ring buffers allow triggering events in the user-space program from eBPF code efficiently. We detail the performance of event triggering in Section 5.4.3. We attach kernel probes [95] (kprobes) on two system calls: `tcp_v4_connect` for TCP sockets and `ip4_datagram_connect` for UDP sockets. Kprobes are breakpoints set on any kernel system call (syscall). Therefore, they allow running eBPF code when a new TCP or UDP socket opens. Kprobes are run before the execution of the syscall,

while kretprobes are run when the syscall returns. Kprobes allows retrieval of initial syscall arguments (before the system call might modify them). Kretprobes allow retrieving results of the syscall, e.g., socket remote peer information after socket initialization. We used kernel probes to create the mapping between sockets and the corresponding PID. Sockets are defined by the attributes of the connection: source IP, source port, destination IP, and destination port. Therefore, it allows us to map arbitrary packets on the network to the originating process.

We also leverage eBPF to attach a function in `SOCKET_FILTER` mode to a network interface. The interface should be the one used by target programs. Then, the function is executed for any packet transmitted over the network interface. The processing time of the function does not impact network latency as the function works on a copy of the original packet. We detect if the packet contains an Ethernet header (which is not mandatory, e.g., layer 3 tun interface) for each packet. Then, we parse TCP or UDP headers (i.e., for TLS over TCP or QUIC). Finally, when relevant packets are detected, we return events to user space over a ring buffer. Packet matching will be used to trigger memory snapshots. Therefore, we have to distinguish between initial and final snapshots. An initial snapshot should be triggered before session keys are stored in memory. For TLS, we identify `ClientHello` record by checking the first 6 bytes of the packet. For QUIC, we identify QUIC `Initial` record. The final snapshot should be triggered when session keys are stored in memory. For example, this condition is valid when the client sends packets encrypted with traffic secrets. Therefore, we detect the first `ApplicationData` packet from client to server as the final snapshot trigger. We store session states in a hash table to avoid triggering a memory snapshot for each `ApplicationData` packet.

In conclusion, the eBPF code allows us to map each network packet to its originating process and to trigger memory snapshots before and after session secrets are written to memory. In the following section, we will detail how we generate a list of key candidates from memory snapshots.

5.3.2 Key Candidates Generation

X-Ray-TLS extracts TLS session secrets from the target process memory. Unfortunately, there is no standard way to store such keys in memory. Therefore, the in-memory structure of the keys depends on the cryptography library used, and the memory location is execution-specific. A naive approach is to generate key candidates from memory content using a sliding window of secret length over memory. Finding naively 2 secrets in memory of size n bytes have a complexity of n^2: this is huge considering typical values of n ranging from MB to GB, i.e., 10^6 to 10^9. Therefore, the memory range should be as small as possible. To this end, simple optimizations can be used. First, the session keys should be in writable memory regions. Dumping only writable regions (i.e., region with `w` flag) significantly reduces dump size. Second, the keys are aligned with CPU word size (often 8 bytes). Therefore, only memory

Method	First event	Following events	Intrapage
full-full	full dump	full dump	yes
rst-partial	RST	partial dump	no
rst-partial-rst	RST	partial dump + RST	no
full-partial	full dump + RST	partial dump	yes
full-partial-rst	full dump + RST	partial dump + RST	yes

Table 5.1: Memory snapshot methodologies. Partial dumping consists of dumping only the pages with dirty flags. The notion of first event/following events is per PID.

addresses that can be divided by the CPU word size will be considered. This helps to divide by 8 the search space. Third, keys are high-entropy byte sequences. Entropy filtering excludes low-entropy parts of memory that are unlikely to be cryptographic keys. However, large programs tend to have a large memory footprint. They often open multiple concurrent TLS connections (which store in memory session keys, certificate chains, etc.) or load in memory content that can have high entropy (e.g., images, compressed archives). Therefore, entropy filtering performs poorly in this context.

In this work, we propose to use those techniques but also further reduce the search space by computing the difference between two memory snapshots. Figure 5.3 illustrates a TLS handshake and memory dumps. The target program starts by sending a ClientHello packet over the network. Note that for TLS over TCP, we do not consider the transport handshake (i.e., 3-way TCP handshake) as part of the TLS handshake. We define T_{SecGen} as the duration between detecting the ClientHello packet by network traffic analysis and when traffic secrets are stored in memory. Similarly, we define T_{stop} as the duration between the detection of the ClientHello and when the target process is frozen to be snapshot. We discuss in Section 5.3.2 different strategies for memory snapshotting. After secrets are stored in memory, the program continues processing and storing data (e.g., downloading a resource). Therefore, T_{noise} should be the lowest possible value. While a low T_{noise} is not required by design, it helps to reduce the number of key candidates and, ultimately, the key search time.

The secrets will be present in the difference between the two snapshots when the secrets do not appear in the first memory snapshot (*Initial dump*) but are present in the second memory snapshot (*Final dump*). This condition is validated if and only if $T_{stop} < T_{SecGen}$. We detail in Section 5.4.3 the validity of this hypothesis. We emphasize that T_{SecGen} is larger than the round-trip time of the network. Indeed, session secrets cannot be generated on the client side before receiving ServerHello (among others) records.

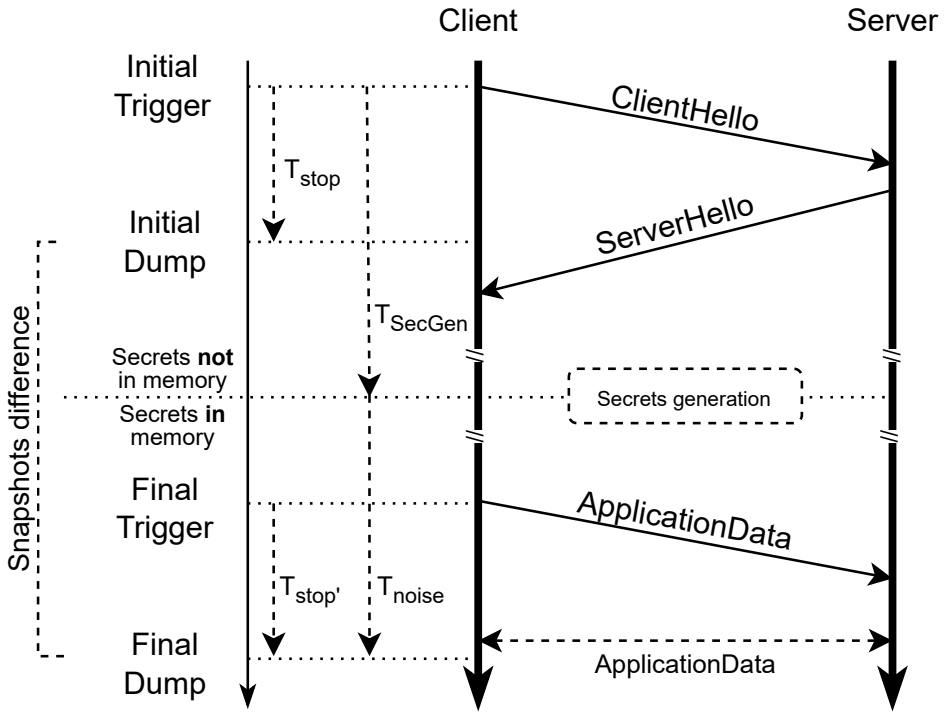

Figure 5.3: Memory dump cinematics in TLS handshake. T_{stop} represents the time to stop the process from a trigger event. T_{SecGen} represents the time to generate session secrets from the initial trigger. T_{noise} represents the time between when secrets are stored in memory and when the final snapshot is performed.

Memory Snapshot Methods

Extracting memory writes without a hypervisor is more challenging due to the lack of Virtual Machine Introspection (VMI) techniques. X-Ray-TLS dumps process memory by copying pages along with page addresses by reading pseudo-files `/proc/{pid}/pagemap` and `/proc/{pid}/mem`. For programs with large memory space or many TLS connections, doing full memory dumps twice per TLS handshake tends to slow down program execution. Therefore, we leverage a kernel feature to track memory changes. Memory tracking [54] was initially introduced in Linux kernel v3.9 (2013) to support the process checkpoint-restore project CRIU [53]. The memory tracking in the Linux kernel relies on the "Soft-dirty bits" mechanism. When the Soft-dirty bits mechanism is enabled, each memory page that is modified after a reference (reset) point will be marked as soft-dirty. This allows easier identification of memory pages modified from a reference point. Therefore, tracking memory changes requires two steps: (i) reset soft-dirty bits (ii) read the page map and check the soft-dirty bit for each page: if set, the respective page was written to since the last reset. Therefore, in case of multiple memory dumps, the following dumps will only dump modified pages since the initial dump.

This allows a significant reduction of the following dump sizes. Therefore, we developed 5 memory snapshot strategies that leverage the soft-dirty bit mechanism. Methods are presented in Table 5.1. The intra-page column refers to the ability to do intra-page differentiation. This is possible only when the first event comprises a full dump. In most cases, memory pages are 4kB in size. Then in the absence of intra-page differentiation, the memory difference can only be made with a 4kB increment (i.e., adding an entire page to the difference). Memory snapshot strategies are listed below:

- *Full-full* method takes 2 complete (i.e., all pages) process memory dumps. This method is the most straightforward, allowing fine-grained difference (i.e., intra-page). However, it has the most impact on the target program with a high freeze time.
- *Rst-partial* method resets dirty flags on the first event and then uses partial dumps (i.e., pages with soft-dirty flag only) without resetting dirty flags. However, this approach tends to accumulate pages to be dumped as soft-dirty flags are not reset.
- *Rst-partial-rst* is similar to rst-partial with the addition of resetting soft-dirty bits after every partial dump.
- *Full-partial* allows fine-grained differences at the expense of a full initial dump.
- *Full-partial-rst* allows intra-page difference while reducing freeze time by dumping soft-dirty pages only for events after the first one. Resetting soft-dirty bits after partial dumps helps to keep the number of pages to dump low. However, a reconstruction algorithm must be used to retrieve memory changes between two arbitrary dumps.

The dump strategy's benefits and limitations will depend on the target program, particularly the number of TLS sessions. For programs that do only one TLS session and exit, resetting soft-dirty flags after the final dump will have no impact as the program will exit. We detail in Section 5.4 the performance of memory dump strategies.

Figure 5.4 illustrates two overlapping handshakes. In the case of a dumping strategy that resets soft-dirty bits after dump (namely full-partial-rst and rst-partial-rst), special care should be taken in case of overlapping handshakes. Indeed, memory dump ② triggers a reset of soft-dirty bits. Therefore, memory dump ③ will not contain memory pages modified between memory dump ① and ②. If secrets of session 1 are written to memory between ① and ②, they will not appear in memory difference. This might happen depending on network RTT and the start time difference between session 1 and session 2. To avoid this problem, we develop a reconstruction algorithm to identify memory changes between two arbitrary snapshots (i.e., point in time) without requiring full dumps for all events. For the context of Figure 5.4, memory changes for session 1 are recovered as follows. Memory pages modified

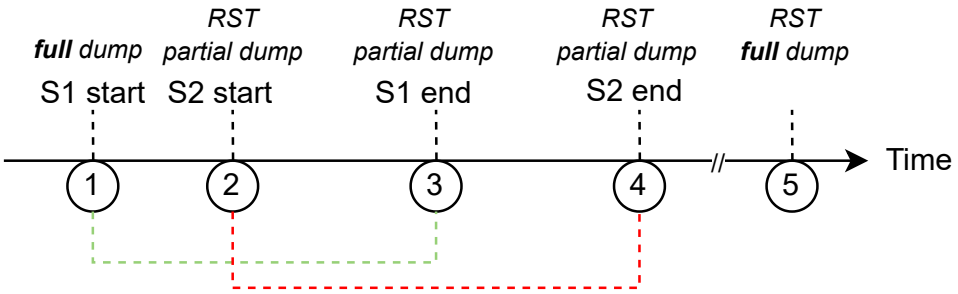

Figure 5.4: Illustration of changes reconstruction algorithm with the full-partial-rst method. Memory snapshots difference for session 1 (S1) is represented using a green dotted line. Memory snapshots difference for session 2 (S2) is represented using a red dotted line.

between ① and ② are present in partial dump ② (and similarly for ③). Memory changes are reconstructed by taking partial dump ② and overloading it with partial dump ③. Overloading a memory dump refers to replacing pages that appear in both dumps with pages of the most recent dump. Generally, changes between dumps i and j, where $j > i+1$, are reconstructed by starting at dump $i+1$ and then overloading every dump by the next dump ranging from $i+2$ to j (excluded). When the first dump for the session is not a full dump but a partial dump, we go backward until we find a full dump or a reset. To avoid having a continuously growing set of memory snapshots for long-lived programs, a full dump along with an RST is done when there is no in-flight TLS *handshakes* (as illustrated as ⑤). This algorithm allows the reconstruction of memory changes between two arbitrary points during process execution by only requiring soft-dirty flags kernel feature (i.e., without fine-grained memory tracking such as using a hypervisor). Memory changes between the beginning and the end of the handshake are not only composed of traffic secrets (that we are interested in). Indeed, memory changes (i.e., memory difference) are often an order of magnitude bigger than the secret size (e.g., kB compared to the secret size of 2×48 bytes). Therefore, we describe in the following section how we look for secrets in memory differences.

5.3.3 Exhaustive Key Search

Heuristics help to reduce the key search space significantly. However, depending on the target program, we still end up with potentially thousands of key candidates. We modified tshark to brute force TLS session keys to test key candidates efficiently. This task is managed by the `KeyFinder` module illustrated in Figure 5.2. Our modified version of tshark takes a path to a binary file containing key candidates (concatenated) and a path to a PCAP file containing encrypted traffic. Tshark will read packet by packet the PCAP file, and as soon as all required material is available (e.g., client random, server random, cipher suite), a brute force loop will

start. To check whether a key candidate is correct, we try to decrypt the packet and check if the HMAC [181] is correct. The brute force loop stops when all secrets are found: client-to-server and server-to-client traffic secrets for TLS 1.3 and master secret for TLS 1.2. Finally, results are printed, and tshark exits. Our patch to tshark is less than 200 lines and allows brute force TLS keys without re-implementing a TLS decryption stack, which is complex and challenging to implement correctly. This approach should also simplify the implementation of new TLS versions or new protocols in the future. The next section will evaluate X-Ray-TLS in different settings.

5.4 Evaluation

5.4.1 Benchmark

Experiments were done on a laptop with Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz with 16GB of RAM. The operating system was Ubuntu 20.04.5 LTS with Docker 20.10.21. Software versions (e.g., TLS library versions) are hardcoded in benchmark source files. In the following section, we present a benchmark of our tool on different programs and contexts. Then, we evaluate conditions for X-Ray-TLS to work.

Baseline

We define a baseline approach as generating key candidates from the memory contents using a sliding window of secret length over the memory. The number of key candidates is proportional to the size of the memory contents. Therefore, the memory extent should be as short as possible. To this end, we consider optimizations detailed in Section 5.3.2: dumping only writable regions, leveraging memory alignment, and excluding low-entropy areas of memory. The entropy threshold is determined using Figure 5.5. We will compare our method against this baseline.

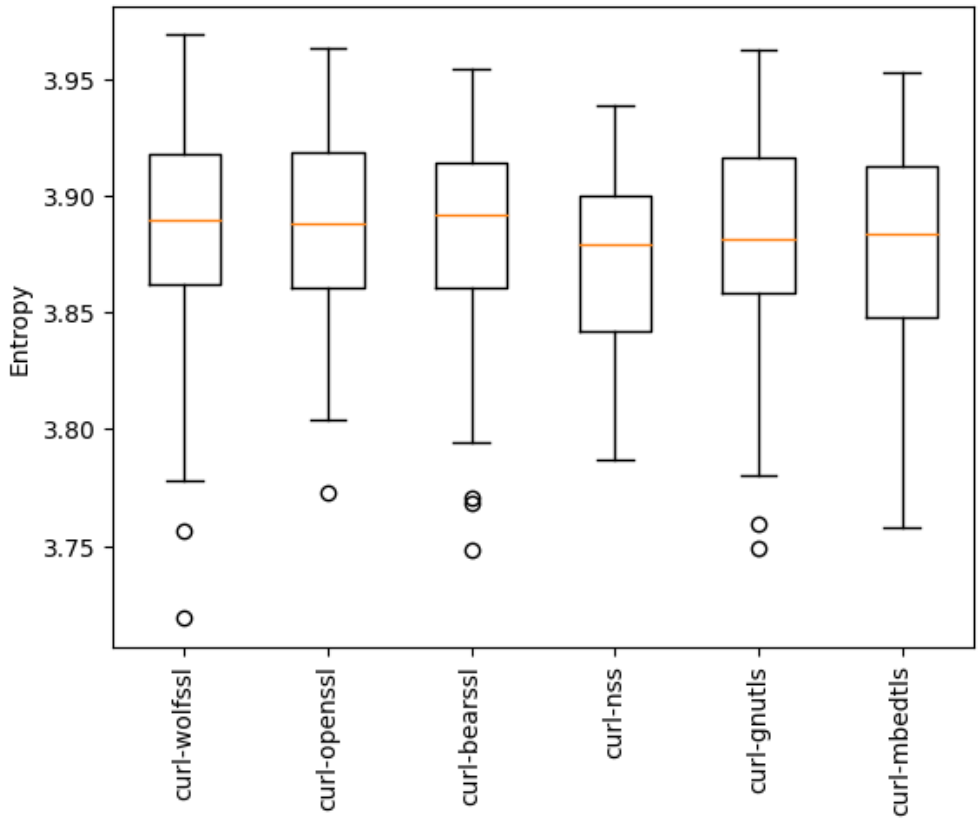

Figure 5.5: TLS secrets entropy per TLS library (computed on hexadecimal representation, thus maximum entropy is 4). We see a threshold (3.75) that can be used for entropy filtering.

TLS libraries

To ensure our approach works on major TLS libraries, we leverage curl support for different TLS backends. This allows us to easily test different TLS libraries with the same interface (i.e., same CLI). We built curl using the following TLS libraries [141]: OpenSSL, GnuTLS, NSS, WoflSSL, mBedTLS, and BearSSL. We have not found studies that estimate the usage of each TLS library. Based on discussions with experts and considering that software rarely implements by themselves the TLS protocol, we estimate that the 6 libraries we tested represent a significant proportion of the TLS libraries used by open-source and commercial software. We also compile curl with QUIC support using GnuTLS. For each program under test, we target 2 HTTPS URLs: a TLS 1.2-only server and a TLS 1.3-capable server. We choose TLS servers we do not control (i.e., on the internet) to highlight that X-Ray-TLS does not require any server collaboration. As X-Ray-TLS supports different memory snapshotting strategies, we benchmarked each strategy independently. Finally, we conclude with a test matrix of TLS library × memory snapshot strategy × URL. It shows that X-Ray-TLS can intercept TLS sessions from all major TLS libraries listed above for TLS 1.2 and TLS 1.3. We benchmarked X-Ray-TLS on CLI programs: wget,

OpenSSL s_client, Python (requests module), and pip. We obtained the same results as on curl: TLS secrets extraction is successful for all snapshot strategies. To assess if our approach works on QUIC, we manually check if QUIC secrets are in memory difference generated by X-Ray-TLS when curl using the GnuTLS backend connects to a QUIC server. We used a complicit QUIC server to dump QUIC secrets. We have successfully validated the presence of QUIC secrets in the memory difference, indicating that X-Ray-TLS also works on QUIC. Therefore, all memory snapshot strategies were successful in retrieving TLS secrets. However, they differ in terms of key extraction speed.

Large programs

We applied both X-Ray-TLS and baseline approach on Firefox, a large software with a large memory footprint. Our approach based on memory difference generates a search space of 13MB compared to 41MB using the baseline approach. This is a reduction by ≈70% of key search space, leading to reduced key search time and smaller memory consumption. Therefore, X-Ray-TLS performs significantly better than naive search on large programs.

Continuous Integration build server

Continuous Integration (CI) is a software development practice that promotes running a regular set of checks on code change (e.g., commit). In practice, a CI orchestrator (e.g., Github Actions, Gitlab CI, Travis CI, Drone) starts a new container that will execute a set of commands to ensure updated code still meets quality requirements. Common steps are to download dependencies, compile source code, and send artifacts to a remote registry. Continuous Integration systems are often used to build artifacts that will later be deployed to production systems. It is, therefore, a critical system for the security of the overall infrastructure. We discuss in Section 5.5.2 security considerations of CI/CD pipelines. CI builds are often managed directly by development teams, and many software stacks are used (e.g., code scanners, package managers, container image builders). Therefore, existing TLS interception methods such as Man-In-The-Middle, SSLKEYLOGFILE, or instrumentation are not usable at scale as they require deep modifications of CI builds, e.g., trusting a new CA for all programs running in the build container or are not easily applicable to a wide range of programs (e.g., instrumentation, SSLKEYLOGFILE). Furthermore, existing solutions in the literature require virtualization: this causes a slowdown of up to 5 times [42] compared to the native execution.

To show to which extent X-Ray-TLS can support continuous integration security, we used [104] to run CI builds locally. We focused on a Python-based project. The CI of the project installs development dependencies (i.e., code linter), builds a Docker image, and pushes it to a remote registry. All TLS connections made by the build container were successfully inspected. Furthermore, TLS inspection does not lead to a statistically significant increase in build

duration. X-Ray-TLS allows us to detect:

- Information disclosure: `pip` package manager adds various information in the HTTP User-Agent header (e.g., in requests to `files.pythonhosted.org`). The header value is a JSON containing the kernel and OpenSSL versions (including patch number), OS, and glib versions, among others. This allows the remote HTTP server to detect if outdated packages (i.e., that might be vulnerable) are in use.
- Build dependencies: we were able to identify and record all resources fetched over the network. This helps to support reproducible builds even if resources are not available in the future (e.g., resources on the Internet). Being able to reproduce builds helps to detect altered artifacts by a malicious CI server.
- Artifacts traceability: detecting uploads of artifacts helps to identify the source of artifacts in the artifact registry. This helps to ensure end-to-end traceability between CI servers, artifact registry, and production systems.
- Support root cause analysis: in case of CI build failure, identifying the root cause from the build log may be challenging (many different logging formats). Decrypting HTTPS traffic allows the recovery of error messages in the HTTP body and status codes directly by parsing HTTP packets. For instance, we could identify that `pip` was trying to fetch a non-existing package by checking for HTTP 404 responses.

Therefore, we show that X-Ray-TLS allows TLS traffic decryption of CI build containers in a simpler and easy-to-use approach than existing solutions. Decrypting TLS traffic helps to identify information disclosure, build dependencies, artifacts traceability, or support root cause analysis in case of build failure.

5.4.2 Memory strategies comparison

After demonstrating different real-world scenarios of X-Ray-TLS, we detail the performance impact of the different parameters, such as the memory strategy. To compare the performance of snapshot strategies, we target a Python-based program that opens 10 consecutive TLS connections. It stores 100kB of random data between TLS sessions in memory to simulate parallel data processing. Below, we describe the interception performance of the 10th TLS session made by the target program (even if all sessions were successfully intercepted). We repeated the experiment 10 times. For programs that open only one TLS connection and exits, dump strategies can be merged into 2 categories as shown in Figure 5.6. Figure 5.7 shows memory snapshots difference size for each memory strategy. Figure 5.8 shows the duration of each step of TLS interception. Memory strategies that reset soft-dirty flags have the lowest

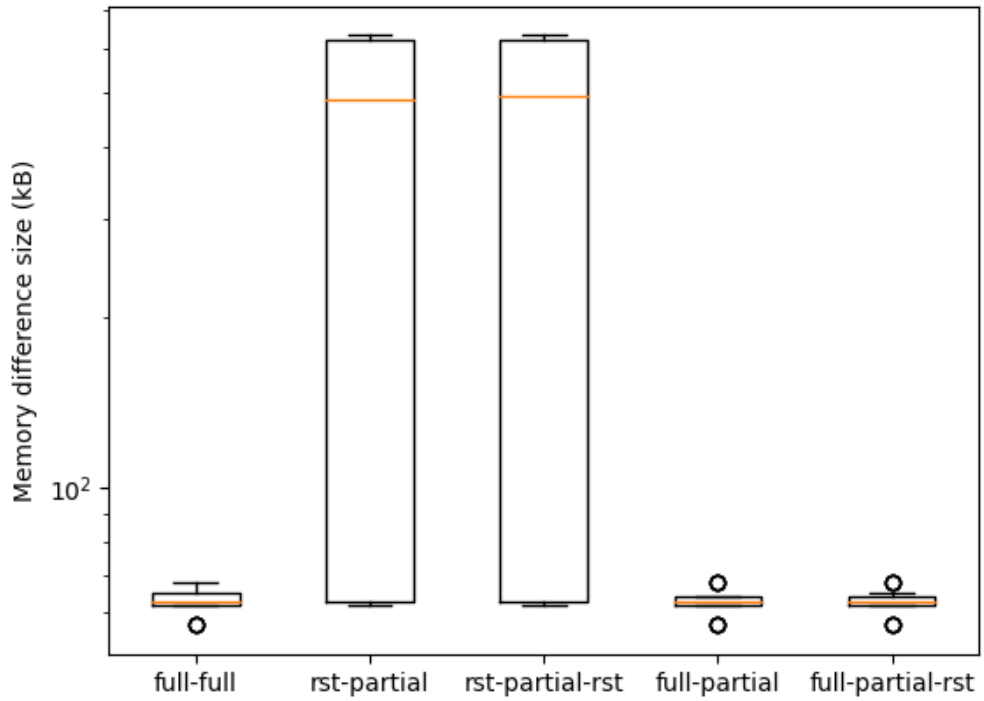

Figure 5.6: Illustration of memory difference size for each memory snapshot strategy for curl with 1 TLS session. Strategies can be divided into two groups: strategies with a full dump allow intra-page granularity, while strategies without a full dump are limited to page-by-page differences.

dump time because they only dump pages modified since the previous snapshot, compared to rst-partial or full-partial, which accumulate modified pages since the initial full dump. An entropy-filtering step is done in Python on the hex-representation of memory diff. Therefore, the performance of this step might be significantly improved with an optimized module (e.g., C binding). Figure 5.9 shows the target program freeze time. Reducing the number of pages to dump reduces the target program freeze time and performance impact. However, even the worst performing method (full-full) has frozen the target program for up to 900ms, which is often negligible compared to the total execution of the target program.

5.4.3 Measuring Time to Stop

As stated in Section 5.3.2, T_{SecGen} is always greater than the network RTT. Our approach requires $T_{stop} < T_{SecGen}$. We can either increase T_{SecGen} or lower T_{stop} to ensure the condition is met. T_{stop} depends on the ClientHello detection method. X-Ray-TLS relies on eBPF to monitor network traffic and push events to a Python user-space program over a ring buffer. Then, the Python program will freeze the target program with a SIGSTOP signal to allow a consistent memory snapshot. We use BCC's *ring_buffer_consume()* method to reduce latency

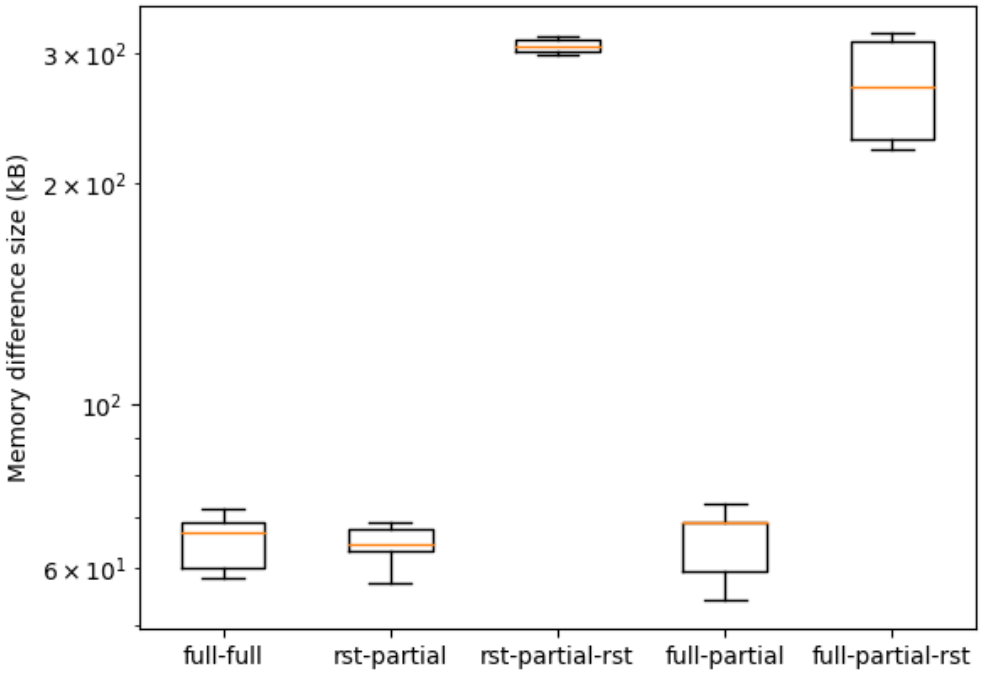

Figure 5.7: Size of memory snapshots difference for each memory dump strategy. Lower is better. Methods with intra-pages diff outperform per-page diff methods.

by continuously polling the ring buffer at the expense of higher CPU consumption. We estimate T_{stop} with the following methodology: after opening a TCP socket, we start a TLS handshake using Python's SSL module. We save the time when the handshake was started using *time.perf_counter_ns()*. Then, we save time when the event is received from the eBPF code. Finally, we measure T_{stop} by taking the difference between the two measured times. We measure with different 1-minute system load averages [73] (from 1 to CPU count) generated using the *stress* tool. The experiment shows that the time to stop is close to 1 ms when the load average is smaller than the number of CPU cores. When the system is overloaded (load average > CPU cores), the time to stop increases to a dozen ms (95%-percentile < 35ms). Results are shown in Figure 5.11. We believe this is because the scheduler does not have enough resources to run tasks as soon as they are ready, thus leading to delays. Considering overloading negatively impacts the performance of applications, production systems are rarely overloaded for long periods of time. Høiland-Jørgensen et al. [89] measure latency variations on internet. They showed that minimal RTT is above 20ms for 90% of Internet users. Therefore, for non-overloaded systems, the approach will work for more than 90% of internet users. However, Round Trip Time (RTT) might be very low (e.g., on the local network, it can be less than 0.2ms). Optimizing the pipeline to lower T_{stop} might not be enough. Therefore, another way to ensure the condition is met is to increase T_{SecGen}. This can be done by artificially delaying ClientHello or ServerHello packets. Delaying ClientHello

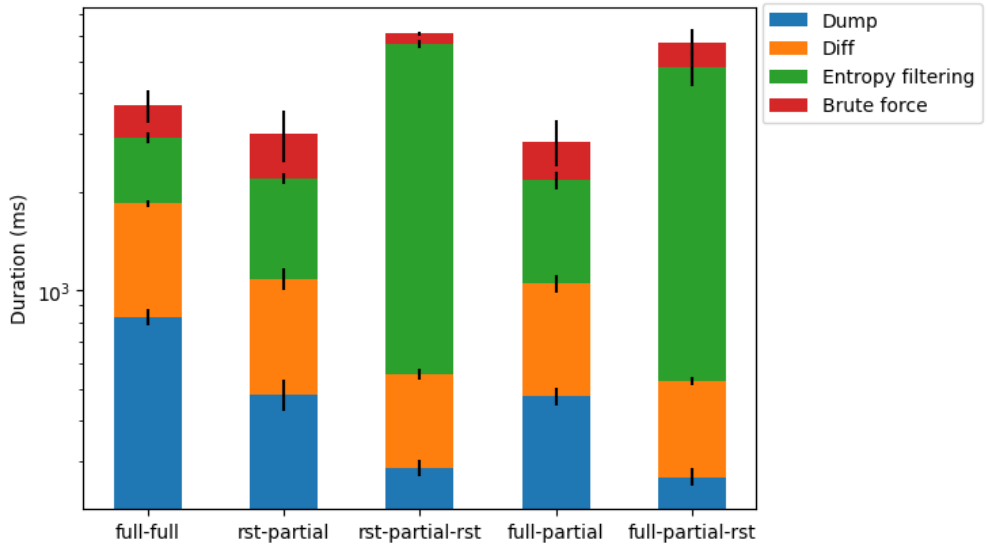

Figure 5.8: Duration of each step of session interception for memory snapshots strategies. Lower is better.

packets brings an implementation challenge. Indeed, ClientHello is used to trigger memory snapshots. Therefore, delaying ClientHello will delay the initial memory snapshot trigger as well. Therefore, it is preferable to delay ServerHello packets. In the case of domains with multiple A/AAAA DNS records, TLS clients might try to use a different IP address if the first IP does not answer before TLS timeout (i.e., as defined in the program code). This is unlikely to happen with a delay of a dozen milliseconds. However, if it happens, this is not an issue, as the client will initiate a new TLS session that will also be processed. We have verified using a proof of concept implementation that packet delaying might be implemented using the Linux traffic control subsystem. Furthermore, it allows fine-grained packet classification using eBPF classifiers to target ServerHello records only.

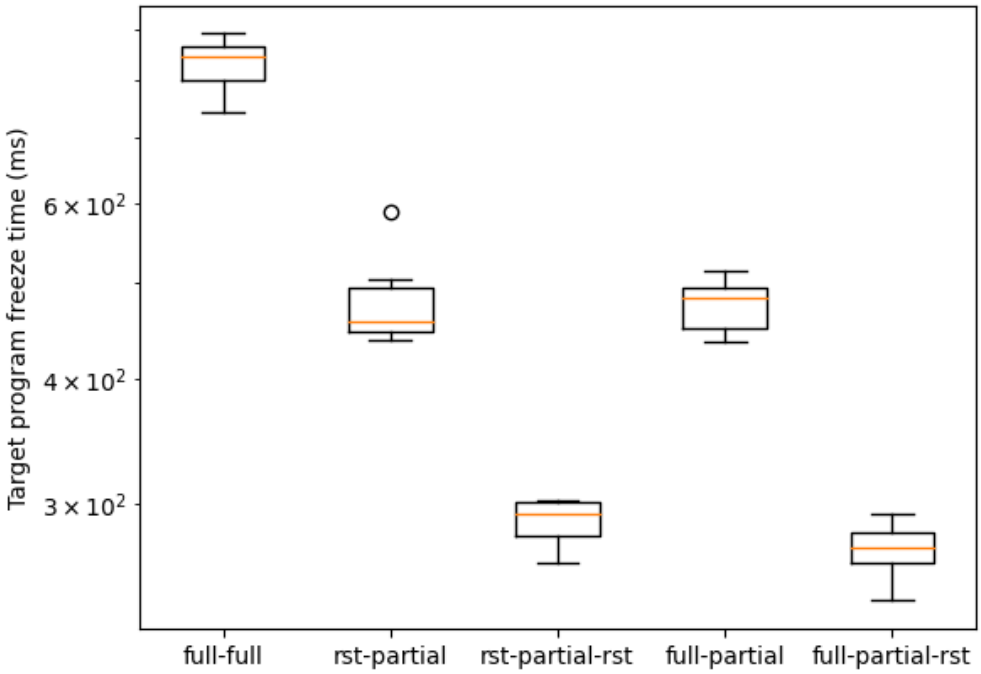

Figure 5.9: Target program freeze time for memory snapshots strategies. Lower is better. Strategies with partial dumps - only pages with soft-dirty bits - outperform methods with one or two full dumps.

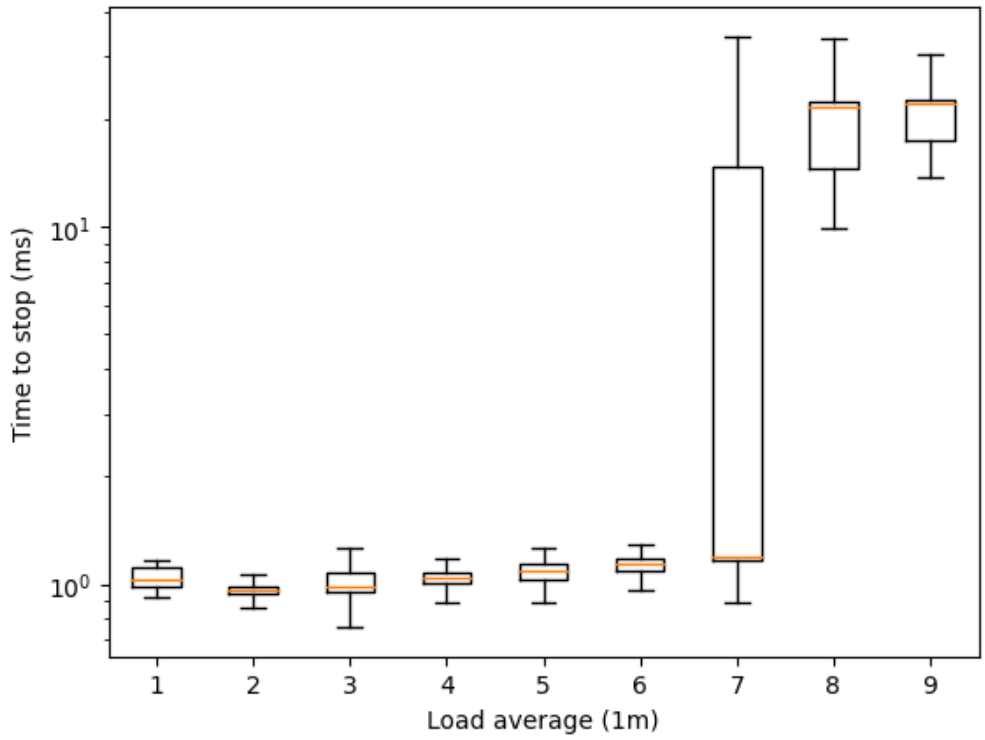

Figure 5.11: Evolution of the time needed to stop a target program (T_{dump}) concerning system 1-minute load average.

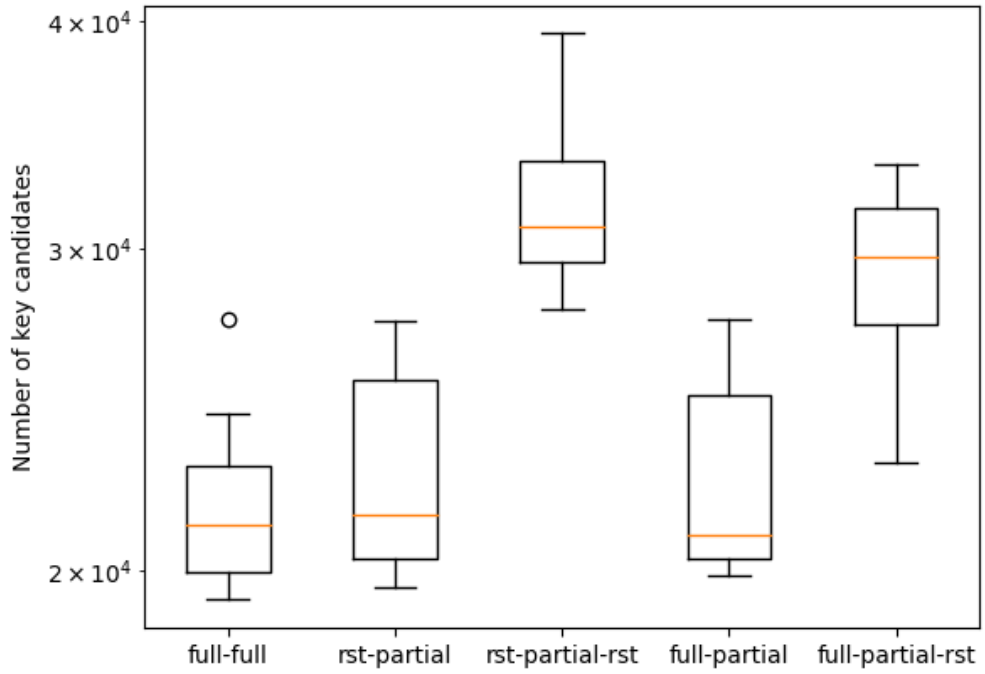

Figure 5.10: Number of key candidates for memory snapshot strategies. Lower is better. Strategies with smaller diff tend to have fewer key candidates. Note this is not linear because of entropy filtering (i.e., larger diff does not necessarily lead to a larger number of key candidates).

5.4.4 Limitations

We describe in the following paragraphs the limitations and possible improvements to overcome them. Limitations can be either fundamental, related to implementation choices, or weaknesses (i.e., that can become limitations if TLS libraries evolve).

Key availability By retrieving keys from process memory, our method is limited to TLS libraries that store keys in memory. At the time of writing, this is how major TLS libraries work. However, libraries could update their threat model to protect against the extraction of secrets from process memory. That would make key extraction with X-Ray-TLS more challenging or even impossible. We detail below mechanisms that can negatively impact the ability of our method to work. TLS 1.3 RFC [90] encourages (SHOULD) TLS implementations to remove secrets from memory as soon as they are not required anymore: "Once all the values to be derived from a given secret have been computed, that secret SHOULD be erased." For instance, handshake secrets should be removed from memory when the handshake ends. Likewise, old keys should be removed from memory on traffic key rotation. Therefore, we expect to find in memory only keys in use (i.e., application traffic secrets after the handshake) or that will

be used in the future (i.e., session resumption) and not keys used in the past (i.e., pre-master secret, master secret, handshake secrets). Instead of storing raw secrets in memory, TLS clients can store XOR-ed versions of secrets in memory. In a different context, this approach is used by rclone to obfuscate credentials in the configuration file: secrets are XOR-ed with a random constant key defined in rclone's source code [143]. While it does not prevent an attacker from decrypting secrets, it still forces the attacker to write custom code to handle rclone's approach. TLS libraries could use a similar approach by XOR-ing TLS secrets with a random key. This key may even be randomly generated at the TLS client process startup. However, if session resumption is expected between runs, the random key should be saved along with the session keys. This would significantly increase the complexity of finding keys in memory as it will not be possible anymore to test keys by trying to decrypt a TLS-encrypted packet. However, the approach might impact the speed of cryptographic operations. TRESOR [122] proposes storing AES symmetric keys in processor registers to limit key availability to attackers. However, TRESOR does not protect keys during handshakes. Therefore, promptly dumping the process's memory can still allow for key recovery. As Taubmann et al. [166] state, a program can outsource cryptographic operations to another process and communicate over a named pipe or equivalent mechanism. Therefore, the process initiating the connection does not store keys in memory. To overcome this limitation, future work would identify linked processes (file descriptor, pipe, shared memory, etc.) and then dump the memory of all processes. Linux kernel allows offloading TLS operations directly to the kernel by using TLS offload feature [97]. Therefore, keys will not be accessible from a user-space process. However, a custom kernel will still allow access to keys at the expense of an increased setup effort. Furthermore, code isolation mechanisms such as Flicker [112] can run TLS code in a secure environment. Hardware-enforced secure memory could also be considered to secure keys in memory from other programs.

Performance impact Our method requires freezing the target program for a short period to ensure a consistent memory dump. The memory dump might be large for large multi-threaded programs, leading to high freeze time.

Short-lived process When the target program is short-lived, we might not have enough time to dump process memory. In other words, the process has gone before we could freeze it to dump memory content. We discovered this limitation with `OpenSSL s_client` without any request body: the program exits immediately after the TLS handshake. We were not able to find another example of this limitation. However, opening a TLS connection without any data transfer is unlikely to happen in real-world usage. This limitation might be mitigated by freezing the target process directly from eBPF or a kernel module.

Security impact Our method requires to have root privilege on the client machine. Therefore, only privileged users can access TLS traffic in plaintext. When inspecting traffic from programs running in Docker containers, our program binds to the container's network namespace. This is required to intercept network-related system calls (e.g., connect()). This approach might introduce vulnerabilities. However, containers should not be considered secure sandboxes. Therefore, the security impact should be low.

Session resumption Session resumption is supported for TLS 1.2 if the initial session was decrypted. Indeed, the same master secret will be reused. In the context of TLS 1.3, the session resumption secret is independent of traffic secrets. Furthermore, we cannot extract this secret before a session resumption: we can not test key candidates to find the resumption key. Therefore, to support session resumption for TLS 1.3, we should keep in memory all the key candidates until session resumption. Unfortunately, this would significantly increase the storage requirement of our method.

Proof of Concept implementation We presented here a Proof of Concept (PoC) implementation of X-Ray-TLS in Python with complete support for TLS 1.2 and TLS 1.3. However, we did not fully optimize the performance of this implementation to reduce processing time (e.g., entropy filtering time, brute force time). Therefore, performance measurements presented in Section 5.4 could be significantly improved (e.g., by writing time-critical sections in C). QUIC support is limited to memory snapshots, but the key oracle test has not been implemented yet. Therefore, we ensured our approach works with QUIC by testing if QUIC session secrets were present in memory diff. In this specific experience, secrets were obtained using a complicit server. QUIC supports live migration of endpoints (e.g., client IP update) using a network-independent connection ID. We did not implement session tracking; we only relied on session tracking through network attributes (IPs, ports). Finally, X-Ray-TLS allows retrieving application secrets only. While this is enough for traffic inspection, Wireshark 3.6.7 cannot decrypt PCAP files with such a key file (i.e., traffic secrets only). We provided a patch to allow easy exploration of PCAP files with Wireshark. We make our proof-of-concept implementation of X-Ray-TLS available at https://github.com/eurecom-s3/x-ray-tls. While our implementation targets Linux, there is no fundamental limitation to porting the approach to other systems (e.g., Windows, Darwin, OpenBSD) as long as they provide the same underlying features: a mechanism to detect the beginning of a TLS session and a mechanism to dump process memory. As TLS is not directly aware of the IP version in use (IPv4, IPv6), our approach can also be adapted to work for IPv6.

The following section will discuss existing TLS inspection methods and use cases where traffic inspection is desirable.

Method	E2E security	Generic	Cooperative	Setup effort
Man-In-The-Middle	no	yes	yes	low
HTTPS Proxy	no	no	yes	low
Instrumentation	yes	no	no	high
SSLKEYLOGFILE	yes	no	yes	low
Memory analysis	yes	yes	no	medium

Table 5.2: Comparison of TLS inspection method families.

5.5 Discussion

TLS inspection solutions are widely deployed [49] and thus raise new questions about TLS security. This section compares existing TLS inspection approaches, considering their properties, merits, and limitations. Then, we present different use cases where TLS traffic inspection is used.

5.5.1 TLS Inspection Solutions and their Limitations

Different approaches were developed to allow TLS traffic inspection. Commercial products (e.g., Netskope [124], Avast [8]) provide easy-to-use monitoring solutions for TLS traffic to support security systems such as intrusion detection, information leak detection, or web anti-tracking solutions. Open-source implementations (e.g., MITMproxy [7], Snuffy [1]) provide interception solutions mostly targeting advanced users. In the following sections, we describe interception approaches along with their limitations.

TLS interception methods can be significantly different and can be defined by the following properties:

1. **E2E security**: no third-party have access to plaintext traffic.
2. **Generic**: work on TLS libraries without prior knowledge of library internals (e.g., without signature database or instrumentation).
3. **No cooperation**: the target program is not required to cooperate with interception such as dumping TLS keys by itself (i.e., SSLKEYLOGFILE), trusting a Certificate Authority, or using another hostname. We further discuss cooperative vs. malicious programs in Section 5.4.4.
4. **Protocols support**: support for widely used protocols: TLS 1.2, TLS 1.3 and QUIC.
5. **TLS hardening**: work with certificate pinning and Perfect Forward Secrecy.
6. **Setup effort**: does not require virtualisation or kernel modification. It can be deployed in minutes (e.g., containers).

7. **Decryption speed**: allow near real-time analysis.

We summarize in Table 5.2 a comparison of the main properties of existing methods. X-Ray-TLS is part of memory analysis methods. We observe that only methods based on memory analysis are generic and do not require target program cooperation. In the following sections, we will describe how existing approaches work.

Man-In-The-Middle (MITM)

MITM approach involves an attacker placing himself between the server and the client. The attacker pretends to be a legitimate server to the client and a legitimate client to the server. Therefore, this attack requires to be an active network attacker. This can be achieved using several methods, such as ARP spoofing, DNS poisoning, or tampering with local network parameters. This approach is commonly used by network security administrators that manage network and user devices (e.g., corporate network with corporate laptops) or by locally installed software such as anti-viruses. In this context, the attacker is often called a MITM proxy. However, this method has serious limitations in terms of usability and security.

MITM allows third-party software (i.e., other than client and server) to access plaintext data. In comparison, some methods only allow the recovery of encryption keys and store them securely, which reduces the risk of plaintext leaks.

Clients must trust certificates emitted by the MITM proxy. They are generated on the fly to match the requested common names and signed by the proxy's CA. It requires adding the proxy's CA to the trust store to avoid breaking certificate validation (i.e., preventing TLS connections from working). However, since some programs, such as Java, rely on their own trust store, enumerating all the relevant trust stores can be tiresome and lead to a significant issue in environments with much different software, such as continuous integration build servers.

Since the connection is now split into two connections, the client relies blindly on the proxy to validate the *real* server certificate. In the past, multiple examples of vulnerabilities introduced by MITM proxies were discovered [49]. Kaspersky [131] used a 32-bit key to identify generated certificates per website. Collisions enabled bypassing server certificate validation by MITM proxy, thus allowing MITM attacks between the proxy and the remote server. Avast Antitrack acts as a MITM proxy to block website trackers and ads. However, it was subject to many vulnerabilities [50], such as not checking server certificates (which then allows trivial MITM attacks), downgrading TLS to version 1.0, and preventing the use of modern encryption algorithms, e.g., that provide Perfect Forward Secrecy. Therefore, when MITM is used, the client depends on the MITM proxy for overall session security. MITM proxy's CA allows

the generation of valid certificates for any website. Therefore, it is sensitive and should be protected while generating on-the-fly certificates. Unfortunately, Kaspersky [130] failed to do so correctly. Finally, MITM will not work when a client uses certificate pinning.

Ensuring remote peer (i.e., the server from a client point of view) is legitimate can be challenging. Indeed, attackers can obtain valid certificates using a rogue or stolen certificate trusted by the client [68, 159]. Furthermore, without certificate pinning, full TLS compromise only requires the attacker to add a self-signed CA in the trust store. To address these concerns, certificate pinning is a client-side mechanism binding a host with a certificate fingerprint, whether the certificate is considered valid or not, with respect to the CA trust store. The expected certificate can refer to the server or any certificate in the chain (notably the root certificate, i.e., CA's certificate). The association can be static (defined in TLS client source code or local storage) or dynamic (discovered on the first connection). This approach is often used when the same entity manages the client and server (e.g., a mobile application connecting to the owner's servers). Special care should be taken when certificates are rotated unexpectedly (i.e., revoked due to suspected compromise). The HTTP Public Key Pinning (HPKP) [55] added certificate pinning support to HTTP. However, this is now deprecated because of its low adoption and the added complexity [133]. Instead of storing expected fingerprints in TLS clients or discovered on the first connection, they can be stored in the DNS, as defined in DNS-based Authentication of Named Entities (DANE [45]). It requires DNS records to be signed by DNSSEC. In conclusion, certificate pining prevents MITM by ensuring the client will receive a real server certificate instead of another certificate by MITM proxy (although valid). For instance, we discovered that VSCode's extension Copilot uses certificate pinning to connect to Microsoft's cloud. Certificate pinning is also widely used by Android applications [165].

HTTPS Proxy

Instead of opening a connection directly to a remote server, the client can be modified to open a connection to an operator-controlled domain that will act as a layer-7 proxy. For instance, a Docker client will connect to `docker.io` to fetch Docker images when using `docker pull image`. However, by using `docker pull dockerio.mydomain.com/image`, the docker client will connect to `dockerio.mydomain.com`, the operator-controlled domain. The controlled domain should be an HTTP proxy to forward HTTP requests and responses between the client and the original domain. However, the main limitation of this approach is that the target program must allow using the different hostnames to access the resource. This is often impossible for features such as telemetry or other hard-coded URLs of proprietary programs. This approach breaks end-to-end security like with MITM, lowering overall security [32].

Server Key Extraction

A TLS session can be decrypted by retrieving session keys. Without Perfect Forward Secrecy (PFS), session keys are encrypted using the server's public key. Therefore, by retrieving the server private key (e.g., owner cooperation, server compromise), an attacker can decrypt TLS sessions. However, this approach can only be used when the attacker controls the remote server (e.g., internet website). Furthermore, this approach does not work with TLS 1.3: PFS is optional in TLS 1.2 but mandatory in TLS 1.3. To ensure PFS, TLS decorrelates the session secret from the long-term secret (the server's private key) using an (EC)DHE key exchange signed using the private key. Thus, disclosing the private key does not reveal session keys from past sessions.

Instrumentation

Instrumenting target programs almost always allows retrieving plaintext traffic. Function hooks can retrieve plaintext traffic before the TLS library encrypts it. Instrumentation can also reduce connection security (e.g., disable certificate pinning [165]). However, function hooks are hard to use for programs statically linked with symbol stripped [2]. Indeed, the memory offset of relevant functions in the code should be determined manually. This is often the approach taken when performing malware analysis, where obfuscations prevent the usage of any other approach, and manual interaction becomes necessary. The instrumentation is challenging for software with large memory footprints, such as electron-based applications. Furthermore, the instrumentation is specific to each target program, thus increasing the setup effort and making it unrealistic for environments with many different software. Finally, hooking program internals adds a risk of introducing instabilities.

SSLKEYLOGFILE

Mozilla proposed the text-based file format NSS Key Log [120] to store TLS session secrets. It aims to provide a standard format for logging TLS secrets across programs. Wireshark supports the NSS Key Log format to decrypt TLS traffic. With compatible applications, dumping session keys is enabled using an environment variable called `SSLKEYLOGFILE` (*de facto* standard environment variable name). However, target programs must support the feature to allow TLS interception. It is up to programs to decide whether they allow TLS key logging: this is a cooperative approach. There is no guarantee that all TLS session secrets will be logged. This feature is often disabled by default at compile time (e.g., in Debian packages). Indeed, it might lower TLS security. An unprivileged process running under the same user can set the environment variable (e.g., in shell startup script) and retrieve plaintext TLS traffic. Therefore, it raises a major issue for large-scale deployment. In addition, keys are dumped by the target

program to the filesystem: all programs running under the same user will have access to TLS secrets. Thus, while this approach is relevant for debugging cooperative programs in a controlled environment, it raises a major issue for large-scale deployment.

Downgrade Attacks

Downgrade attacks are a common type of attack to force communication to use a less secure channel. We distinguish HTTP downgrade (e.g., HTTPS to HTTP) and TLS version downgrade.

HTTP downgrade Blocking HTTPS connections (e.g., by blocking port TCP 443) can force a client to use HTTP as a fallback mechanism. However, HTTP Strict Transport Security (HSTS) [87] allows a web server to inform a web client through HTTP headers that insecure connections should be rejected (e.g., HTTP or HTTPS with invalid certificate). HSTS is a Trust On First Use (TOFU) approach, which protects only following connections after a successful connection. To overcome this limitation, web browsers like Firefox, Chrome, and Edge implement an HSTS preloaded list. This list contains websites that support HSTS. Furthermore, HSTS requires a local storage capability, which is not always available (e.g., ephemeral containers). Web browsers do not allow users to access a website (no security bypass mechanism) with an invalid certificate if HSTS is enabled for this website. According to F5 [30], in 2021, 20% of the top 1 million websites use HSTS.

TLS version downgrade An active attacker could attempt to force the session to use TLS 1.2 (or an older TLS version) while both the server and the client support TLS 1.3. However, TLS 1.3 implements a downgrade protection mechanism to prevent such an attack. To this aim, TLS 1.3 servers facing older clients include the "DOWNGRD" byte string in the server random field. Therefore, if a TLS 1.3 client is proposed to use TLS 1.2, but the server random includes the "DOWNGRD" string, the connection might be under a downgrade attack, and the client should refuse the connection. Furthermore, an attacker cannot modify the server random as it is used to generate shared secrets: changing the server or client random strings will break the key exchange.

Memory Analysis

TLS clients and servers must keep track of session keys in secure and fast storage. As shown in Section 5.4, major TLS libraries store TLS keys in process memory. X-Ray-TLS leverages this property to extract keys directly from memory. However, each TLS implementation has its own code architecture and way of storing keys in memory. Multiple TLS libraries are available [180], both open-source and closed-source. Having different TLS implementations

increases the overall resiliency as a vulnerability in one implementation might not impact another implementation. In the context of session decryption using process memory, it increases the complexity as the method should be generic enough to work with different TLS stacks. Furthermore, the TLS 1.3 specification specifies that secrets should be removed from memory as soon as they are not required anymore. In particular, we observed that this recommendation is followed by the widely-used library OpenSSL. For instance, handshake secrets are removed from memory as soon as the handshake is completed. Therefore, memory snapshots should be done promptly to extract secrets.

OS protections The operating system can also protect process memory from undesirable access with different mechanisms. Address Space Layout Randomization (ASLR) randomizes the memory stack and heap addresses at the operating system level. This helps to mitigate unauthorized access to memory content by buffer overflow or similar techniques. Similarly, Akamai's *secure heap* [25] protects memory pages containing RSA keys. Akamai's *secure heap* adds guard pages (pages with the flag *PROT_NONE*) around the secure memory, leading to a segfault on invalid accesses near the secure memory. To prevent pages containing TLS secrets from being swapped to disk, the kernel provides *mlock* system call to flag memory pages containing secrets. These pages are guaranteed to be kept in RAM until unlocked. However, these protections do not impact X-Ray-TLS as we leverage kernel pseudo-file `/proc/{pid}/mem` to access process memory.

Multiple, simultaneous connections TLS clients may use multiple independent TLS connections to fetch resources (e.g., web pages). Usually, there is one TLS connection per host (i.e., IP and port). For instance, Singanamalla et al. show that loading one page generates an additional median of 16 TLS handshakes [155]. Therefore, interception methods should be able to decrypt TLS sessions to allow near-real-time traffic analysis quickly. Furthermore, TLS sessions can be started simultaneously (i.e., in a lower delay than network round trip), so handshakes may overlap. This might impact, especially for methods that rely on incremental memory snapshots (e.g., using soft-dirty flags). The following section will focus on different use cases where TLS inspection by extracting session keys from memory is desirable.

5.5.2 Use Cases

TLS inspection can be used in multiple contexts: security analysis, to support root cause analysis, or to assist the program's internal discovery (e.g., reverse engineering). MITM is commonly used for corporate network analysis. Despite its complexity, instrumentation is preferred for analysis of obfuscated programs (e.g., malware). Using SSLKEYLOGFILE or HTTP proxy is the easiest option when target programs are cooperative. Finally, for local

non-cooperative programs, X-Ray-TLS is an appropriate option with little setup effort. Below, we detail different contexts where X-Ray-TLS is relevant.

Security analysis of non-cooperative programs For security and privacy researchers, inspecting TLS traffic can help better understand how programs work. Encrypted traffic from programs using TLS hardening, such as certificate pinning, cannot be easily decrypted: MITM will not work, and instrumentation is a time-consuming and complex task. For instance, Copilot [63] is an AI-based peer programming assistant. It relies on Microsoft's cloud (over HTTPS) for making code predictions based on user development workspace. Inspecting TLS traffic allows assessing what data is sent to the cloud for code predictions. Similarly, Google Chrome processes various personally identifiable information (PII) as stated in their privacy policy [69]. Verifying which data is sent to Google may be challenging. While Chrome currently supports SSLKEYLOGFILE, there is no guarantee that all TLS keys are logged (e.g., telemetry) or that they will continue to work in the future.

Container hosting X-Ray-TLS allows container hosting providers to decrypt customer traffic for security purposes without prior knowledge or modification of the container. For instance, 5G networks are composed of containers managed by customers. Network providers have little knowledge about what is running in containers. Decrypting TLS traffic without requiring any change from the customer side and with minimum overhead can be leveraged to detect service abuse, information leaks, or network threats. This can also be provided as a service to the customer for increased observability of their workloads.

Intrusion detection According to F5 [57], 71% of malware installed through phishing hides in encryption (e.g., HTTPS, IMAPS). Our approach allows corporate security managers to increase visibility on encrypted network flows without the drawbacks of MITM. Plaintext traffic could then be forwarded to an Intrusion Detection System for further analysis as an end-point security mechanism.

Improving security of CI/CD pipelines CI systems have become a target [101, 128] for compromising production systems. Indeed, gaining access to CI environment allows attackers to gain significant privileges such as altering production artifacts, accessing pipeline secrets such as access tokens (that are very often over-privileged [100]), or scanning the local network of CI/CD workers nodes. For instance, dependency confusion [14] is an effective method to run malicious code in a CI environment from the internet (i.e., without access to source code management systems or internal network). Furthermore, a one-time compromise of a CI/CD pipeline can enable long-term compromise using a self-replicating mechanism without

leaving any trace in the source code [118]. Therefore, there is a strong interest in solutions that allow the monitoring of network traffic in CI/CD environments. As discussed in Section 5.4.1, existing methods for TLS inspection are difficult to use at scale, considering the high diversity of software used in CI/CD pipelines. X-Ray-TLS can be deployed on CI/CD worker nodes to allow decryption of TLS sessions made by CI/CD environments without requiring any modification of pipelines and with low overhead. Then, plaintext traffic can be saved for post-attack analysis or forwarded to an intrusion detection system for live threat detection.

Root cause analysis of software failures Continuous Integration (CI) is widely used in software engineering to run regular checks on code changes. These checks (called a CI build) often rely on remote resources (e.g., package dependencies, API). When they fail, root cause analysis may be challenging. CI logs are often noisy because of the number of error messages and are highly heterogeneous, thus challenging to analyze. Inspecting HTTPS traffic might help to leverage the strong semantics of HTTP protocol (i.e., status code) along with the error message in the HTTP body. This ultimately helps to provide automatic root cause analysis of build failures.

Inferring internal program behavior OpenConflit [22] proposes inferring the internal state of video games using memory analysis. Our method can be used similarly to inspect traffic between the local program and remote servers (i.e., in multiplayer mode). If traffic is protected by a custom algorithm (i.e., not TLS), our method should continue to work with minimal modifications. Indeed, the only modifications required will be to identify network events that represent the beginning and end of the handshake and write a custom key test oracle (i.e., to test key candidates). We do not claim to support hardened programs such as malware (that may heavily obfuscate their memory). However, since memory obfuscation is hard to implement in practice and makes debugging more complex, many programs still use standard libraries for processing TLS.

5.6 Proof of concept implementation

The latest version of the proof-of-concept implementation of X-Ray-TLS is available online at https://github.com/eurecom-s3/x-ray-tls. The initial version of X-Ray-TLS (i.e., when the paper was published) is available at https://doi.org/10.5281/zenodo.10341300.

5.7 Conclusion and Future Work

We presented X-Ray-TLS, a new approach for generic and automated inspection of TLS traffic of local programs. It requires little setup effort and does not directly interact with the target program for minimum intrusiveness. X-Ray-TLS works on major TLS libraries and large software such as Firefox, does not require virtualization, and is fully generic to support current and future applications. Therefore, it directly applies to environments where many different software are used, such as CI/CD environments. To support future work using data-driven approaches, X-Ray-TLS collects pre-key and post-key memory content (i.e., content before and after secret keys) and memory offset of keys for each run. Data-driven approaches would help to reduce the search space by automatically discovering key storage patterns and, therefore, helping to focus the key search on specific memory areas. Data-driven pattern detection would combine the effectiveness of signature-based approaches while still being generic.

5.8 Summary

While internet communications have been originally all in the clear, the past decade has seen secure protocols like TLS becoming pervasive, significantly improving internet security for individuals and enterprises. However, encrypted traffic raises new challenges for intrusion detection and network monitoring. Existing interception solutions such as Man-In-The-Middle are undesirable in many settings: they tend to lower overall security or are challenging to use at scale. We present X-Ray-TLS, a new target-agnostic TLS decryption method that supports TLS 1.2, TLS 1.3, and QUIC. Our method relies only on existing kernel facilities and does not require a hypervisor or modification of the target programs, making it easily applicable at scale. X-Ray-TLS works on major TLS libraries by extracting TLS secrets from process memory using an algorithm to reconstruct memory changes. X-Ray-TLS supports programs that use TLS hardening techniques, such as certificate pinning and perfect forward secrecy. We benchmark X-Ray-TLS on major TLS libraries, CLI tools, and a web browser. We show that X-Ray-TLS significantly reduces the manual effort required to decrypt TLS traffic of programs running locally, thus simplifying security analysis or reverse engineering. We identified several use cases for X-Ray-TLS, such as large-scale TLS decryption for CI/CD pipelines to support root cause analysis or detecting software supply chain attacks.

Chapter 6

Fault localization in CI/CD build logs

This thesis explores different strategies to provide automated root cause analysis of CI/CD failures. Fault localization is a possible strategy to assist developers in troubleshooting CI/CD failures. Therefore, this chapter explores strategies for inspecting network traffic, especially encrypted traffic.

6.1 Introduction

CI systems produce a large number of logs. When failures occur, which can happen thousands of times a day for large CI systems, operators have to investigate to find the root cause of the failure. We propose a new approach that leverages the belief evolution of RNNs during the inference phase to provide automated root cause analysis of failures. We aim to answer the following research questions:

- **RQ6.1**: Does the belief evolution of a recurrent neural network (RNN) allow the identification of properties (e.g., discriminative pattern mining) of log given as input?
- **RQ6.2**: How to leverage RNNs belief evolution to localize the root cause message in CI failed build logs?

Unlike the traditional approach of machine learning algorithms, ChangeMyMind is a new approach to leverage the belief evolution of recurrent neural networks. Therefore, ChangeMyMind has the following properties:

1. **Easy to build dataset**: it only requires successful and failed build logs, not root cause messages.
2. **No preprocessing**: no preprocessing, such as log parsing, is required as the algorithm will work on the sequence of tokens.

3. **Generic**: No prior knowledge of logs is required. Therefore the algorithm can be applied to any kind of log labeled as success or failure: CI/CD build logs, transaction logs, test logs, etc.

4. **Fast**: Log analysis in done in less than 0.5s for usual log size and hardware (detailed in Section 6.6.3).

In this chapter, we detail the training and analysis phases of ChangeMyMind. We propose a method to build a synthetic dataset using fault injection, and we manually label a dataset of failed build logs. We then evaluate ChangeMyMind and compare results to a baseline method. We release a Python implementation and artifacts used in this chapter at https://purl.org/changemymind-artifacts.

The remainder of this chapter is structured as follows. In Section 6.2, we describe what is a continuous integration build and what are the major challenges for root cause analysis of broken builds. In Section 2.4, we review approaches explored in the literature to perform automated root cause analysis of CI failures. In Section 6.5, we detail ChangeMyMind, a new approach that leverages RNNs belief evolution to localize root cause messages in build logs. In Section 6.6, we present the datasets we developed and the evaluation of our approach. In Section 6.7, we discuss failure patterns in CI systems and the use of expert systems for failure analysis. Finally, we conclude and outline some future works in Section 6.8.

6.2 Root cause message in CI/CD build logs

CI orchestrators save the output of commands ran during the build (i.e., stdout, stderr on Linux) to a build log. Therefore, the build log is a textual document that concatenates the output of all commands. Depending on the CI orchestrator, the build log can be divided into smaller logs for each command. In CI orchestrators such as Jenkins or Travis, the full build log is shown to the user for investigation when a build fails. Therefore, users are strongly interested in automated solutions that would reduce the search space for the root cause of a failure.

A build is an ordered sequence of commands. When a command fails, the build will usually be stopped. However, the root cause of the failure might not be easily identified at the end of the build log. For instance, a test framework such as pytest will continue to run all tests even if a test fails. Commands can run in parallel and will not be stopped until they exit. Finally, the root cause might create other failures not directly after the root cause. For instance, if a file is not created, the build will fail only when the file is required, potentially a long time after the error related to the creation of the file. Searching for error terms such as `error` or `fatal` leads to numerous results in build logs. Real-world datasets such as the one we use in Section 6.6

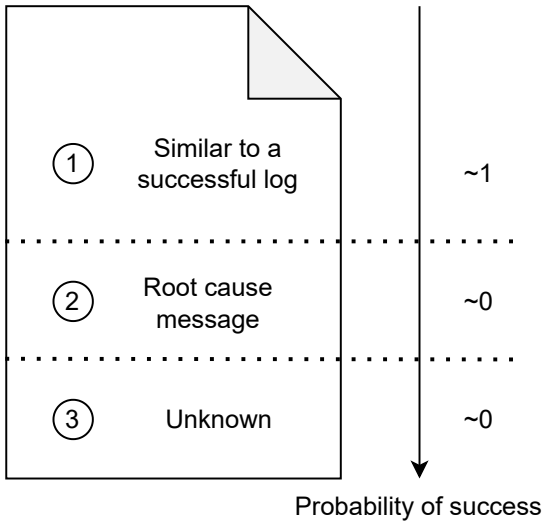

Figure 6.1: Overview of the different parts of a continuous integration failed build log. On the right is the probability of success of the build for each part.

have, on average, 20 error lines per failed build. Furthermore, some error lines, such as retries, are irrelevant to identifying the root cause.

We define the root cause message as the first log message, such as the failure of the build is inevitable. In other terms, when the root cause message is encountered in the log, the probability of failure of the build is 1. Therefore, a failed build log can be divided into three parts, as shown in Figure 6.1. Before the root cause message (part 1), the log is similar to a successful build. Log messages in this part may contain error messages (e.g., retries, suppressed failures); however, they are unrelated to the cause of the build failure. During part 1, the probability of success is high, considering that a small fraction of all builds fail. At some point (part 2), the root cause message is logged. Note that the root cause message can be one word, one line, or even several lines. After the root cause message (part 3), log messages cannot be easily anticipated: they depend on the root cause. For instance, part 3 can be a stack trace, cleanup scripts, other errors because of the original failure, or even no log (e.g., system freeze). Build logs have different sizes. Indeed, the execution flow is different because of non-deterministic events (e.g., retries) or different code inputs (e.g., commits). Therefore, the algorithm must allow to work on sequences of variable lengths. In our approach, we will leverage the change in the prediction of success to locate the root cause message.

6.3 Recurrent neural networks

Artificial neural networks (ANNs) are part of a field in machine learning that leverages a neuronal organization. An ANN is a set of neurons connected by edges. Training a neural network is done by updating the weights of neurons and edges to improve the performance of the classification tasks. An optimization algorithm (e.g., SGB, Adam) determines how to update weights. Deep learning refers to the use of multiple layers in the ANN. Recurrent Neural Networks (RNNs) are a class of ANN where neurons can create a cycle, allowing output to depend on the input but also from an internal state of the network. RNNs allow for processing variable sequence length input and have been proven successful in processing sequential data (e.g., speech recognition).

6.4 Model Behavior Analysis and RNNs Interpretability

AI interpretability refers to the methods and techniques used to explain and understand the decisions, predictions, or actions made by artificial intelligence (AI) models, particularly those considered "black boxes" like deep learning models. They aim to enhance transparency, trust, and understanding in AI systems by making their internal workings more accessible and comprehensible to humans. In the context of this thesis, we postulate that AI interpretability can provide root cause analysis. Therefore, we explore existing work on AI interpretability. Considering CI/CD logs have varying lengths, we focus on Recurrent Neural Networks (RNNs), which allow the processing of variable-length inputs.

Multiple solutions to increase the interpretability of RNNs have been proposed in the literature. LIME [145] generates locally interpretable models from individual predictions by perturbing the input and observing the corresponding changes in the output. However, the interpretability provided by LIME can be unstable [3]. Layer-wise Relevance Propagation [102] (LRP) assigns relevance scores to input features by propagating them through the network layers. It helps to understand the impact of each element on the output. However, LRP might be computationally expensive for large models [4]. Gradient-based methods analyze the gradients of the model with respect to the input features to determine their importance. They are based on partial derivatives of the prediction function. The main disadvantage of this method is the noise in the function [36]. Finally, occlusion-based explanations [105, 192] iteratively occlude parts of the input sequence to observe the impact on predictions. It enables the identification of regions of the input sequence that impact the prediction. However, occlusion-based explanations require significant computation time, especially for long sequences. Model behavior analysis was also used to support troubleshooting of software programs implementing deep neural networks [127, 175, 176].

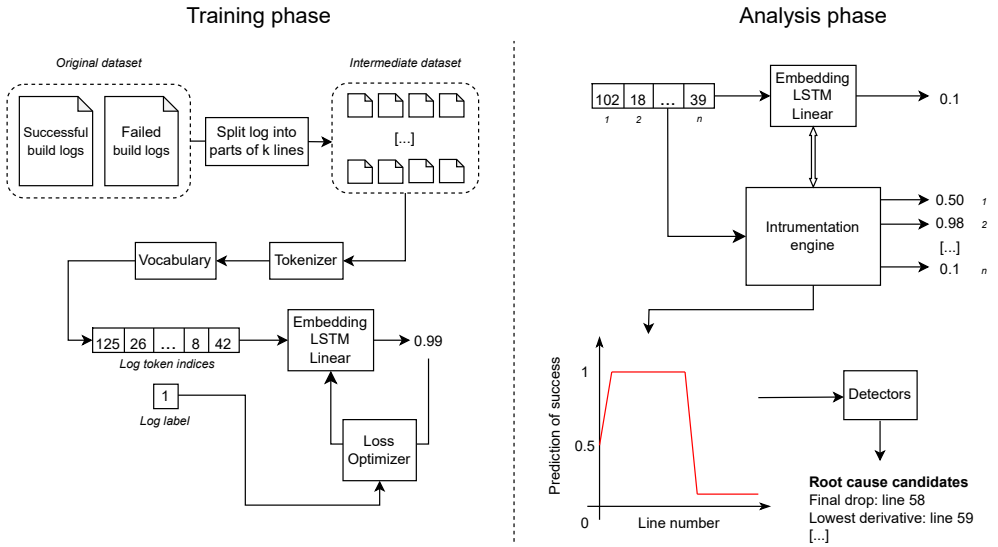

Figure 6.2: Overview of ChangeMyMind. During the training phase, the algorithm learns failure patterns from successful and failed build logs. During analysis, an instrumentation engine extracts intermediate predictions that detectors use to identify root cause candidates.

6.5 ChangeMyMind

We propose ChangeMyMind, a new approach that leverages RNNs belief evolution to identify root cause messages in logs. We applied the proposed approach to CI/CD build logs. As described in Section 6.2, CI builds are labeled as either successful or failed. When a build is labeled as failed, identifying why it has failed is called root cause analysis. ChangeMyMind helps to identify a subpart of the log where the root cause message is likely to be present. A high-level overview of the approach is depicted in Figure 6.2.

During the training phase, the algorithm learns root cause patterns. Unlike classical supervised training, it does not require a labeled dataset of root cause patterns. Instead, we propose to train the algorithm on a downstream task to predict whether a build is successful or failed. We call it a downstream task (or a pretext task) as it is not directly useful for end users: the CI/CD orchestrator already provides whether a build is successful or failed. However, this task aims to train the algorithm to distinguish failed builds from successful builds. We postulate this is a way to train the algorithm to learn build failure patterns. The training phase can be done offline on a large corpus of logs. Building a dataset of successful and failed logs is trivial compared to labeling root cause messages in logs. We detail in Section 6.5.2 the training process.

During the analysis phase, we aim to localize the root cause of failed builds. The failed build is used as input for the downstream task (i.e., predicting if the build is a failed or a successful

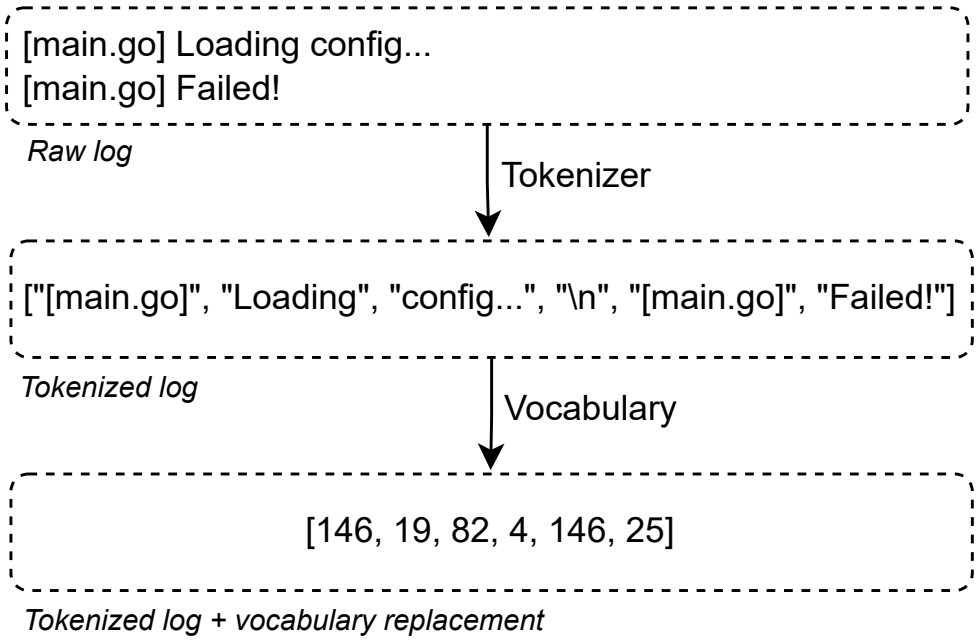

Figure 6.3: Example of the tokenization process. The raw log is first tokenized before being transformed into vectors using a vocabulary.

build). While this prediction task is useless,[1] it allows the instrumentation engine to save intermediate predictions made during the inference phase. We refer to it as the belief evolution of the algorithm. The belief evolution is different for different logs. The algorithm is fed on a token-by-token basis. Indeed, a line-by-line approach would require identifying log events (i.e., separate static and dynamic parts of the log message) to represent identically two log messages from the same log template. However, extracting log events is challenging in CI builds logs, as described in Section 2.4. Therefore, intermediate predictions will refer to the algorithm's output for each token. We present the details of the analysis process in Section 6.5.3.

6.5.1 Tokenization

ChangeMyMind does not require prior knowledge of logs, such as the log structure (i.e., to extract log events). This is an essential property for CI/CD logs because log parsing is challenging. Indeed, CI/CD pipelines usually involve many different software (e.g., scripting, CLI, test framework), leading to highly heterogeneous log message formats. Logs, in general, and build logs, in particular, are textual documents. We adopt a simple tokenization process that consists of splitting logs by spaces and $\backslash n$ characters. Line returns are kept in the tokenized log, but spaces are not. Indeed, line returns have a strong semantic in logs as they often indicate distinct log messages (Figure 6.3).

[1] We already know this is a failed build

Tokens are converted into integers before being fed into the embedding layer using a vocabulary. A vocabulary is a bijective mapping (i.e., dictionary) that assigns a unique index for each token. The vocabulary size is the number of tokens in the vocabulary. Usual values for vocabulary size range from a thousand to dozens of thousands of tokens. We detail in Section 6.6 the values used in our experiments. If the vocabulary size is equal to n, only the n tokens that appear the most often will be encoded, while other tokens will be replaced by the special token `<unk>`, corresponding to out-of-vocabulary tokens.

6.5.2 Training Phase

The training phase refers to the algorithm's training to classify successful logs from failed logs. This is a downstream task. The training dataset should be balanced: it should contain the same number of successful builds and failed builds. Considering there are more successful builds than failed builds, the dataset size is usually limited by the number of failed builds. Section 6.5.3 details the reasoning behind the need for a balanced training dataset. We randomly split the dataset into 90% for training and 10% for testing. The test will measure the classification accuracy on logs not seen during training. This setup, while differing from the classical train/validation/test split, still provides guarantees that the model's performance is verified on unseen data, maintaining the integrity of the testing process and helping to prevent overfitting. If the dataset is sufficiently large, the test set will be representative, making a two-way split into training and testing sets still adequate for evaluating model performance.

We split logs into parts of k lines only for the training phase. The number of lines per log part, k, is a parameter of the algorithm. All parts of each log, referred to as sublogs, will have a label as follows: if the parent log (i.e., the log they come from) is a failed log and the sublog contains error terms, then the label will be 0 (i.e., failed log), otherwise the label will be 1 (i.e., successful log). We consider that the sublog contains an error term if it contains (case-insensitive) any of the following terms: error, fatal, exception, fail, abort, problem, or issue. Splitting logs is implemented on the fly during the training phase. Therefore, it does not require modifying the original dataset. If the number of sublogs labeled as successful is greater than the number of sublogs labeled as failed (which is likely), we perform random sampling to create a balanced dataset.

6.5.3 Analysis Phase

The analysis phase refers to locating root cause messages in a failed build. The build to investigate is fed as input to the downstream task. During inference, the instrumentation engine will save intermediate predictions (i.e., belief evolution). At the beginning of the log (i.e., without even reading the first token), we expect the algorithm to have no information

about the label of the log because the algorithm was trained on a balanced dataset. Indeed, with a balanced dataset, at the beginning of the log, the algorithm cannot predict which label is more likely as both labels are equally likely. Therefore, the belief evolution should be close to 0.5 at the beginning of the log. Then the belief will evolve when the algorithm is fed with the sequence of tokens.

Line prediction

The instrumentation engine saved intermediate predictions for each token during the analysis phase. However, in the context of a log, we focus on a line-by-line basis. We explored multiple strategies to assign a belief to a logline: average, median, minimum, and maximum of the belief of all tokens of the line. This allows for graphing the belief evolution with log numbers on the x-axis instead of tokens, which have less semantics for users. The belief evolution does not give by itself where the root cause message is located in the log. We used different heuristics called detectors to identify areas of the log likely to contain root cause messages.

Detectors

After extracting the evolution of the prediction concerning log lines, an analysis should be performed to identify areas of interest in the input log. This analysis of the belief evolution is done by detectors. Detectors can be applied to log lines' belief evolution or tokens' belief evolution. Detectors return one or multiple log lines that are predicted to be root-cause messages. We will refer to them as root cause candidates. We propose different detectors detailed below.

Belief drop Considering failed build logs are labeled as 0 and successful build logs as 1, we expect the belief evolution to drop when the algorithm encounters root cause patterns during the processing of the log. Therefore, we propose two alternative detectors.

Final drop detector identifies the root cause of the build when the belief evolution goes under 0.5 and never crosses the 0.5 threshold again. The belief might cross multiple times the 0.5 threshold. The latest decreasing crossing will be identified as the root cause location in this context. The detector can be implemented using the following formula:

$$r_{final} = max(\forall x \text{ such as } y(x) > 0.5))$$

where y is the belief evolution function and x the line number. This detector returns zero (i.e. if the belief evolution never crosses the threshold) or one root cause candidate.

First drop detector identifies the root cause of the build failure when the belief evolution goes

under 0.5 for the first time. Considering the belief evolution is expected to start close to 0.5, we propose to add an offset t. This helps prevent the detector from being triggered at the beginning of the log, where the belief evolution is volatile. However, it prevents the detection of a root cause before the offset. The detector can be implemented using the following formula:

$$r_{first} = min(\forall x > t \text{ such as } y(x) > 0.5))$$

The usual value of t was chosen to be 10% of the log size. We presented detectors with a belief threshold of 0.5. However, the belief threshold can be set arbitrarily. We experimented with values of 0.5 and 0.1 for the first and final detectors. Therefore, we used 4 detector variants based on belief drop.

Lowest derivatives This detector identifies the root cause where the left derivative of the belief function is minimal (i.e., strongest drop). The belief function is derivable on $]0; n[$ where n is the size of the log. We set the derivative to 0 for $x = 0$ and $x = n$.

$$r_{ld} = x \text{ such as } y'(x) = min(y'(x)\ \forall x)$$

where y' is the left derivative of the belief evolution function, and x is the line number. This detector can be extended to return the top k lowest derivatives. In this context, the detectors will return multiple root cause candidates sorted by the derivative (i.e., from lowest to highest).

No detector Instead of relying on a machine-based detector, we tried to return the belief evolution without further processing. Therefore, the user can explore different areas of the log based on his interest.

In Section 6.6, we evaluate the performance of the different detectors.

6.5.4 Hyperparameters Optimization

Finding optimal hyperparameters (i.e., embedding dimension, batch size) is challenging. Unlike classical supervised machine learning tasks where the output can be compared with known labels, we do not rely on a dataset of labeled root cause locations. Indeed, the root cause prediction is made by analyzing the belief evolution during the inference phase of a downstream task. Considering successful builds are labeled as 1 and failed builds as 0, we expect the belief evolution to be close to 1 before the root cause message and close to 0 after. Therefore, we can compute a score that estimates to which extent the belief evolution of a build log is close to this pattern.

We define a custom metric C_x (*Correctness*) for one log using:

$$C_{log} = \sum_{i=1}^{r} (1 - y_i)^2 + \sum_{i=r}^{n} y_i^2$$

where r is the predicted line of the root cause message, n the number of lines in the log, and y_i the algorithm's output for the line i (intermediate predictions).

This score does not measure if the root cause is correctly identified but only if the belief evolution is close to the expected pattern. This score is computed on the entire log, similar to during the analysis phase (Section 6.5.3).

The score of a dataset of logs is the sum of the scores of its logs:

$$C_{dataset} = \sum C_{log_i}$$

By construction, the correctness score of a dataset depends on the logs in the dataset, particularly the number of logs and the size of the logs. Therefore, dataset correctness scores can be compared only if computed on the same dataset. For instance, correctness scores are useful for comparing hyperparameter settings to find the set of hyperparameters that allow beliefs to be closest to the expected pattern. However, the correctness score does not allow for improved training (e.g., update weights). In the following section, we evaluate the performance of ChangeMyMind on synthetic and industrial datasets.

6.6 Experiments and Evaluation

This section details our experiments to evaluate ChangeMyMind. We measure the correctness of the method (i.e., are detectors correctly identifying root cause messages) and the performance impact of our approach. We define the prediction error as the absolute difference value between the predicted line and the real root cause line. The prediction error pe can be expressed as $|r_t - r_p|$ where r_t is the real root cause line and r_p is the predicted root cause line. We propose two developer-focused thresholds for the prediction error: 5 and 50 lines. Indeed, if the prediction error is below 5 lines, it means that the real root cause will be in a subset of the log of 10 lines (5 lines backward and 5 lines upward). Ten lines of the log can be easily integrated into the CI system to show up on the build page, allowing almost immediate root cause analysis by the operator. The same reasoning applies if the error prediction is below 50 lines: 100 lines can usually be shown on a single page, thus allowing quick root cause analysis

(i.e., no scrolling required).

Experiments have been done with the following neural network: an embedding layer (dimension 64), a bidirectional LSTM layer (hidden size 256, 2 layers), a dropout layer (probability 0.2), and a linear layer (output 2 classes). We used the CrossEntropyLoss loss function, the Adam optimizer, and the learning rate scheduler StepLR. The learning rate scheduler step is only triggered if the loss of the test split is not decreasing, which indicates the model is overfitting. We used a batch size of 4, an initial learning rate of 0.001, and 20 epochs. We split the log into chunks of 5 lines ($k = 5$). We used a vocabulary size of 5000 tokens. These parameters have been found using a grid search approach to minimize the *correctness* score defined in Section 6.5.4. The implementation is written in Python using PyTorch. We release the implementation as part of the artifacts.

From the industrial dataset defined in Section 3.2, we extract the build status (success/failure) and the associated build log. CI builds are highly heterogeneous: they come from more than 13,000 CI jobs from more than 30 CI servers. More than 370 development teams are involved in over 2700 code repositories used for these builds. Different languages such as C, C++, Python, Java, Go, and Rust are present in the code repositories, with each language having its own tools suite for CI (e.g., package manager, test framework, code scanners). The average number of lines is 606 lines (standard deviation of 253 lines). We kept only logs with less than 1,000 lines for performance reasons, representing more than 90% of total logs. The distribution of log size (in lines) is shown in Figure 6.4. Continuous integration is managed using Jenkins [93], an open-source CI orchestrator. CI can use a library of ready-to-use CI steps (e.g., code scanning, code linters) or custom code written in Groovy (e.g., shell commands, conditional branches). While other metadata are retrieved during the process (e.g., project name, build triggers, steps of the build), only build status and build log will be used for training the algorithm. We only kept builds triggered by branch events (e.g., commit, new branch) and by the user (manual action). We exclude builds started by timers (i.e., scheduled builds). Scheduled jobs are started significantly more frequently than jobs triggered by code change or user and are therefore over-represented in the dataset. Furthermore, these builds are rarely investigated when they fail, reducing their interest in automated root cause localization.

6.6.1 Synthetic Dataset Using Fault Injection

We built a synthetic dataset of root causes: a dataset where root causes are injected manually inside the build log. First, we have collected a dataset of 65,380 successful builds from the software company CI systems. We kept logs with between 100 and 500 lines to avoid the performance impact of large logs. Indeed, while this is not a fundamental limitation, our implementation requires the sequences (i.e., logs) to be padded to the same size in order to be batched, and batching significantly improves training speed.

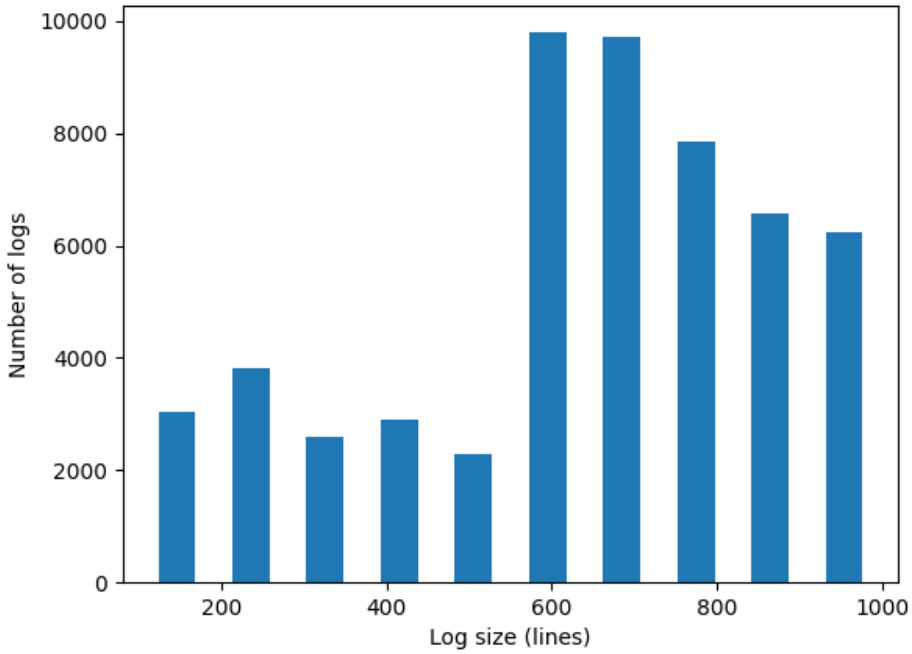

Figure 6.4: Distribution of log sizes for the industrial dataset.

We randomly selected 50% of the builds and set their labels to 0 (i.e., failed builds). In builds marked as failed, we randomly injected a root cause message `THIS IS A ROOT CAUSE` between the middle and the end of the build. We added 10 random characters before and after the root cause message (separated by a space) in the same line to add noise in the root cause line (i.e., not all root cause lines are identical). In other terms, root cause messages are distributed uniformly between line $n/2$ and n where n is the number of lines in the log. Indeed, root cause messages tend to be close to the end of the build log, as the build often exits when a fatal error is encountered. We also save the location of the root cause message in the log in the build metadata. We split the dataset as follows: 90% for training and 10% for testing.

We run experiments on the test split. An example of a belief evolution on a random element of the test set is given in Figure 6.5. Other elements from the test split have a similar belief pattern. The belief evolution is very close to the expected ideal belief pattern. Indeed, the *correctness* score described in Section 6.5.4 has an average of 12.6 (standard deviation 111.8) for test split. Comparatively, the *correctness* score has an average of 423.5 (standard deviation 256.1) before training the algorithm (i.e., randomly initialized weights).

We measure the prediction error on the test split using the final drop detector with a threshold of 0.5. The test set contains 3,279 failed logs. The average log size is 244 lines with a standard deviation equal to 107 lines. The average prediction error is 0.17 lines with a standard

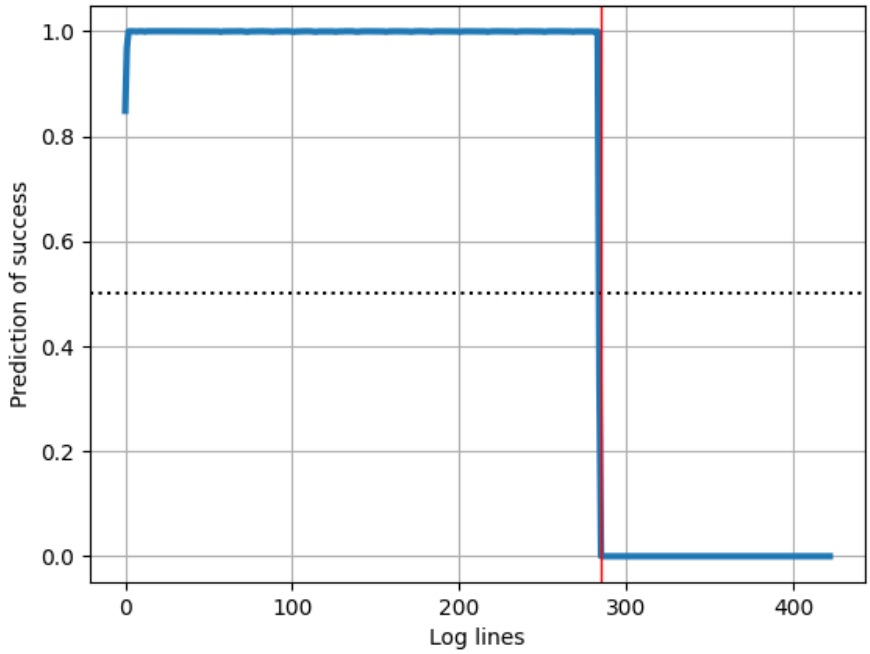

Figure 6.5: Belief evolution of on a synthetic log. The vertical red line represents the location of the real root cause. We observe that the final drop is co-located with the root cause location.

deviation equal to 1.19 lines. 2,844 over 3,279 (87%) were perfectly predicted (i.e., $r_t = r_p$). Such very good results are not surprising, considering the root cause analysis task is simplified compared to real-world failures. Indeed, only one root cause message was injected (although noise was added to the line). Furthermore, all failed logs shared the same root cause, which made training easier. However, the target of this experiment is to show the ability to extract properties of the log using the belief evolution on a small-sized dataset (i.e., thousands of logs). It showed that 5 tokens (`THIS IS A ROOT CAUSE`) randomly inserted into the log can be detected as discriminating patterns. In the following section, we will focus on real-world failures.

Answer to RQ6.1. RNN belief evolution can be used to identify properties of the input log, such as the location of a discriminating pattern. With manually fault-injected logs, we locate the root cause message with an average prediction error of 0.17 lines.

6.6.2 Industrial labeled Dataset

Synthetic datasets allowed us to evaluate ChangeMyMind quantitatively (3,279 labeled builds). In the following section, we evaluate ChangeMyMind on a smaller manually annotated dataset of root causes.

Dataset Construction

We built an equally balanced industrial dataset of 54,808 builds, with 27,404 successful builds and 27,404 failed builds. Considering there are fewer failed builds than successful builds, we did a random sample of successful builds to match the number of failed builds. Finally, we randomly split the dataset into 90% for training (49,327 builds) and 10% for testing (5,481 builds). Both subsets are perfectly balanced (i.e., the exact same number of successful and failed builds). The root cause messages are not labeled in failed logs. Indeed, identifying the root causes of failed logs is a time-consuming process (i.e., a few minutes per build log), and therefore the whole dataset cannot be labeled manually. We randomly extracted and annotated 102 failed build logs from the test split, indicating the localization of the root cause. 2 developers validated root cause annotations. The training process does not require labeled logs. Therefore, they are no interest in labeling logs in the training set.

During our experiments, we experienced what we called the *late decision problem*. It refers to the belief evolution not evolving significantly until the end of the log. Indeed, in the build logs in our dataset, the final line of all logs was either `FINISHED: SUCCESS` or `FINISHED: FAILURE`. This line is added by the CI orchestrator (in our case, Jenkins) for all build logs. However, it negatively impacts the performance of our approach. Indeed, the performance measure of the downstream task (success/failure prediction of the build) refers to the classification accuracy when the entire log is processed. Therefore, the algorithm has no *incentive* to find root cause messages in the log if the final line is sufficient to achieve good performance. Removing the final line is insufficient as the problem occurs with the line before. Indeed, lines before the end of the log are highly discriminating lines (e.g., stack trace). However, they are not relevant to identify the failure's root cause. According to the definition we proposed, the root cause message is the first log message, such as the failure of the build is certain. Therefore, if the belief evolution is impacted only by lines at the end of the log, we cannot localize the root cause message. To mitigate this issue, during the training phase (Section 6.5.2), we split logs into sublogs of k lines. Logs are not split for the analysis phase or hyperparameters optimization. In the following section, we benchmark ChangeMyMind on this dataset. We used the entire dataset for the training phase (as there is no need for root cause labels) and the manually annotated dataset to assess the root cause localization performance.

Baseline Approach

As shown in Section 2.4, many methods in the literature rely on expert knowledge for immediate root cause analysis or to extract features for machine learning algorithms. However, considering the high heterogeneity of our dataset, we cannot use any expert method as it would require the help of a dozen experts in the company. Therefore, we focus on baseline methods that are generic enough to work on the entire dataset. We identified two baseline

Detector	Median	Mean (std)	$pe < 1$	$pe < 5$	$pe < 50$	No result
Final drop 0.5	**4.0**	**62.9 (116.7)**	**5.0**	**54.5**	**70.3**	**1**
Final drop 0.1	5.0	63.6 (116.7)	20.8	50.5	71.3	1
First drop 0.5	36.0	127.1 (174.4)	27.7	37.6	54.5	1
First drop 0.1	27.0	110.8 (158.0)	19.8	40.6	57.4	1
Lowest derivative	176.0	218.2 (177.0)	0.0	8.9	21.8	0
Baseline first error	28.0	123.1 (170.7)	32.7	42.6	56.4	0
baseline final error	240.0	300.5 (245.2)	0.0	0.0	16.8	0

Table 6.1: Mean, standard deviation, and median of prediction error pe for different detectors and baseline on the labeled industrial dataset. We observed that the detector final drop provides the best results for $pe < 5$ with a threshold of 0.5.

methods: the first error line and the final error line. An error line is defined identically as in Section 6.5.2, i.e., if the line contains any word from a list of words related to failure. Results of baseline algorithms are presented in Table 6.1. We observe that the method returning the first error line performs significantly better than the method returning the final error line. This is to be expected, considering failures often cause multiple error messages.

ChangeMyMind Approach

We trained ChangeMyMind on the industrial dataset we collected. The training took 9 hours on a Nvidia RTX 3090. This low training time allows us to regularly retrain the models on recent logs. The instrumentation engine returns a belief of each token. However, as described in Section 6.5.3, operators and detectors expect a belief on a line basis. We explored median, mean, min, and max strategies to assign a belief to a line from the beliefs of its tokens. No statistically significant differences were discovered using different strategies. Therefore, we used the average strategy, i.e., the belief of a line is the average belief of the tokens in the line. Figure 6.6 shows an example of belief evolution for a labeled log. The final drop detector made a good prediction of the root cause location, but the first drop detector was triggered at the beginning of the log. Indeed, on line 12, there is an error message `ERROR: No data found`. While the error was not fatal to the build, it negatively impacted the belief evolution (i.e., drop). To evaluate the performance of detectors on a representative set of logs, we run detectors on the 102 labeled logs. Results of different detectors are presented in Table 6.1. We provided the median, average, and standard deviation of prediction error pe for all logs in the labeled dataset. For one over 102, detectors based on belief drop fail to provide a root cause candidate. Indeed, the belief never crosses the 0.5 threshold for this build. We provide the ratio of logs with a prediction error pe below 1 line (i.e., equal to 0 lines), below 5 lines, and below 50 lines. We observe that the final drop detector with a threshold of 0.5 outperforms other detectors. Indeed, it provides a perfect prediction for 5.0% of the logs. For 54.5%, it provides a prediction error below 5 lines, and for 70.3%, it

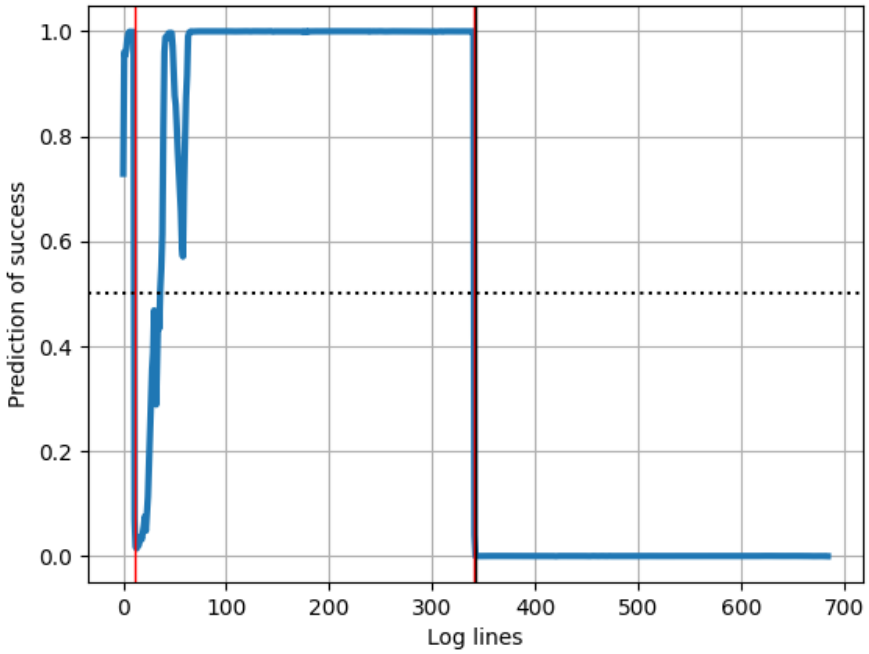

Figure 6.6: Belief evolution of on a real-world log. The vertical red lines represent the root cause candidates given by first and final drop detectors with a threshold of 0.5. We observe that the final drop is co-located with the root cause location. The first drop is triggered by an error log which is not fatal.

provides a prediction error below 50 lines. Compared to the baseline approaches, the operator will have fewer logs to investigate, thus making troubleshooting the build failure faster.

Answer to RQ6.2. Detectors can leverage RNN belief evolution to extract root cause message locations in a real-world dataset of CI/CD build logs. The final drop 0.5 detector outperforms the baseline approach with an average prediction error of 62.9 lines.

6.6.3 Implementation and Performance Impact

We have implemented a proof-of-concept of ChangeMyMind in PyTorch. The implementation reads logs from a JSON-lines file (i.e., one JSON document per line), splits logs, trains the algorithm on the downstream task, and allows running the analysis phase of any log (i.e., including logs not in the dataset). We have also implemented a web application based on the Dash framework [136] depicted in Figure 6.8. It allows operators to easily inspect subparts of the log based on root cause candidates given by detectors or its own interest.

As presented in Section 6.5, ChangeMyMind does not require model modification or any

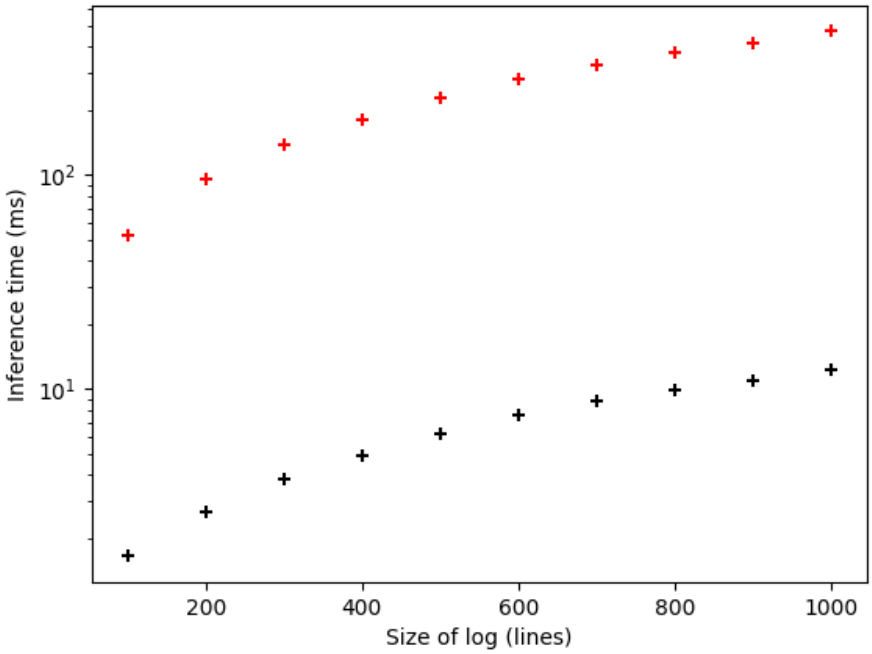

Figure 6.7: Measure of the inference time with belief saving enabled (red) and without (black) for different document sizes.

instrumentation during training. Indeed, belief evolution is only saved during inference. We measured the impact on the inference time on an Nvidia RTX 3090 with Python 3.10. We used the *timeit* Python module to measure inference time with and without belief evolution saving. We made 1000 inferences for each document size (from 100 lines to 1000 lines by step of 100 lines where a line is, on average, 54 characters). The inference time with and without belief saving for different log sizes are shown in Figure 6.7. We observe that saving belief evolution has an impact on inference time. However, inference time stays under 0.5s for documents of 1000 lines. Therefore, our method applies to real-time analysis of build logs for many standard workloads.

6.7 Discussion

ChangeMyMind is a new method that leverages the belief evolution of a neural network to provide root cause analysis. This section discusses the analogy between interpretable AI and root cause analysis. Next, we discuss the properties of failures in continuous integration and the limitations of expert systems.

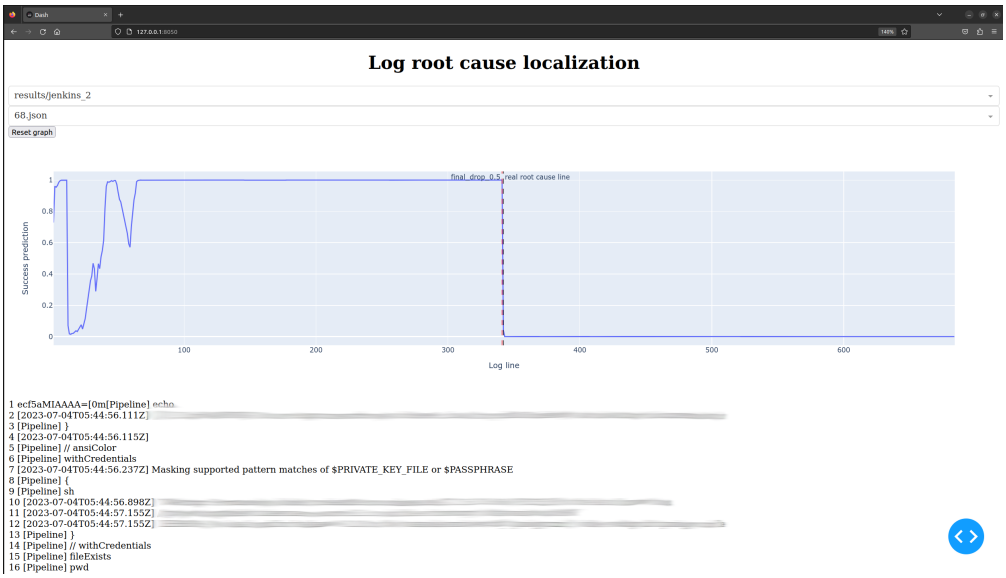

Figure 6.8: Belief evolution showed using the web interface. The vertical black line represents the real root cause (this is a labeled log), and the vertical red line represents the root cause predicted by the final drop 0.5 detector.

6.7.1 Prediction Interpretability and Root Cause Analysis

Artificial intelligence interpretability enables the operator to understand the prediction made by AI models. We observe an analogy with root cause analysis: explaining why an algorithm predicted a build log as "failed" should help to understand why the build has failed. Therefore, we posit that existing approaches [61] for AI interpretability can be applied for root cause analysis. Similarly to ChangeMyMind approach, after training an algorithm to distinguish successful builds from failed builds, interpretability methods can allow the extraction of properties of the build, such as the root cause. Root cause analysis of CI builds is challenging as CI systems are often complex and depend on other systems.

6.7.2 Failure Patterns in Continuous Integration

Continuous integration builds are executed very often during the software development cycle. In large companies, tens of thousands of builds can be executed daily. A part of these builds will fail, up to thousands every day. There are two categories of failure: failures related to software development (legitimate failures) and failures related to the infrastructure (illegitimate failure). Legitimate failures allow the developer to be quickly alerted (shift-left approach) when a code change raises an issue (e.g., code does not compile anymore). However, illegitimate failures are often non-deterministic (e.g., network failure, unavailable remote service). Furthermore, such failures can be caused by partial outages, e.g., a remote service fails to reply correctly to a small part (e.g., 1%) of the total requests. Therefore, a build might fail because a remote

service was not responding correctly, even if the infrastructure monitoring for this service is not logging an ongoing incident.

In the industrial dataset, there are, on average, 20 lines with error terms in failed logs. Therefore, the challenging part of root cause analysis is identifying which error leads to the failure of the build. Indeed, a root cause message might be fatal in one context but not another. For instance, download failure is not fatal if it succeeds after a retry. Failed exit code can be suppressed (e.g., *docker –bad-arg | true*), leading to the exact error message that can be considered fatal in another context.

Automated root cause analysis systems are impacted by error messages that are not fatal to the build. One of the possible use cases of ChangeMyMind can be to extract a subset of the log where only the fatal error is present, thus reducing unwanted noise from non-fatal error messages for building classification algorithms.

6.7.3 Expert Systems for Root Cause Analysis

Commercial software (e.g., Splunk [158]) offers automated log parsing, but only for a limited set of well-known software (e.g., Apache). For unsupported logs, users have to provide regular expressions to identify log events in their logs. However, maintaining an up-to-date database of regular expressions for log parsing is challenging. As software becomes larger, the number of log templates increases, and thus it often becomes unmanageable to support all log templates. A software update may lead to the addition, modification, or removal of log templates. Therefore regex database has to be maintained to remain efficient. Determining the set of log templates used in a software is not straightforward, even when the software's source code is available [188] - which is not always the case. Therefore, regexes are usually crafted using log messages generated during software execution, which implies that events must occur to be considered (log messages that rarely happen are unlikely to be included). Logs containing lines from different software are harder to parse, considering the increased number of log templates. Finally, considering the large volume of logs, regular expressions must be highly specific to avoid incorrect event identification (e.g., `.*timeout.*` will match in many different contexts). In conclusion, while useful in specific contexts, expert systems fail to scale to thousands of different root cause patterns, such for CI/CD.

6.8 Conclusion

In this chapter, we proposed ChangeMyMind, a new method based on RNN belief evolution, to localize root cause patterns in build logs. We empirically demonstrate that ChangeMyMind allows to localize root cause messages in CI/CD build logs with a prediction error below 5 lines for 54% of the logs and below 50 lines for 71% of the logs. We release artifacts used in

this chapter at https://purl.org/changemymind-artifacts. The code provides the required functions to run the training and analysis phases. It also includes a web application to easily browse logs and their belief evolution, including results from different detectors. While we are not allowed to release the industrial datasets we used in this chapter, ChangeMyMind is generic and thus can be applied to any CI/CD build logs for open-source projects or the industry. Furthermore, unlike supervised learning, it does not require a dataset of root cause patterns, which significantly simplifies practical use. Indeed, root cause patterns will be learned automatically from successful and failed logs during training.

Considering the approach does not require prior knowledge of the logs, future works can apply the approach to different logs labeled as successful or failed, such as test logs, transaction logs, or workflow logs. Rotten green tests [35], which are tests marked as successful but that did not work as expected (e.g., the test should have failed), can lower the trust in test campaigns. Furthermore, rotten green tests are hard to detect as they do not directly indicate a failure. Future work on ChangeMyMind could focus on detecting such failures by leveraging belief evolution.

6.9 Summary

We propose ChangeMyMind, a new approach to localize root cause messages in logs. Our system does not require a labeled dataset of root cause messages, making it straightforward to apply. We also do not need to parse and interpret logs, making our approach generic to any system that outputs successful/failed logs (e.g., test logs, transaction logs). First, the algorithm is trained to classify successful logs from failed logs as a downstream task. The algorithm will identify patterns that lead to failure, namely root cause messages. Second, in the analysis phase, our approach leverages the belief evolution of the recurrent neural network (RNN) algorithm during the inference phase (prediction). We propose different heuristics to extract the exact location of the root cause message in the log using this belief evolution of the RNN. We evaluate our approach on a synthetic dataset and real build logs that are manually labeled. We empirically demonstrate that ChangeMyMind allows localization of root cause messages in CI/CD build logs with a prediction error below 5 lines for 54% of the logs and below 50 lines for 71% of the logs. Therefore, it significantly reduces the number of lines to be investigated by operators on failure, thus making build failure troubleshooting easier and faster.

Chapter 7

Security of CI/CD pipelines

7.1 Introduction

Nowadays, CI/CD orchestrators make extensive use of containers. Indeed, containers allow identical environments between a developer laptop, a CI build environment, and the production servers. When a CI build starts (e.g., on code change), the orchestrator starts a new container. The container is based on a container image called the CI build image. This image is generally common to all builds of a specific family (e.g., Java-based software, Python-based software). It must contain all the tools required to perform the CI steps: compilers, linters, code analyzers, test frameworks, company-specific tools, etc. Therefore, it is common for developers to customize this image to include the necessary tools. This means that the construction of the CI image is considered another software; therefore, it is often built using the CI system itself. As a consequence, CI systems are considered self-hosted architectures [182]. Thompson [169] showed that self-hosted architectures raise new questions about security. This is particularly the case with a C compiler written in C. Thompson shows that detecting such malware is challenging if malware is present in the compiler and re-injects itself when it is built. Therefore, to what extent should you trust that your system is malware-free? We show that the same reasoning can be applied to continuous integration systems. In particular, we make the following contributions:

- We show that a CI system can backdoor production software without leaving any trace in the source code repository
- We show that malware can persist on updates in a self-hosted CI system
- We show that malware in a CI system can be updated using hidden channels
- We implement a proof of concept implementation targeting Docker-based CI systems

The chapter is structured as follows. In Section 7.2, we show how to achieve a CI malware

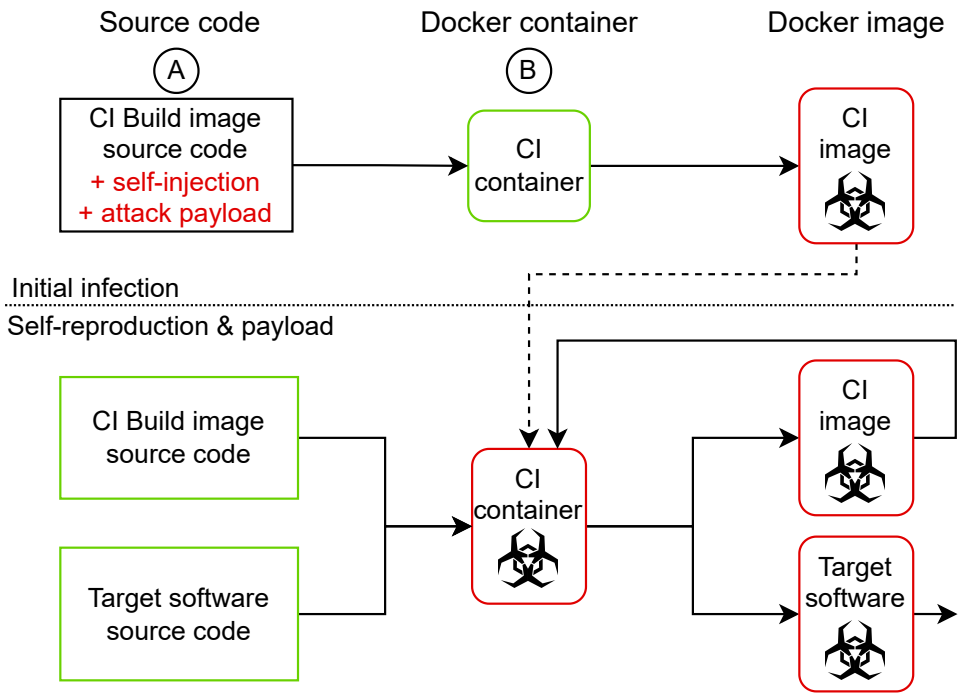

Figure 7.1: Overview of initial infection and after infection modes (self-reproduction and payload). The source code free of malware is in green. The infected containers and images are in red. (A) refers to initial infection using hypocrite commits, and (B) refers to initial infection using an in-container vulnerability (e.g., dependency confusion).

without any trace in the source code. In Section 7.3, we detail a proof of concept implementation targeting Docker. In Section 7.4, we explore the limits of the approach and mitigation strategies. In Section 7.5, we introduce the artifacts that facilitate the reproduction of our findings. In Section 7.6, we conclude and show future improvement paths.

7.2 Our approach

We propose a new approach to compromise continuous integration systems. The approach does not leave any traces in source code repositories. Therefore, the compromise cannot be detected by auditing code repositories, a widely used way to assess software security. Figure 7.1 shows a general overview of the approach. Our approach requires that at least one component of the CI system use a self-hosted architecture. The following section details how we identify the best location for the malware. Then, we detail the self-injection process to persist on CI updates. We present possible strategies for the initial infection process. Finally, we present a few possible attack payloads and how the malware can be remotely controlled using hidden channels.

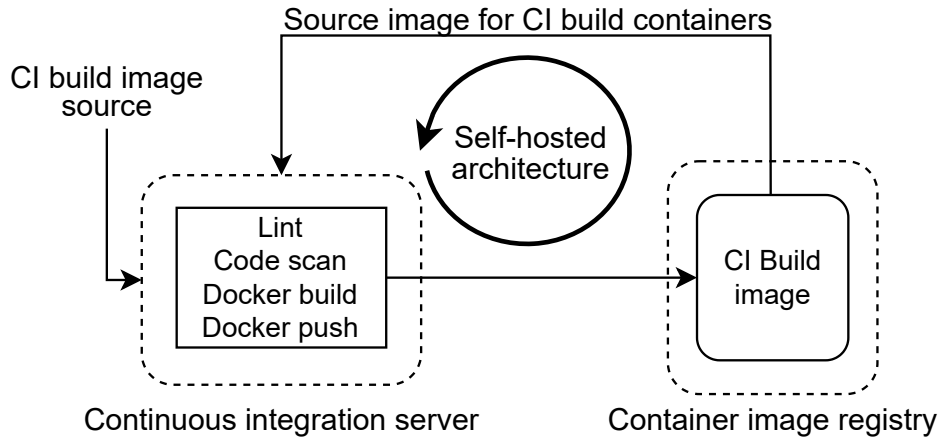

Figure 7.2: Overview of a Continuous Integration self-hosted architecture. A CI container is created from the CI build image, and the CI build image is built in the CI container.

7.2.1 Self-hosted architecture

We propose malware that does not leave any trace in source code repositories. Therefore, this excludes storing the malicious code in the CI system code repository. Furthermore, the malicious code must be stored in a component (e.g., Docker client, Docker daemon, shell, test framework) that is in a self-hosted architecture. Without self-hosted architecture, the malware would be overwritten by a clean component version (i.e., build from repository source code free of malware) on CI updates. Custom build images are commonly used to avoid reinstalling all the necessary tools (e.g., linters) each time the CI build is started, leading to a self-hosted architecture. As CI updates might happen often (e.g., new features, bug fixes, upgrade tool versions), targeting a component in a self-hosted architecture is required for long-term compromise. Figure 7.2 shows the workflow for building the CI build image. It shows that the CI build image is built from the CI build container. Furthermore, the CI build container is created from the CI build image. Therefore, this is a self-hosted architecture for components inside the build container. However, some components might be external to the CI container. Figure 7.3 shows different strategies for building Docker images in a continuous integration build container. When building a Docker image using Docker client (1) or in-container tool (2) such as Kaniko [70], this is a self-hosted architecture. However, this is not a self-hosted architecture if the building is done by an external component such as host Docker daemon, Docker-in-Docker, or distributed system. In practice, Docker-in-Docker is a very common architecture pattern for CI as it helps to target a full containerized architecture. Using the host Docker daemon is a bad practice as it gives root access to the host system from a CI build container [167]. External systems (e.g., distributed build systems) are tailored for specific needs, such as very large build volumes, and are rarely used. Therefore, to support Docker-in-Docker (dind) architecture, we must implement the malware in a self-hosted component.

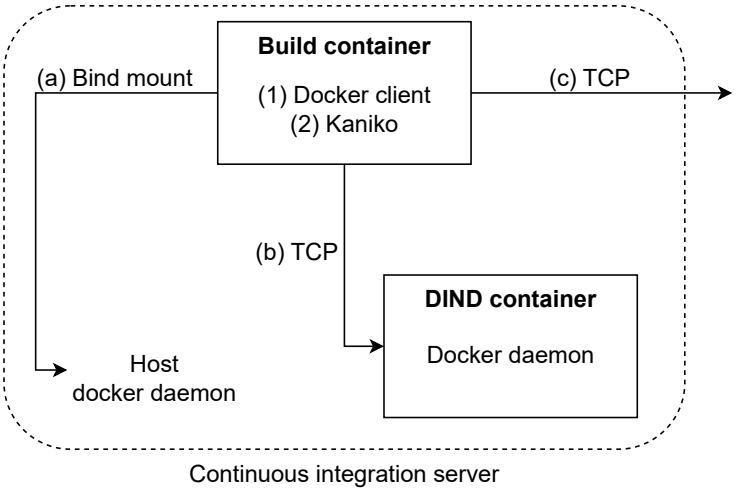

Figure 7.3: Overview of different strategies for building Docker images from a CI container. Docker client (1) or in-container build tool (2), such as Kaniko, can be used. The Docker client connects either to the host Docker daemon (a), a daemon in another container, i.e., Docker-in-Docker (b), or an external daemon over the network (c).

It means targeting software that is stored in the CI build container. This is the case with the Docker client and in-container build tools such as Kaniko. Docker client is a Command Line Interface (CLI) tool that interacts with the Docker daemon, whether the daemon is in another container (dind), on the host system over a bind mount, or in an external system. Therefore, we choose to implement the malware in the Docker client instead of the Docker daemon. The same approach can be applied to Kaniko. However, as the Docker client is more widely used than Kaniko, we focus on the Docker client. Indeed, the Docker client interacts with the Docker daemon to send build context (i.e., Dockerfile, source files, parameters) to the daemon. In the following section, we detail how the Docker client can re-inject the malware when building the CI image that contains the Docker client.

7.2.2 Reproduction mechanism

When building a Docker image in the CI, the malware must be able to re-inject itself when a new CI build image is built: this is the self-reproduction mechanism. As presented in the previous section, we choose to hide the malware in the Docker client as it is present in the CI build image. It can be installed manually by developers (i.e., through a package manager or rarely by compilation) or taken from a base Docker image (e.g., `docker:latest` image). The Docker client should detect if the image being built is the CI build image or an image of interest for the attack payload. We detail in Section 7.2.4 potential attack payload actions. The Docker client might rely on the image name or build context heuristics to identify the image being built. As the Docker client has access to the build context, it can check for the

presence of specific files in the build context (i.e., source code). For instance, it might check for the presence of the program `/usr/bin/docker`. We can re-inject the malware in any Docker image containing the client. However, to reduce the detection risk, we would rather try to re-inject the malware only when the CI image is built. For this purpose, we rely on image names and other rules, such as filenames that only appear in CI build images (e.g., test framework, Docker client). Before being able to self-reproduce, an initial infection must be performed.

7.2.3 Initial infection

The initial infection requires modifying the CI build image. Executing arbitrary code can accomplish this during image build or by directly altering the CI image in the registry. Below, we detail different strategies that internal or external attackers can apply. Therefore, our approach does not require a complicit actor (e.g., a company employee). Initial infection should only be done once. Then, self-reproduction will ensure the malware stays in the CI systems even on updates.

Malicious commit The most straightforward method to execute attacker-controlled code during CI image build is to insert malicious code in the source code repository that defines the CI system. Qiushi et al. [185] show that patch proposals injecting new vulnerabilities can be accepted in large open-source software projects like the Linux kernel. Therefore, the same method can be considered in a corporate environment. However, directly committing malicious code to the repository might be challenging when enforcing peer review and code scanning. To reduce the risk of detection, malicious code can be hidden by proposing a code change that looks legitimate (e.g., adding a new code linter in CI) and relies on a remote compromised package. However, the malicious commits approach requires access to a code management system (i.e., internal network access, credentials), which might be challenging for an external attacker. Another option is to leverage a vulnerability in dependencies management.

Compromised dependency Builds often rely on a large number of dependencies. These dependencies can be internal to the company or public (e.g., open-source). Public dependencies can be leveraged to execute malicious code without committing to the target repository source code. Indeed, dependency confusion [14] allows running attacker-controlled code in a continuous integration build server. The approach leverages how package managers (e.g., pip, npm) handle multiple registries. By default, they look up all configured registries and select the package with the highest version. Furthermore, public registries are commonly enabled by default (e.g., PyPI for Python pip). Therefore, creating a package on a public registry with the

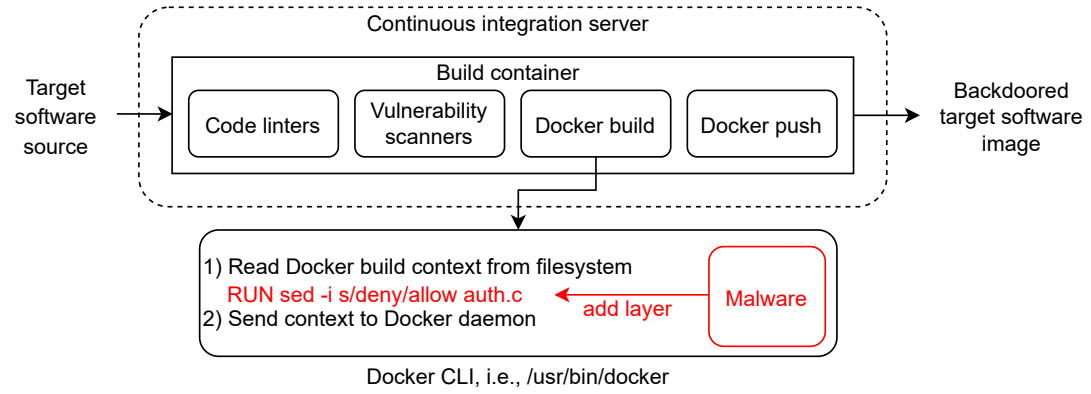

Figure 7.4: Overview of on-the-fly injection of malicious code during Docker build. The malware code is inside the Docker client in red. Backdoor code is injected in the build context before sending it to the Docker daemon.

same name as an internal package and a higher version leads to dependency confusion. JFrog Artifactory, a widely used software to host internal artifacts, was vulnerable to dependency confusion. While attempts [147, 163] have been made to mitigate these attacks, they are difficult to deploy at scale (e.g., not backward-compatible). Therefore, dependency confusion is still applicable to a wide range of systems.

Image registry compromising Another way to compromise a CI image is to directly alter the registry image. Every CI build runs with a user that has privileges. For the build to succeed, these privileges are required: pull source code, download dependencies, or push artifacts. However, a poorly configured CI system might allow CI users to access or modify resources out of the scope of the current build. For instance, the CI users might have full read/write access to the container image registry, allowing any build to alter any stored image. This can happen when a unique user is shared across builds or privileges are assigned to the CI server (e.g., EC2 IAM policy [9], IP-based permission). Credentials might be wrongly populated [88] to pull requests (PRs). Furthermore, opening PRs is often allowed for a large group of users. This might lead to the compromise of registry credentials, allowing manipulation of stored images.

7.2.4 Payload

Malware has a payload, i.e., a part of code that can take action. For malware targeting desktops, the payload often allows monitoring of the user: recording keyboard keys, listening to ambient sound, copying files, etc. In a CI build environment, malware can access and modify sensitive data. Indeed, CI is often used to build artifacts that will be deployed to production. It aims to provide a more secure solution than developers' laptops. An attack payload might record and exfiltrate sensitive data such as environment variables or program source codes. Furthermore,

the payload can alter the source code before compilation. In practice, it might create vulnerabilities in production software, such as authentication bypass. Finally, the payload can consist of network analysis and lateral movement over the network. Indeed, CI servers are often in the internal network perimeter. The attack's payload depends on the attacker's motivation and can change over time. To this aim, we present strategies for remotely updating the malware in the following section.

7.2.5 Command & Control

After initial infection, malware might require to be updated. Indeed, target programs evolve (e.g., new code), or new security systems can be deployed. To this end, we developed a methodology for Command and Control (C&C) for the malware. C&C aims to be less detectable than reusing the initial infection workflow. In the context of a CI server, downloading packages from public registries (e.g., PyPI, npm) is common. Therefore, we can leverage this property to C&C the malware. By uploading a package to a public registry that the malware will download, we can C&C the malware. To further reduce the risk of being detected, the package should look like a standard package (e.g., code linter). However, some files inside the package will contain hidden material using steganography. Therefore, detecting that downloaded packages contain malicious code will be challenging. Other notable options for C&C include DNS tunneling [174] or git-based tunneling. Encrypted files in git (e.g., SOPS [121]) are widely used to store and share secrets required for software development securely. However, they create a blind spot for code scanners as they do not have encryption keys. This can be leveraged to hide malicious code behind encryption. Encrypted files will be decrypted by malware in the CI to update themselves. In practice, C&C will highly depend on the victim's infrastructure and security mechanisms. The next section details how we created a proof of concept implementation targeting Docker-based continuous integration systems.

7.3 Implementation

To demonstrate practical self-replicating and invisible malware in CI systems, we wrote a proof of concept implementation targeting the Docker client [40]. The latest version of the artifact is available at https://doi.org/10.5281/zenodo.7777331.

Docker Engine follows a client-server architecture consisting of a client communicating with a Docker daemon, acting as the server, using the HTTP or HTTPS protocol over TCP or a socket. The daemon creates and manages Docker objects, such as images and containers, whereas the client sends commands to the daemon and shows output (e.g., logs) to the user. When the client instructs the daemon to build a new image, the client sends a file consisting of build commands to assemble an image (referred to as a "Dockerfile"), along with any other required

files (referred to as "context"). As detailed in Section 7.2.1, we target the Docker client.

7.3.1 Bootstrapping

In the context of CI systems, the Docker client runs within the CI container: this is a self-hosted architecture. This makes it an apt target for creating self-replicating malware; if the client rebuilds itself, it can be patched with a self-replicating procedure like Thompson's [169]. Patching the client requires a bootstrapping [179] phase before the initial infection of any CI system. The bootstrapping phase is performed by a custom script that produces a Dockerfile containing build commands to produce a client capable of self-replication.

Achieving self-replication To achieve self-replication, two separate files are needed: (1) a Dockerfile with build commands to compile a Docker client (referred to as the "genesis Dockerfile" to avoid confusion), and (2) a patch of the Docker client source code. The build commands in the genesis Dockerfile must contain all required material to build a Docker client. The patch of the Docker client source code must include a procedure that detects when a target Dockerfile sent to the daemon (1) uses a base image containing a Docker client or (2) when a Docker client is assembled from scratch. Notably, the original (i.e., non-malicious) Docker client can come from the Docker base image, the package manager (e.g., apt), or built from source. When the procedure detects one of these scenarios, it applies the Docker client patch to the client version compiled in the build commands of the genesis Dockerfile. Afterward, the procedure must replace the build commands in the target Dockerfile corresponding to the Docker client (base image or assembly) with the commands from the genesis Dockerfile. This procedure is based on the self-replication logic of a quine [170]. We choose to compile the malicious Docker client in the build container to avoid fetching a malicious Docker client over the network (intrusion detection systems could detect that). However, it is also possible to fetch the malicious client from a remote registry to avoid the complexity of compiling it during initial infection or self-reproduction.

Performing bootstrapping The custom script that performs bootstrapping produces a Dockerfile with self-replicating logic based on the files introduced in the previous paragraph. The CI can be considered infected once this Dockerfile is used to assemble a CI image. Every rebuild of a similar image will then re-inject the self-replicating client patch. As illustrated in Figure 7.5, the custom script performs the following steps to create the aforementioned Dockerfile:

1. Insert the original genesis Dockerfile as a payload within the client patch.
2. Insert this client patch as a payload into itself.

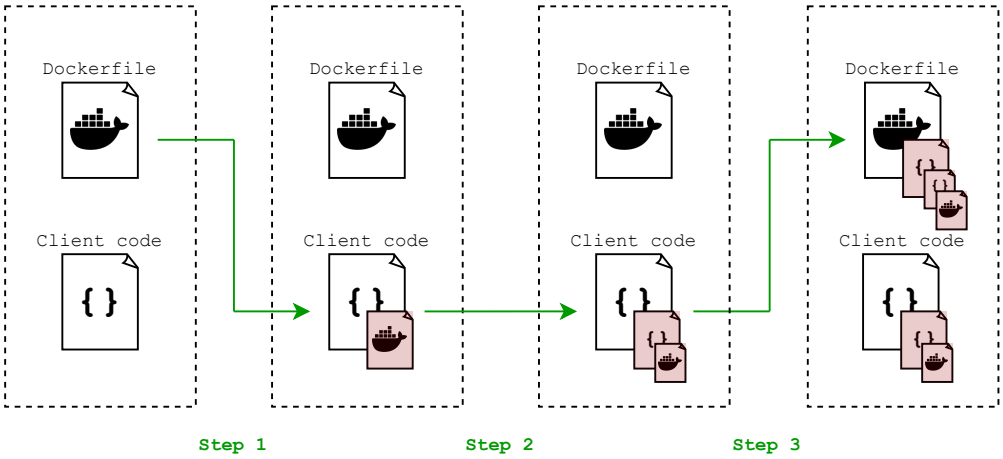

Figure 7.5: Illustration of bootstrapping flow. Green represents a step taken, while red represents a payload within a program. The leftmost box consists of a genesis Dockerfile and patches client code. Step 1 inserts the genesis Dockerfile into the patched client code as a payload. In step 2, the patched client code is inserted into itself as a payload. Finally, the patched code is inserted into the genesis Dockerfile in step 3. The final Dockerfile and client code versions now contain enough information to perform self-replication.

3. Insert this final patch version as a payload into the original genesis Dockerfile.

When this Dockerfile is used in real CI systems, the client patch in the payload will be used to compile the next Docker client. This new client will then have all the necessary material to rebuild itself by performing steps 1 and 2.Combined with the steps outlined in the previous paragraph, we have a fully self-contained and self-replicating version of the original Docker client. Thus, when a CI image is built using this new client, the CI image will be reinfected.

7.3.2 Initial infection

Initial infection is the first time our implementation is injected into a CI system. We tested our approach against a GitLab instance and a Docker-based GitLab runner. The initial infection is done in 2 steps: (*a*) altering a target Dockerfile and (*b*) assembling a CI image from the Dockerfile in the CI.

Altering a Dockerfile The first step of the initial infection is to alter a target Dockerfile. A target Dockerfile is any Dockerfile that would result in the build of a CI image. The alteration includes modifying the target Dockerfile by replacing the base image or creating a Docker client with the build commands from a bootstrapped Dockerfile. This corresponds to (A) in Figure 7.1. This step can be achieved by any approach mentioned in Section 7.2.3. Any approach that injects the build commands from a bootstrapped Dockerfile will result in the

build of an infected CI image.

Assembling the CI image Once the build commands are injected into a target Dockerfile, a new CI image has to be assembled based on the Dockerfile. In the initial infection, a clean CI container will assemble the new, malicious CI image corresponding to (B) in Figure 7.1. The very first build of the malicious CI image may be detected, as the target Dockerfile must contain the build commands from the bootstrapped Dockerfile. Additionally, the clean CI image will report all build commands performed in the image build as logs. This is the default behavior of Docker clients, as the logs support root cause analysis on failure. Hence, the initial infection may be revealed unless the target Dockerfile alteration is removed and logs from the clean CI container are concealed.

Future builds of the CI image Once the initial infection has been successfully performed and the malicious CI image exists in the CI system, any consecutive build of CI images will result in malicious CI images. Thanks to the self-replicating design of the Docker client patch, assembling a CI image will always result in a malicious image, regardless of whether the incoming Dockerfile includes the bootstrapping steps. This eliminates the need to alter a target Dockerfile anew; no changes to a target Dockerfile are required and do not need to exist in the source code or code repositories. The Docker client patch masks any output corresponding to build commands from the bootstrapped Dockerfile. Thus, the malicious CI container will not output any steps taken in self-replication, minimizing the chance of detection.

7.3.3 Attacking target software

The Docker client patch introduced in Section 7.3.1 includes, in addition to its self-replicating code, any extraneous payload, such as malware. As this payload is packaged with the malicious CI image, it can perform any action the CI container is authorized to perform. Moreover, as the CI container may be used to build other software, the payload can be targeted toward this build step. This gives a potential malicious actor a broad attack surface towards any software.

A custom attack was created for a toy project to test this capability. The toy project is a C program that checks for credentials a user provides, granting access if the credentials are correct. Moreover, the project is a compiled program within a Docker container. The compilation step is performed as the project image is assembled. The custom attack targets this compilation step during the image build.

When the Docker client within the malicious CI image recognizes that an image for this toy project is being assembled, it modifies the compilation build command. The modification

instructs the Docker daemon first to replace code in the toy project and then compile it, similar to what Figure 7.4 depicts. The replacement disables the credentials correctness check in the toy project, granting access upon any input. This capability is enabled by the client being allowed to read the Dockerfile and context of the toy project. In this case, the client recognizes the toy project by reading the context tags provided by a user.

7.3.4 Challenges

Many challenges were faced during the implementation of our proof-of-concept. Challenges were related to the malware's design, tooling constraints, and design concealment of the initial infection.

Self-replication Conceptualizing self-replication in the context of Docker is the first challenge of the implementation. Whereas the self-replication presented in [169] only requires the compiler source code to be changed, infecting a CI system requires modifying a Dockerfile and patching the Docker client source code. As Figure 7.5 illustrates, the Dockerfile and the client patch contain multiple payload levels. Each level is required to achieve self-replication; every level is required such that the client patch can behave as a quine, outputting its source code dynamically and re-injecting the output into a CI image.

Payload encoding Dockerfiles have a custom domain-specific language for building commands [41]. The `RUN` command executes listed arguments in a shell. When bootstrapping the genesis Dockerfile in Section 7.3.1, the Docker client patch is inserted into the genesis Dockerfile as a payload. The payload is used to modify the source code of the Docker client before compiling it, where the client is written in the Go programming language [40]. This modification step uses a `RUN` command and the client patch. This may result in a clash in syntax as both shell and Go syntax may have overlaps (such as the escape character \ for strings). To overcome this issue and any other potential syntax clashes, all payloads are base64 encoded before any insertion step in Figure 7.5.

Dockerfile line length Lines in Dockerfiles are restricted to 65535 characters, restricting the length of build commands. This length restriction was discovered empirically. As the bootstrapped Docker client patch exceeds 60000 characters, it will exceed the line length restriction once the payload is base64 encoded. All payloads were compressed with Gzip to overcome this restriction before being base64 encoded. As an additional side-effect, the compression aids in minimizing the footprint of the malware.

Log filtering Another challenge in the implementation was the masking of output logs. As the daemon executes build commands, they are logged with a prefix denoting the command's execution number. The logs are produced in the following manner, assuming three commands:

```
Step (1/3) <output of command1>
Step (2/3) <output of command2>
Step (3/3) <output of command3>
```

Build commands injected from the bootstrapped Dockerfile would be logged along with commands from the target Dockerfile. This poses a challenge in achieving invisibility as logs leave a visible trace. Although the daemon produces the logs, they are outputted by the client. Overcoming this challenge thus required the custom client to parse received logs and deduce which build commands to mask. For this purpose, we created a procedure for parsing the infected target Dockerfile before passing it to the daemon. The procedure counts the number of commands to be executed and then subtracts the number of commands passed by the bootstrapped Dockerfile. Before outputting the final logs, each log line containing the `Step` prefix was manipulated to show a subtracted amount. Re-using the example above and assuming that the line containing `<command2>` is part of the bootstrapped Dockerfile, the procedure produced the following output:

```
Step (1/2) <output of command1>
Step (2/2) <output of command3>
```

This enhanced the invisibility of self-replication and lowered the probability of discovery.

Custom attacks Attacks must be tailored for the target environment before attaching them as payloads to the source code in (A) of Figure 7.1. This requires pre-existing knowledge of the target environment to infect target software successfully. This may pose a challenge to an attacker without prior knowledge of the said environment, as an infection may fail and potentially reveal itself. An attacker may consider targeting a long-running container by inserting a backdoor. This would allow the attacker to enter the target environment and explore it further before creating a targeted attack. However, this would still challenge the attacker to know which container to target with a backdoor, requiring prior knowledge of the environment. Nevertheless, implementing a C&C system would allow the attacker to update the malicious content incrementally.

7.3.5 Limitations

Our proof of concept has a few limitations that might be resolved with enough engineering effort. We detect a CI image by checking the image name. However, the image name can be changed by developers. Detecting CI images based on build context would be more resilient. When re-injecting, the Docker client is compiled. This takes time, which increases the risk of being detected even if the logs are hidden. The patch for the Docker client targets a specific version of the client. While the code we rely on for the patch should be quite stable, we do not handle Docker client updates.

The proof of concept does not have any update mechanism in place. Thus, any change in the target environment may reveal the malware or self-replication. Nevertheless, as outlined in Section 7.2.5, command and control can be implemented through different means, such as steganography.

No fallback mechanism exists in case of errors caused by the proof of concept. This may prevent the building of CI images or target software, which is detectable by users. A potential mitigation strategy would be to notify a command and control server and request updates. Another strategy would be to revert any attempt at infection to escape detection.

7.4 Discussion

Our approach requires to have a self-hosted architecture to provide a long-term compromise. However, self-hosted architecture can be avoided. For instance, an independent CI/CD system can build images for the main CI/CD system. Or images can be built manually directly on developers' laptops. Furthermore, CI containers can use images built by other actors (e.g., community images). We implement the malware in the Docker client code stored in the CI build image. Rkhunter [139] is an open-source tool that scans for rootkits by comparing SHA-1 hashes of local files with an online database of well-known software. In the context of this work, deploying such a tool for checking binary file integrity in CI images might help to detect such compromise. Similarly, Prisma Cloud [125] provides Docker image scanning capabilities to detect vulnerable software and common misconfiguration. They rely on a database of hashes of malicious software. Custom rules can be developed to detect CI image manipulation. Docker Content Trust [37] (DCT) provides a signature mechanism to authenticate the author of the image and ensure that the image has not been tampered with. However, as images are built and signed by CI, DCT does not help to detect image manipulation by the CI. Indeed, DCT was designed to detect attacks that alter images in the Docker registry. We tested our attack on a GitLab with a self-hosted runner. However, our approach works on major CI providers as far as the self-hosted condition is verified. GitHub Actions, Jenkins, and Travis allow the use of custom images for building containers and do not prevent the use of CI images in the CI itself,

leading to self-hosted architecture. Therefore, they are vulnerable, too.

7.5 Artifact Description

7.5.1 Introduction

Reproducible research aims to enable the replication of experiments presented in research papers. This is a growing trend in many fields of research. The ability to reproduce published results can facilitate future research, such as comparing new methods or identifying implementation issues. To support our contributions, we provide a ready-to-use system to reproduce our results. The provided artifact aims to demonstrate the answer to the following research questions:
RQ7.1: Can a malicious CI image alter on the fly the Docker images being built in the CI?
RQ7.2: Can a malicious code in a CI image reinject itself when CI is built to persist on CI updates?
RQ7.3: Does altering on-the-fly Docker images during CI build execution leave any trace in the CI build log?
To this end, we provide a step-by-step procedure. The latest version of the artifact is available at https://doi.org/10.5281/zenodo.7777331. It contains a `README.md` detailing the instructions to re-run our experiments.

7.5.2 Container orchestration

A full-featured Continuous Integration (CI) system requires multiple components: a CI orchestrator, a CI executor, a Docker daemon, and a Docker registry. We choose GitLab [65] for the proof of concept. GitLab provides an open-source and free CI solution (Community Edition). Furthermore, they provide an all-in-one container (GitLab Omnibus [67]) that greatly simplifies GitLab setup. We deployed a GitLab along with GitLab runner [66]. It listens for CI jobs and executes them in Docker containers (when using Docker executor). It requires access to a Docker daemon. While we can provide access to the host Docker daemon using a bind mount on the Docker socket, we preferred to avoid interacting with the host Docker daemon during the experiment. Therefore, we propose a Docker daemon in a Docker container. This approach is known as Docker-in-Docker [135]. Finally, we deployed a Docker registry [39] to store container images. We used Docker Compose [38] to define and orchestrate these containers. Container configuration (e.g., GitLab initial setup, GitLab Runner registration) is time-consuming and error-prone. Therefore, we provide an archive `volumes.tar.xz` (in the artifacts archive) containing the docker bind mounts' data. It allows starting containers to be configured and contain all required data for experiments. The whole experiment can easily be replayed by following cleanup instructions and starting from scratch.

7.5.3 Initial infection

Before starting experiments, the CI system is free of malware. We provide a bash script that will take care of the initial infection. Thanks to bash scripting, it is straightforward to identify which commands are used. The script will:

- Compile the malicious Docker client using Docker (Docker is used because it is easier to build client source code inside; it is not required by design)
- Copy the malicious Docker client to the CI system git repository
- Alter the Dockerfile
- Commit & push
- Wait 30 seconds to let GitLab start a CI/CD pipeline and pull code
- Revert the modification and force push to remove the malicious commit from the git tree

After the initial infection, the malicious code can be used to alter production images. In the context of this experiment, the production image is an API protected by a secret token defined in the source.

7.5.4 Attack payload

After the initial infection, the malicious Docker client includes an attack payload. The attack payload targets a secured API that requires authentication by providing a secret token defined by an allow list in the API source code. The API is packaged as a Docker image built by the CI/CD system. The payload searches each Dockerfile to be built for a specific string relating to the target API. Once found, the payload adds on-the-fly an attacker-controller token to the allowed tokens list. This does not leave any trace in the source code repository. The attacker-controller token allows the attacker to bypass the authentication mechanism after the image is deployed to production. This answer RQ7.1.

7.5.5 Self-reproduction

The image used by continuous integration containers is often rebuilt to update the programs installed inside (e.g., security scanners, test frameworks). Therefore, considering the malware does not leave any trace in the source code repository, the malicious CI image would be replaced by a clean image on the CI update. However, the CI image is built, as other images in the CI, by a CI container. This container is based on the CI image. Therefore, we implemented

a check to detect whether it is the CI image that is being built. If the malicious Docker client identifies that the CI image is being built, it adds on-the-fly an extra step to the Dockerfile to copy itself (i.e., Docker client binary) to the new Docker image. Furthermore, it adds another step to ensure no other Docker client will take precedence (e.g., `/usr/local/bin` often takes precedence over `/usr/bin`). Therefore, the new Docker image of the CI will contain the (malicious) Docker client present in the CI image. This is a simplified approach (e.g., the Docker client will not be updated to a new version as it is always taken from the CI image). In a real-world scenario, rebuilding the Docker client with a malicious patch might be a better option. This answer RQ7.2.

7.5.6 Log traces

We show that our malicious code can alter production images (i.e., here, an API) and re-inject itself when the CI image is built. However, developers often look at CI build logs, especially when the build has failed. To hide the alteration of production images or the self-reproduction mechanism, the Docker client filters logs from the Docker daemon. It keeps track of the number of extra steps added (e.g., COPY, RUN) and corrects the daemon output on the fly. Indeed, it rewrites the total number of steps to a corrected number of steps (i.e., total minus added steps). It also completely removes logs related to added steps. Therefore, no logs related to the attack payload or the self-reproduction mechanism appear in the build log. This answer RQ7.3.

7.5.7 Discussion

Our proof of concept could be improved with engineering efforts. Currently, the attack payload adds a Docker layer [74] with a `sed` command to modify the source of the target program. Modifying the source code of the target program directly in the build context would be better without altering the Dockerfile. Indeed, adding a command to the Docker image could be detected using `docker history` command. During the initial infection, we pushed a malicious commit and rewrote the git history. However, the CI pipeline that was started is still present in GitLab. Malicious commit is only one option for initial infection. Real-world scenarios would prefer to rely on dependency confusion [14] to run arbitrary code directly in the CI build container. For image identification, we rely on string matching in Dockerfile (e.g., match “CI system” in Dockerfile). Advanced scenarios would prefer image identification based on build context (e.g., presence of specific files) or environment variables populated by GitLab CI (e.g., CI_REPOSITORY_URL).

7.5.8 Summary

This proof of concept was designed to showcase our contributions. Starting from a clean CI system, we propose an automated infection step based on a malicious commit. Then, we show that a malicious CI image can alter on-the-fly production images and re-inject itself when the CI image is built. Finally, we show that a malicious Docker client can hide logs from the Docker daemon to reduce further the likelihood of being detected. We provide an all-in-one portable solution based on Docker Compose to facilitate its use for this experiment and future experiments on CI/CD. This experiment highlights that if a CI system is compromised once (e.g., dependency confusion, malicious commit), it becomes very difficult to ensure the system is completely malware-free. Indeed, the absence of malicious code in the source code repositories does not indicate that the CI system is not compromised.

7.6 Conclusion

We proposed an effective method for long-term compromise of continuous integration systems. Initial infection can be done remotely by leveraging vulnerabilities such as dependency confusion. After the initial infection, no traces are present in the source code repository, greatly reducing the risk of being detected. In addition, C&C can be done through covert channels using public package registries, which allows the attack payload to be updated with a very low risk of detection. Therefore, our approach raises new questions about responding to security incidents involving continuous integration systems: how do we know the system has not been compromised without leaving any trace in the source code?

7.7 Summary

Revisiting the famous compiler backdoor from Ken Thompson, we show that a container-based CI system can be compromised without leaving any trace in the source code. Therefore, detecting such malware is challenging or even impossible with common practices such as peer review or static code analysis. We detail multiple ways to do the initial infection process. Then, we show how to persist during CI system updates, allowing long-term compromise. We detail possible malicious attack payloads such as sensitive data extraction or backdooring production software. We show that infected CI systems can be remotely controlled using covert channels to update attack payload or adapt malware to mitigation strategies. Finally, we propose a proof of concept implementation tested on GitLab CI and applicable to major CI providers.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Continuous Integration/Continuous Deployment (CI/CD) practices have emerged as critical components of the software development lifecycle in modern times. Through this thesis, we explored multiple research fields, ranging from collecting and analyzing CI/CD log data, automating the classification of CI/CD failures, locating root cause messages in logs, and network and CI/CD security.

Our first research question focuses on the learning potential from a large CI/CD build logs corpus, particularly identifying build failure patterns. In Chapter 3, we collected a large corpus of CI/CD executions from an industrial context. We proposed a method to automatically identify failure reasons by leveraging the corrective action made by developers (e.g., code patch) between two subsequent builds. This method allows the labeling of about 10% of historical failed builds. Notably, we identified that many builds are fixed without code change, suggesting some flakiness. Considering the lack of public CI/CD logs datasets, we build a dataset from public code repositories using GitHub Actions. The public dataset comprises 116k Github Action workflows spread into 28k repositories across 20 programming languages. Then, we analyze pipeline steps defined in these workflows. We find out that even in Github Actions provide reusable actions, 56% of workflows' steps are shell commands, and they account for 82% of the execution time. Therefore, this encourages studies on CI/CD pipelines to focus on shell steps, which are often ignored in the literature.

Our second research question focuses on the automated classification of CI/CD build failures. Chapter 4 explores the use of Natural Language Processing techniques on build logs for classifying build failures between non-deterministic failures (e.g., instabilities) and deterministic failures (e.g., code mistakes). We obtained a classification accuracy of 78% on a balanced industrial dataset. Considering that the build logs represent only a proportion of overall build data, we attempted the same classification task using knowledge graphs. First, we proposed

an ontology to make the semantics of the CI/CD ecosystem explicit. Then, we use Knowledge Graph Embedding techniques to do link predicting, which is equivalent to classifying build failures. This approach achieves an accuracy of 94% on a balanced industrial dataset, thus outperforming the natural processing approach. This work leads to a publication [119] to showcase the use of knowledge graphs for the classification of CI/CD builds.

CI/CD builds highly interact with their environment, especially through the network. As such, analyzing network interactions can help identify reasons for failure. However, network traffic is almost completely TLS-encrypted, and existing methods for TLS inspection are difficult to use in CI/CD environments, for instance, because of the high versatility of software running in the pipelines. Chapter 5 investigates strategies for transparent and universal decryption of TLS-encrypted network traffic with an application to CI/CD build environments. This work leads to a publication [117] to describe X-Ray-TLS, a new method based on memory introspection for transparently decrypting TLS traffic.

Unlike classifying builds, we explore how to define and locate root cause messages in CI/CD build logs. Indeed, predicting root cause location is another method that enables easier troubleshooting of build failures. In Chapter 6, we propose ChangeMyMind, a method to locate root cause messages in CI/CD build logs. First, we propose a definition of root cause messages. Then, we trained a neural network algorithm to classify successful and failed logs. Finally, by monitoring the inference phase of the neural network, we can identify the area of the build logs that likely contain the root cause message. The method does not require a labeled dataset of root cause messages, which is often difficult to build in practice, making a major change from existing approaches that rely on existing knowledge about failure patterns. We empirically demonstrate that ChangeMyMind allows localization of root cause messages in CI/CD build logs with a prediction error below 5 lines for 54% of the logs and below 50 lines for 71% of the logs. Therefore, it significantly reduces the number of lines to be investigated by operators on failure, thus making build failure troubleshooting easier and faster.

Build logs provide insights into processes and actions in CI/CD pipelines. However, are logs sufficient to detect CI/CD security compromises? In Chapter 7, we explored the security properties of self-hosted CI/CD systems. Revisiting the famous compiler backdoor from Ken Thompson [169], we show that a container-based CI/CD system can be compromised in the long term without leaving any trace in source code repositories or build logs. Therefore, detecting such malware is challenging or impossible with common practices like peer review, static code analysis, or analyzing build logs. This work leads to a publication [118] that showcases a proof of concept targeting Gitlab CI.

This thesis aims to make a step toward improving the automated analysis of CI/CD failures by categorizing builds or locating root cause messages. The next section discusses opportunities for future work.

8.2 Future work

This section explores potential research avenues that could significantly benefit the industrial and software development communities as a follow-up to the work presented in this thesis.

In Chapter 3, we empirically demonstrated that 56 % of GitHub Actions workflow steps are shell steps despite the availability of Actions. We postulate that further research is needed to understand better why developers use shell steps instead of Actions. Comparing Actions and shell steps requires estimating whether they provide the same functionality, e.g., setting up a build environment. However, it can be challenging as Actions can use virtually any language thanks to containerization, compared to shell steps that use shell code. Finally, identifying actions and shell steps with the same functionality will allow for comparisons, such as execution time or failure rate. Furthermore, while Actions provide benefits, new questions arise about their reliability and security. Actions, as running in CI/CD pipelines, are sometimes privileged (e.g., release branch). Therefore, they can access or modify non-public or confidential data (e.g., secret credentials, closed-source software code). For community-driven Actions, many practitioners can contribute to the source code of the Action, raising questions about the safeguards in place to prevent the integration of malicious code. Furthermore, Actions can have many dependencies (e.g., an Action can depend on hundreds of npm packages [34]). Therefore, an Action can run malicious code if one of its dependencies is vulnerable, making the security assessment much harder than analyzing the Action code. This raises concerns about software supply chain security and possible remediation.

We showed that 82.6 % of the execution time is spent in shell commands. Therefore, time optimization of workflows requires focusing more on shell steps than Actions. We analyzed execution time variability and postulated high execution variability indicates possible optimization opportunities. Further research is needed to identify which properties impact the execution time precisely. This can include the code base, the configuration of the tooling (e.g., whether caching is enabled), or the selection of tests to be run (e.g., partial test execution based on modified files). Looking ahead, execution time analysis can offer practical suggestions to developers for enhancing their pipelines.

A significant gap exists in the availability of datasets for research on CI/CD. Although there are some large datasets from Travis CI [12, 47], this orchestrator has largely declined recently. We paved the way by providing a dataset of hundreds of thousands of GitHub Actions runs to support large-scale analysis and data-driven approaches. However, the dataset lacks ground truth root causes for failed runs, which hampers the ability to use it as a comparison dataset to benchmark root cause analysis methodologies in an open and reproducible way. We did not label failures in terms of classification (i.e., labels for the failed runs) or localization (i.e., location of the root cause message in the log) as it requires heavy human-in-the-loop effort. However, crowd-based approaches might be considered, such as partnering with GitHub.

This thesis concentrates on the automated analysis of the root causes of CI/CD failures, with an extension on rectifying non-deterministic failures automatically. However, we proposed a simple fixing action only for non-deterministic failures: restarting the build. This approach can be improved to predict when the build should be restarted to have the highest success probability, e.g., after a constant delay or depending on the failure category. Future research could expand the scope to encompass additional failure patterns, such as deterministic failures. For example, failures due to human errors, like code typos or linting issues, can probably be rectified automatically by automated code patch generation. Research topics include building high-quality datasets of failed CI/CD runs, fixing code patches, and automated generation of corrective code patches. Corrective code patches can be generated iteratively (e.g., until CI/CD pass) guided by failure pattern and test suite analysis. However, addressing certain failure patterns, such as determining whether a failure is due to faulty code or test scripts, remains a complex challenge. For instance, when a test is failing deterministically, is the test code or the source code under test which is faulty? Automatic resolution of test failures requires the machine to comprehend the underlying intent of the code, essentially the specifications, to assess if the test failure is legitimate. Thus, automatically learning program specifications to generate tests represents a potential research direction.

A key enabler for developing automated fix solutions for CI/CD failures is gathering data related to the root cause, targeting high-quality and machine-friendly data. This includes log messages, especially error messages, but other data sources can be considered. For instance, the code change that triggered the run can explain the failure, e.g., if it contains a syntax error. We strongly believe that network insights, such as HTTP code status, are relevant for identifying failure patterns. For example, if an HTTP code above 500 is encountered, the failures are most likely related to the remote server. In Chapter 5, we paved the way for transparent and generic decryption of TLS traffic suitable for CI/CD environments. However, future work is required to assess the benefit of decrypting TLS traffic regarding root cause analysis prediction. Nevertheless, full network observability would allow exploring other benefits, such as automated identification of network dependencies, recording and replaying the run (mocking external interactions), or real-time analysis to detect and mitigate network threats. Although TLS decryption can be done with minimal impact, the deployment might be difficult in some contexts, such as in highly regulated environments where encrypted traffic should not be decrypted. Therefore, it is worth exploring whether network insights can only be generated from encrypted traffic. This would allow the detection of transport failures (e.g., TCP timeout) or predict the HTTP response based on traffic properties, such as response size. In conclusion, future work is required to identify data sources that will help to improve automated root cause analysis.

We discussed the main characteristics of two CI/CD orchestrators, GitHub Actions and Jenkins, as well as failure patterns that are commonly encountered. Furthermore, we proposed an

ontology for Jenkins orchestrator for integrating builds data into knowledge graphs. However, to our knowledge, no ontology or taxonomy that applies to all CI/CD systems exists. Similarly, there is little work for exploring generic failure patterns, i.e., that apply to all CI/CD orchestrators. Therefore, this makes comparisons between CI/CD systems harder. Therefore, we encourage future work to propose a taxonomy to define concepts used in CI/CD pipelines and their usage in different CI/CD orchestrators. Each system has its semantics, although they represent similar actions. For instance, GitHub Actions refers to runs, while Jenkins refers to builds. However, both terms refer to the same concept. This taxonomy can be extended to an ontology to integrate data into knowledge graphs in a generic way, i.e., independently from the CI/CD orchestrator in use.

Fault localization techniques, i.e., predicting the location of the root cause message in logs, assist developers in troubleshooting build failures. However, it is important to measure the impact on troubleshooting time when developers are assisted by fault localization solutions. This would include experiments with human-in-the-loop, such as comparing troubleshooting time for homogeneous groups of developers with and without assisting tools. Furthermore, this could be extended to estimating the impact on the efficiency of software development as a whole, giving decision-makers a clearer picture of the value of these solutions. Build logs usually contain multiple error messages that might have or not have an impact on the failure. This tends to confuse automatic log analysis solutions. Therefore, another use case for fault localization techniques can be to narrow the build log to be analyzed, thus limiting the presence of error messages that are unrelated to the root cause of the failure. In other terms, fault localization can be seen as the first step of a build failure classification pipeline. Finally, we applied ChangeMyMind (Chapter 6) to CI/CD build logs, focusing on identifying root cause messages. However, no prerequisites limit the use of ChangeMyMind to CI/CD logs only. The method applies to any dataset with successful/failed couples that are sequences (e.g., text documents). Therefore, we encourage the use of ChangeMyMind with different data, such as test logs or transactional logs. Furthermore, the approach can be extended to support data with more than two classes, opening other opportunities to use the approach.

CI/CD orchestrators provide observability about processes and actions running in pipelines mainly through build logs. However, we showed that build logs are not sufficient to detect compromise of pipelines, including persistent compromise. Therefore, we encourage research on methods for assessing the integrity of CI/CD systems. This includes research on reproducible CI/CD builds, supply chain attacks, or code integrity protection.

In conclusion, CI/CD has gained traction in recent years, making a shift in software development for open-source and commercial projects. Plenty of challenges remain for ensuring safe and large-scale usage of CI/CD. This suggests more research for properly defining and proposing solutions to these challenges.

Bibliography

[1] Alessandrod. Snuffy. https://github.com/alessandrod/snuffy. Accessed on 12-08-2023.

[2] Alessandrod. Intercepting zoom's encrypted data with bpf. https://confused.ai/posts/intercepting-zoom-tls-encryption-bpf-uprobes, 2020. Accessed on 12-08-2023.

[3] David Alvarez-Melis and Tommi S. Jaakkola. On the robustness of interpretability methods, 2018.

[4] Leila Arras, Franziska Horn, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. Explaining predictions of non-linear classifiers in NLP. In *1st Workshop on Representation Learning for NLP*, pages 1–7, Berlin, Germany, August 2016. Association for Computational Linguistics.

[5] Arstechnica. Sabotage: Code added to popular npm package wiped files in russia and belarus. https://arstechnica.com/information-technology/2022/03/sabotage-code-added-to-popular-npm-package-wiped-files-in-russia-and-belarus/. Accessed on 09-02-2023.

[6] Nicolas Aussel, Yohan Petetin, and Sophie Chabridon. Improving performances of log mining for anomaly prediction through NLP-based log parsing. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 237–243. IEEE, 2018.

[7] MITMproxy's authors. Mitmproxy. https://mitmproxy.org/. Accessed on 12-08-2023.

[8] Avast. Avast. https://www.avast.com. Accessed on 12-08-2023.

[9] AWS. Iam roles for amazon ec2. https://docs.aws.amazon.com/en_en/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html. Accessed on 06-02-2023.

[10] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[11] Julen Kahles Bastida. Applying machine learning to root cause analysis in agile CI/CD software testing environments. Master's thesis, Aalto University, Espoo, Finland, 2018.

[12] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[13] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Tredan. Experience report: Log mining using natural language processing and application to anomaly detection. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 351–360. IEEE, 2017.

[14] Alex Birsan. Dependency confusion: How i hacked into apple, microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, 2021. Accessed on 06-02-2023.

[15] Sanja Bonic, Janos Bonic, and Stefan Schmid. Enhancing continuous integration systems with assisted root cause analysis, 2023.

[16] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-Relational Data. In *26th International Conference on Neural Information Processing Systems (NIPS)*, pages 2787—-2795. Curran Associates Inc., 2013.

[17] Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. https://software-lab.org/publications/icse2024_workflows.pdf, 2024. Accepted in 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24).

[18] Carolin E Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. Logchunks: A data set for build log analysis. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 583–587, 2020.

[19] Thanh Bui. Analysis of docker security. *CoRR*, abs/1501.02967, 2015.

[20] Bootstrappable Builds. Bootstrappable Builds. https://www.bootstrappable.org. Accessed on 10-02-2023.

[21] Reproducible Builds. Reproducible Builds. https://reproducible-builds.org. Accessed on 10-02-2023.

[22] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. Openconflict: Preventing real time map hacks in online games. In *2011 IEEE Symposium on Security and Privacy*, pages 506–520, 2011.

[23] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. BUILDFAST: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53, 2020. ISSN: 2643-1572.

[24] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. Let's supercharge the workflows: An empirical study of github actions. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 01–10, 2021.

[25] Alex Clemmer. How does akamai's 'secure heap' patch to openssl work? - blog post. https://blog.nullspace.io/akamai-ssl-patch.html, 2014. Accessed on 12-08-2023.

[26] Cloudflare. Cloudflare radar. https://radar.cloudflare.com/adoption-and-usage. Accessed on 12-08-2023.

[27] Theo Combe, Antony Martin, and Roberto Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3:54–62, 09 2016.

[28] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[29] Hetong Dai, Heng Li, Weiyi Shang, Tse-Hsun Chen, and Che-Shao Chen. Logram: Efficient log parsing using n-gram dictionaries. 2020.

[30] Sander Vinberg David Warburton. The 2021 TLS Telemetry Report. https://www.f5.com/content/dam/f5-labs-v2/article/pdfs/2021-TLS-Telemetry-Report.pdf, 2021.

[31] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53, 01 2010.

[32] Xavier de Carné de Carnavalet. *Last-Mile TLS Interception: Analysis and Observation of the Non-Public HTTPS Ecosystem*. PhD thesis, Concordia University, May 2019.

[33] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, pages 235–245. IEEE, 2022.

[34] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. A preliminary study of github actions dependencies. In *SATToSE'23: Seminar on Advanced Techniques & Tools for Software Evolution, June 12–14, 2023, Salerno, Italy*, 2023.

[35] Julien Delplanque, Stephane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien. Rotten green tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 500–511. IEEE, 2019.

[36] Y. Dimopoulos, Paul Bourret, and Sovan Lek. Use of some sensitivity criteria for choosing networks with good generalization ability. *Neural Processing Letters*, 2:1–4, 12 1995.

[37] Docker. Content trust in docker. https://docs.docker.com/engine/security/trust/. Accessed on 06-02-2023.

[38] Docker. Docker Compose. https://docs.docker.com/compose/. Accessed on 17-03-2023.

[39] Docker. The Docker Registry 2.0 implementation for storing and distributing Docker images. https://hub.docker.com/_/registry. Accessed on 17-03-2023.

[40] Docker, Inc. Docker CLI. https://github.com/docker/cli/releases/tag/v20.10.12. Accessed on 26-01-2022.

[41] Docker, Inc. Dockerfile Reference. https://docs.docker.com/engine/reference/builder/. Accessed on 10-02-2023.

[42] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 839–850. ACM Press, 2013.

[43] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[44] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[45] V. Dukhovni and W. Hardaker. The dns-based authentication of named entities (dane) protocol: Updates and operational guidance. Internet Requests for Comments, October 2015.

[46] Thomas Durieux. Parfum: Detection and automatic repair of dockerfile smells, 2023.

[47] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

[48] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on travis ci. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 254–264, 2020.

[49] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. The security impact of HTTPS interception. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[50] David Eade. Avast antitrack does not check validity of end web server certificates. https://www.davideade.com/2020/03/avast-antitrack.html, 2020.

[51] Takuma Ebisu and Ryutaro Ichise. Toruse: Knowledge graph embedding on a lie group. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1819–1826. AAAI Press, 2018.

[52] Andreas Ekelhart, Fajar J Ekaputra, and Elmar Kiesling. The SLOGERT framework for automated log knowledge graph construction. page 16, 2021.

[53] Pavel Emelyanov. Checkpoint/restore in userspace, or criu. https://criu.org/.

[54] Pavel Emelyanov. mm: Ability to monitor task memory changes (v3). https://lwn.net/Articles/546966/, 2013. Accessed on 12-08-2023.

[55] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for http. http://www.rfc-editor.org/rfc/rfc7469.txt, April 2015.

[56] F-Secure. Virus:w32/induc.a. https://www.f-secure.com/v-descs/virus_w32_induc_a.shtml. Accessed on 10-02-2023.

[57] F5. How to boost your security with increased visibility. https://www.f5.com/resources/library/encrypted-threats/ssl-visibility, 2022. Accessed on 12-08-2023.

[58] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[59] Taher A. Ghaleb, Safwat Hassan, and Ying Zou. Studying the interplay between the durations and breakages of continuous integration builds. *IEEE Transactions on Software Engineering*, 49(4):2476–2497, 2023.

[60] Omer Gil. Bypassing required reviews using github actions. https://www.cidersecurity.io/blog/research/bypassing-required-reviews-using-github-actions/, 2021. Accessed on 09-02-2023.

[61] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pages 80–89. IEEE, 2018.

[62] Github. Best practices for using the rest api. https://docs.github.com/en/rest/guides/best-practices-for-using-the-rest-api.

[63] Github. Github copilot - your ai pair programmer. https://github.com/features/copilot. Accessed on 12-08-2023.

[64] Github. Using workflow run logs. https://docs.github.com/en/actions/monitoring-and-troubleshooting-workflows/using-workflow-run-logs, 2023. Accessed on 26-07-2023.

[65] GitLab. GitLab Documentation. https://docs.gitlab.com/. Accessed on 17-03-2023.

[66] GitLab. Gitlab Runner. https://docs.gitlab.com/runner/. Accessed on 17-03-2023.

[67] GitLab. Omnibus GitLab Documentation. https://docs.gitlab.com/omnibus/. Accessed on 17-03-2023.

[68] Google. Further improving digital certificate security. https://security.googleblog.com/2013/12/further-improving-digital-certificate.html. Accessed on 12-08-2023.

[69] Google. Google chrome privacy notice. https://www.google.com/chrome/privacy/. Accessed on 12-08-2023.

[70] Google. kaniko - build images in kubernetes. https://github.com/GoogleContainerTools/kaniko. Accessed on 06-02-2023.

[71] Google. Https encryption on the web. https://transparencyreport.google.com/https/overview, 2022. Accessed on 12-05-2023.

[72] Vanessa Gratzer and David Naccache. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 48–58, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[73] Brendan Gregg. Linux load averages: Solving the mystery. https://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html. Accessed on 12-08-2023.

[74] Gunter Rotsaert. Docker Layers Explained. https://dzone.com/articles/docker-layers-explained. Accessed on 17-03-2023.

[75] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM international on conference on information and knowledge management*, pages 1573–1582, 2016.

[76] Foyzul Hassan. Tackling build failures in continuous integration. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1242–1245, 2019. ISSN: 2643-1572.

[77] Foyzul Hassan, Na Meng, and Xiaoyin Wang. Uniloc: Unified fault localization of continuous integration failures. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[78] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhichun Chen, Kangkook Jee, Zhenyu Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[79] Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, and Adam Bates. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[80] Hongjun He, Jicheng Cao, Lesheng Du, Hao Li, Shilong Wang, and Shengyu Cheng. Constbin: A tool for automatic fixing of unreproducible builds. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 97–102, 2020.

[81] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. Towards automated log parsing for large-scale log data analysis. volume 15, pages 931–944. IEEE, 2017.

[82] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[83] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. A survey on automated log analysis for reliability engineering. 54(6):1–37, 2021.

[84] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[85] Zhangyue He, Yanni Tang, Kaiqi Zhao, Jiamou Liu, and Wu Chen. Graph-based log anomaly detection via adversarial training. In Holger Hermanns, Jun Sun, and Lei Bu, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 55–71, Singapore, 2024. Springer Nature Singapore.

[86] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 426–437, 2016.

[87] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). http://www.rfc-editor.org/rfc/rfc6797.txt, November 2012.

[88] Sarah Hodne. Security advisory: Encrypted environment variables. https://blog.travis-ci.com/2016-07-07-security-advisory-encrypted-variables. Accessed on 06-02-2023.

[89] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 473–480, New York, NY, USA, 2016. Association for Computing Machinery.

[90] Internet Engineering Task Force (IETF). The transport layer security (tls) protocol version 1.3. https://datatracker.ietf.org/doc/rfc8446/, 2018. Accessed on 12-08-2023.

[91] Nubeva inc. Nubeva session key intercept, supported signatures. https://docs.nubeva.com/en/latest/files/Signatures.html#linux. Accessed on 12-08-2023.

[92] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. https://www.rfc-editor.org/info/rfc9000, May 2021.

[93] Jenkins. Jenkins. https://www.jenkins.io/, 2023. Accessed on 26-07-2023.

[94] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):249–267, 2008.

[95] Masami Hiramatsu Jim Keniston, Prasanna S Panchamukhi. Kernel probes (kprobes). https://docs.kernel.org/trace/kprobes.html. Accessed on 12-08-2023.

[96] Julen Kahles, Juha Torronen, Timo Huuhtanen, and Alexander Jung. Automating root cause analysis via machine learning in agile software testing environments. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 379–390. IEEE, 2019.

[97] Linux kernel's authors. Kernel tls offload. https://docs.kernel.org/networking/tls-offload.html. Accessed on 12-08-2023.

[98] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[99] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431, 2021.

[100] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, Boston, MA, August 2022. USENIX Association.

[101] P. Ladisa, H. Plate, M. Martinez, and O. Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[102] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, 10:e0130140, 07 2015.

[103] Darragh Leahy and Christina Thorpe. Zero trust container architecture (ztca): A framework for applying zero trust principals to docker containers. *International Conference on Cyber Warfare and Security*, 17:111–120, 03 2022.

[104] Casey Lee. Run your github actions locally. https://github.com/nektos/act.

[105] Jiwei Li, Will Monroe, and Dan Jurafsky. Understanding neural networks through representation erasure, 2017.

[106] Yufei Li, Yanchi Liu, Haoyu Wang, Zhengzhang Chen, Wei Cheng, Yuncong Chen, Wenchao Yu, Haifeng Chen, and Cong Liu. Glad: Content-aware dynamic graphs for log anomaly detection, 2023.

[107] Zhong Li, Jiayang Shi, and Matthijs van Leeuwen. Graph neural networks based log anomaly detection and explanation, 2023.

[108] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111, 2016.

[109] PBOM Ltd. Open software supply chain attack reference (osc&r). https://pbom.dev. Accessed on 07-02-2023.

[110] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[111] A. Martin, Simone Raponi, T. Combe, and Roberto Pietro. Docker ecosystem – vulnerability analysis. *Computer Communications*, 122, 03 2018.

[112] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, apr 2008.

[113] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[114] Afonso Soares Mendonça. Predicting build flakiness by classifying ci/cd log files. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2023.

[115] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, pages 167–177, 2018.

[116] Masayoshi Mizutani. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 2013.

[117] Florent Moriconi, Olivier Levillain, Aurélien Francillon, and Raphael Troncy. X-Ray-TLS: Transparent Decryption of TLS Sessions by Extracting Session Keys from Memory. In *19th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2024.

[118] Florent Moriconi, Axel Ilmari Neergaard, Lucas Georget, Samuel Aubertin, and Aurélien Francillon. Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems. In *17th IEEE Workshop on Offensive Technologies*, pages 219–227, 2023.

[119] Florent Moriconi, Raphael Troncy, Aurélien Francillon, and Jihane Zouaoui. Automated Identification of Flaky Builds using Knowledge Graphs. In *International Conference Knowledge Engineering and Knowledge Management (EKAW), Poster Session*, 2022.

[120] Mozilla. Nss key log format. https://firefox-source-docs.mozilla.org/security/nss/legacy/key_log_format/index.html. Accessed on 12-08-2023.

[121] Mozilla. Sops: Secrets operations. https://github.com/mozilla/sops. Accessed on 06-02-2023.

[122] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR runs encryption securely outside RAM. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[123] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117. IEEE, 2010.

[124] Netskope. Netskope. https://www.netskope.com/. Accessed on 12-08-2023.

[125] Palo Alto Networks. Prisma cloud. https://www.paloaltonetworks.com/prisma/cloud. Accessed on 06-02-2023.

[126] Maximilian Nickel, Lorenzo Rosasco, and Tomaso A. Poggio. Holographic embeddings of knowledge graphs. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1955–1961. AAAI Press, 2016.

[127] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. Automatic fault detection for deep learning programs using graph transformations. *ACM Transactions on Software Engineering and Methodology*, 31(1), 2021.

[128] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[129] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584, 2007.

[130] Tavis Ormandy. Kaspersky: Local ca root is incorrectly protected. https://bugs.chromium.org/p/project-zero/issues/detail?id=989, 2016. Accessed on 12-08-23.

[131] Tavis Ormandy. Kaspersky: Ssl interception differentiates certificates with a 32bit hash. https://bugs.chromium.org/p/project-zero/issues/detail?id=978, 2016. Accessed on 12-08-2023.

[132] OWASP. Owasp top 10 ci/cd security risks. https://owasp.org/www-project-top-10-ci-cd-security-risks/. Accessed on 09-02-2023.

[133] Chris Palmer. Intent to deprecate and remove: Public key pinning. https://groups.google.com/a/chromium.org/g/blink-dev/c/he9tr7p3rZ8/m/eNMwKPmUBAAJ.

[134] Lucas A. Payne. Log file anomaly detection using knowledge graphs and graph neural networks. Master's thesis, The University of Tennessee at Chattanooga, 2023.

[135] Jérôme Petazzoni. Using docker-in-docker for your ci or testing environment? think twice. http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/, 2020. Accessed on 07-02-2023.

[136] Plotly. Dash. https://plotly.com/dash/. Accessed on 01-02-2024.

[137] Nikita Popov. Update on git.php.net incident. https://news-web.php.net/php.internals/113981, 2021. Accessed on 09-02-2023.

[138] IO Visor Project. Bpf compiler collection (bcc). https://github.com/iovisor/bcc. Accessed on 12-08-2023.

[139] Rootkit Hunter project's authors. The rootkit hunter project. https://rkhunter.sourceforge.net/. Accessed on 06-02-2023.

[140] Nathaniel Quist. The anatomy of an attack against a cloud supply pipeline. https://www.paloaltonetworks.com/blog/2021/10/anatomy-ci-cd-pipeline-attack/, 2021. Accessed on 30-03-2023.

[141] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint tls stacks. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 637–657, Cham, 2022. Springer Nature Switzerland.

[142] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355, 2017.

[143] rclone's authors. Rclone's source code - obscure.go. https://github.com/rclone/rclone/blob/313493d51b390d7f73f0780d15bf31698f2a919a/fs/config/obscure/obscure.go#L17. Accessed on 12-08-2023.

[144] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. Automated patching for unreproducible builds. ICSE '22, page 200–211, New York, NY, USA, 2022. Association for Computing Machinery.

[145] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[146] RyotaK. Remote code execution in homebrew by compromising the official cask repository. https://blog.ryotak.net/post/homebrew-security-incident-en/, 2021. Accessed on 09-02-2023.

[147] Baruch Sadogursky. Going beyond exclude patterns: Safe repositories with priority resolution. https://jfrog.com/blog/going-beyond-exclude-patterns-safe-repositories-with-priority-resolution/. Accessed on 06-02-2023.

[148] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[149] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engg.*, 29(1), may 2022.

[150] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. Learning ci configuration correctness for early build feedback. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1006–1017, 2022.

[151] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 121–130, New York, NY, USA, 2023. Association for Computing Machinery.

[152] John Regehr Scott Bauer, Pascal Cuoq. Deniable backdoors using compiler bugs. *International Journal of Proof-of-Concept or Get The Fuck Out (PoC‖GTFO)*, 0x08, 2015.

[153] Keiichi Shima. Length matters: Clustering system log messages using length of words, 2016.

[154] Gustavo Silva, Carla Bezerra, Anderson Uchôa, and Ivan Machado. What factors affect the build failures correction time? a multi-project study. In *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 41–50, 2023.

[155] Sudheesh Singanamalla, Muhammad Talha Paracha, Suleman Ahmad, Jonathan Hoyland, Luke Valenta, Yevgen Safronov, Peter Wu, Andrew Galloni, Kurtis Heimerl, Nick Sullivan, Christopher A. Wood, and Marwan Fayed. Respect the origin! a best-case evaluation of connection coalescing in the wild. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 664–678, New York, NY, USA, 2022. Association for Computing Machinery.

[156] Yrjan Skrimstad. Improving trust in software through diverse double-compiling and reproducible builds. 2018.

[157] SLSA. Supply chain Levels for Software Artifacts. https://slsa.dev. Accessed on 10-02-2023.

[158] Splunk. Splunk | turn data into doing. https://www.splunk.com. Accessed on 26-07-2023.

[159] sslmate. Timeline of certificate authority failures. https://sslmate.com/resources/certificate_authority_failures. Accessed on 12-08-2023.

[160] Yicheng Sui, Yuzhe Zhang, Jianjun Sun, Ting Xu, Shenglin Zhang, Zhengdan Li, Yongqian Sun, Fangrui Guo, Junyu Shen, Yuzhi Zhang, Dan Pei, Xiao Yang, and Li Yu. Logkg: Log failure diagnosis through knowledge graph. *IEEE Transactions on Services Computing*, pages 1–14, 2023.

[161] Lionel Tailhardat, Yoan Chabot, and Raphaël Troncy. Noria-o: an ontology for anomaly detection and incident management in ict systems. *Semantic Web Journal*, 2022.

[162] Lionel Tailhardat, Raphaël Troncy, and Yoan Chabot. Leveraging knowledge graphs for classifying incident situations in ict systems. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery.

[163] Liran Tal. snync. https://github.com/snyk-labs/snync. Accessed on 06-02-2023.

[164] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794, 2011.

[165] Benjamin Taubmann, Omar Alabduljaleel, and Hans P. Reiser. Droidkex: Fast extraction of ephemeral TLS keys from the memory of android apps. *Digit. Investig.*, 26 Supplement:S67–S76, 2018.

[166] Benjamin Taubmann, Christoph Frädrich, Dominik Dusold, and Hans P. Reiser. Tlskex: Harnessing virtual machine introspection for decrypting TLS communication. *Digit. Investig.*, 16 Supplement:S114–S123, 2016.

[167] Tenable. Ensure only trusted users are allowed to control docker daemon. https://www.tenable.com/audits/items/CIS_Docker_Community_Edition_L1_Linux_Host_OS_v1.1.0.audit:6b5f6af12d7a9a4ce9130106434e64d7. Accessed on 06-02-2023.

[168] Sam L. Thomas and Aurélien Francillon. Backdoors: Definition, deniability and detection. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing, 2018.

[169] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, aug 1984.

[170] Ray Toal. Quine Programs. https://cs.lmu.edu/~ray/notes/quineprograms/. Accessed on 07-02-2023.

[171] Travis CI. Travis CI Now Introduces Changes in Job Logs Availability. https://www.travis-ci.com/blog/22-9-30-joblogs/, 2022. Accessed on 26-07-2023.

[172] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126. Ieee, 2003.

[173] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[174] Tom van Leijenhorst, Kwan-Wu Chin, and Darryn Lowe. On the viability and performance of dns tunneling. 2008.

[175] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *44th International Conference on Software Engineering (ICSE)*, page 561–572. Association for Computing Machinery, 2022.

[176] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: Fault localization for deep neural networks. In *43rd International Conference on Software Engineering (ICSE)*, page 251–262. IEEE Press, 2021.

[177] Tyler Welton. Exploiting continuous integration (ci) and automated build systems. https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20spaceB0x-Exploiting-Continuous-Integration-UPDATED.pdf. Accessed on 09-02-2023.

[178] David A. Wheeler. Fully countering trusting trust through diverse double-compiling. *CoRR*, abs/1004.5534, 2010.

[179] Wikipedia. Bootstrapping. https://en.wikipedia.org/wiki/Bootstrapping#Installers. Accessed on 10-02-2023.

[180] Wikipedia. Comparison of tls implementations. https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations. Accessed on 12-08-2023.

[181] Wikipedia. Hmac. https://en.wikipedia.org/wiki/HMAC. Accessed on 01-02-2024.

[182] Wikipedia. Self-hosting (compilers). https://en.wikipedia.org/wiki/Self-hosting_(compilers). Accessed on 06-02-2023.

[183] L. Williams. Trusting trust: Humans in the software supply chain loop. *IEEE Security & Privacy*, 20(05):7–10, sep 2022.

[184] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[185] Qiushi Wu and Kangjie Lu. On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. 2021.

[186] Claud Xiao. Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store. https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/. Accessed on 10-02-2023.

[187] Yongzheng Xie, Hongyu Zhang, and Muhammad Ali Babar. Loggd: Detecting anomalies from system logs with graph neural networks. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 299–310, 2022.

[188] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Large-scale system problems detection by mining console logs. page 17, 2008.

[189] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. https://arxiv.org/abs/1412.6575, 2014.

[190] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela F Ciocarlie, Vinod Yegneswaran, et al. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *NDSS*, 2021.

[191] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. 25(2):1095–1135, 2020.

[192] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV*, pages 818–833, Cham, 2014. Springer International Publishing.

[193] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[194] Jingwen Zhou, Zhenbang Chen, Ji Wang, Zibin Zheng, and Michael R. Lyu. Trace bench: An open data set for trace-oriented monitoring. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 519–526, 2014.

[195] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023.

[196] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing. 2019.

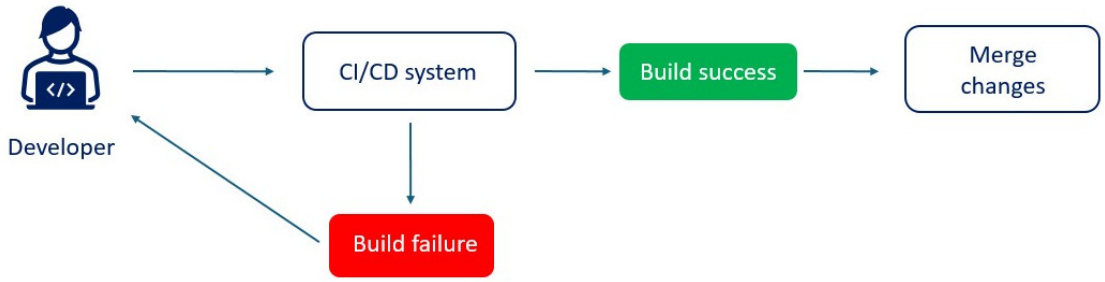

Figure 1: Illustration de la boucle de retour via la CI/CD.

Résumé en français

A.1 Introduction

L'importance du génie logiciel dans le paysage technologique contemporain n'est plus à démontrer ; il représente la colonne vertébrale de l'innovation, de l'efficacité et de l'avantage concurrentiel dans divers secteurs. Chez Amadeus IT Group, un leader des solutions informatiques pour l'industrie mondiale du voyage et du tourisme, le génie logiciel est central pour atteindre les objectifs commerciaux, améliorant l'efficacité opérationnelle et l'expérience de voyage de l'utilisateur final.

Les pratiques de développement logiciel comme l'Intégration Continue et le Déploiement Continu (CI/CD) facilitent les tests et déploiements automatisés, permettant des améliorations itératives rapides et sûres. Cependant, des vérifications (dont la conclusion est succès ou échec) régulières peuvent entraîner un grand nombre d'échec. L'analyse automatique des causes de défaillance devient essentielle pour identifier et corriger rapidement les problèmes.

Des méthodologies modernes telles que DevOps visent à comprimer le cycle de vie du développement système pour permettre des mises à jour régulières et sécurisées. L'approche DevOps favorise l'intégration continue pour minimiser la boucle de rétroaction, garantissant que les développeurs sont rapidement informés des problèmes liés aux modifications du code. La Figure 1 illustre la boucle de retour fournit par la CI/CD.

Les recherches d'IBM [31] mettent en évidence l'impact financier exponentiel de la correction d'un bug à différents stades du cycle de développement logiciel, soulignant l'importance de détecter les défauts logiciels tôt pour réduire les coûts de correction.

Le CI concerne l'intégration automatisée des modifications de code de plusieurs contributeurs dans un seul projet logiciel, tandis que le CD automatise la mise en production. Les systèmes CI/CD suivent des motifs architecturaux communs, comme le "Pipeline as Code", les déclencheurs événementiels, la conception modulaire, et l'exécution parallèle et séquentielle.

Différents systèmes de CI/CD sont disponibles, à la fois de façon commerciale et en sources ouvertes, avec des acteurs comme GitHub (Microsoft), GitLab Inc., et Jenkins.

Les journaux de build CI/CD se distinguent des journaux système par leur focalisation sur le cycle de vie du développement logiciel, documentant les processus de construction, de test et de déploiement des projets logiciels, contrairement aux journaux de production qui enregistrent le fonctionnement des applications en environnement de production. Ainsi, les journaux de build CI/CD possèdent des caractéristiques différentes des journaux de production.

Identifier les types d'échecs de build en CI/CD est crucial pour maintenir l'efficacité et la fiabilité du processus de développement logiciel. Les échecs peuvent être déterministes ou non-déterministes, chacun ayant des caractéristiques et implications spécifiques.

L'analyse des causes racines aide à déterminer la raison fondamentale d'un échec de build, facilitant ainsi sa résolution rapide. Cette analyse peut être réalisée par diverses méthodes, y compris l'étiquetage d'un échec, la localisation du message de cause racine au sein du log, ou la proposition de correctifs.

La sécurité des pipelines CI/CD est essentielle, car toute vulnérabilité peut entraîner des risques de sécurité significatifs, incluant l'accès non autorisé aux données sensibles ou le déploiement de code malveillant.

Cette thèse se lance dans une exploration approfondie à travers plusieurs domaines de recherche, entrelaçant les disciplines du génie logiciel, de la gestion des connaissances, de l'interprétabilité de l'IA, et de la sécurité logicielle. De ce fait, les questions de recherche couvrent plusieurs domaines.

- **RQ1** : Que pouvons-nous apprendre d'un large corpus de journaux de construction CI/CD, et en particulier, pouvons-nous identifier les raisons des échecs ?

  La première question de recherche se concentre sur l'extraction de connaissances précieuses à partir d'un vaste corpus de journaux de construction CI/CD pour améliorer les méthodologies et l'efficacité du développement logiciel. Elle explore principalement la faisabilité d'identifier et de classer automatiquement les causes des échecs de construction. En analysant les motifs récurrents présents dans les journaux de construction à grande échelle, cette recherche vise à approfondir la compréhension des processus et des outils employés au sein des pipelines CI/CD. Une telle analyse peut faciliter des stratégies de dépannage plus efficaces, identifier les goulets d'étranglement de l'infrastructure ou les erreurs de configuration, et identifier les opportunités d'optimisation.

- **RQ2** : Comment les échecs CI/CD peuvent-ils être classifiés automatiquement ?

La classification automatique des échecs de construction CI/CD fait référence à la catégorisation des échecs de construction dans des catégories prédéfinies définissant les erreurs de code, les erreurs de configuration, ou les défaillances transitoires. Ce processus nécessite généralement le traitement des données de construction pour extraire les caractéristiques pertinentes, telles que les messages d'erreur, les modifications de code, ou les détails de l'environnement de construction, qui peuvent ensuite être utilisés pour entraîner un modèle à reconnaître les motifs associés aux différents types de défaillances. L'objectif ultime est de permettre une identification automatisée de la nature d'une défaillance sans intervention humaine, facilitant ainsi un dépannage plus rapide, des corrections automatisées, ou la surveillance des tendances de défaillance à grande échelle.

- **RQ3** : Comment définir et localiser les messages de cause racine dans les journaux de construction CI/CD ?

  Étiqueter les constructions échouées offre des aperçus précieux sur la catégorisation des échecs, mais obtenir une classification détaillée peut être difficile en raison du besoin potentiel de nombreuses étiquettes. Une approche alternative pour aider les opérateurs à résoudre les problèmes implique d'extraire les messages de journal des journaux de construction qui aident à réparer les constructions. Le défi initial réside dans la définition et la caractérisation des messages de cause racine. Le défi subséquent consiste à automatiser la reconnaissance de ces messages de journal, car l'identification manuelle s'avère être laborieuse et peu pratique pour une mise à l'échelle, en particulier lorsqu'il s'agit de gérer des milliers d'échecs de construction quotidiens.

- **RQ4** : Les journaux sont-ils suffisants pour détecter une compromission CI/CD ?

  L'intégrité, la confidentialité, et la disponibilité du logiciel sont considérablement influencées par la sécurité des pipelines CI/CD, car ces pipelines jouent un rôle crucial dans l'automatisation des processus de développement et de déploiement logiciels. Une brèche dans un pipeline CI/CD peut permettre aux attaquants d'incorporer des vulnérabilités ou des portes dérobées dans le logiciel, posant des risques de violation de données, d'accès non autorisé à des informations sensibles, et potentiellement permettant une exploitation plus poussée de l'infrastructure du système. Bien que les journaux des processus CI/CD offrent des aperçus cruciaux, leur efficacité pour assurer la sécurité des systèmes CI/CD soulève des questions. Cela incite à une investigation sur le degré auquel les données de journal peuvent offrir des assurances de sécurité pour les systèmes CI/CD.

La thèse comprend huit chapitres traitant des défis de recherche dans l'analyse des journaux, les graphes de connaissances, la sécurité des réseaux, la localisation des fautes et la sécurité des CI/CD, menant à trois publications dans des conférences internationales [117, 118, 119].

A.2 État de l'art

Dans le chapitre sur l'état de l'art de cette thèse, nous explorons le corpus existant couvrant plusieurs domaines de recherche pertinents aux investigations menées ici. Notre exploration a débuté par l'examen de jeux de données publics de journaux d'Intégration Continue/Déploiement Continu (CI/CD), où nous avons identifié une pénurie notable. Nous soulignons que les jeux de données disponibles sont insuffisants pour mener des analyses de machine learning à grande échelle en raison de diverses limitations.

Par la suite, notre attention se déplace vers les méthodes d'analyse de journaux, un composant critique étant donné son importance dans les méthodologies d'analyse des causes racines. Dans ce contexte, nous examinons la littérature qui utilise des techniques sémantiques, en particulier celles qui emploient le traitement du langage naturel (NLP), comme une autre approche pour l'analyse des journaux. Cette revue examine les Graphes de Connaissances (Knowledge Graphs) comme une approche novatrice pour améliorer l'analyse des journaux et faciliter une identification plus efficace des causes racines.

De plus, notre investigation englobe les phénomènes de défaillances dans les processus de CI/CD. Ici, nous couvrons un spectre de recherches allant des études empiriques à la prédiction, la classification, la localisation et la réparation automatique des défaillances, offrant un aperçu des méthodologies actuelles et de leurs applications.

En outre, la thèse explore le domaine de l'analyse du comportement des modèles d'intelligence artificielle, mettant en lumière les méthodologies actuelles et leurs implications pour les processus de CI/CD. Nous examinons également les méthodes d'inspection TLS afin d'augmenter la quantité de données disponibles pour l'analyse de cause racine.

Enfin, nous orientons notre focus vers le domaine de la sécurité CI/CD, en passant en revue les recherches existantes pour comprendre le paysage des pratiques, des défis et des avancées en matière de sécurité dans les environnements CI/CD.

A.3 Collection des builds de CI/CD

L'intégration continue et le déploiement (CI/CD) ont gagné en popularité ces dernières années, tant dans la communauté open-source que dans l'industrie. Pour soutenir une bonne expérience CI/CD, GitHub a lancé GitHub Actions (GHA) en 2019. En 2023, près de 12% des projets avec plus de 100 étoiles utilisent régulièrement GitHub Actions. Une caractéristique notable de GHA est le concept d'*actions* qui sont des blocs de construction réutilisables pour les flux de travail CI/CD. Plusieurs études ont exploré l'utilisation des actions dans les pipelines CI/CD. Cependant, malgré l'existence des *actions*, les flux de travail CI/CD utilisent également des commandes shell. Bien qu'elles représentent la méthode la plus traditionnelle (et parfois

la seule dans certaines infrastructures CI/CD), il n'existe, à notre connaissance, aucune étude existante sur la manière dont les développeurs utilisent de telles commandes dans les flux de travail de GitHub Actions. Dans ce travail, nous réalisons une analyse de l'utilisation des commandes shell dans les flux de travail CI/CD de GHA. Nous étudions dans quelle mesure les commandes shell sont utilisées, quelle part du temps d'exécution elles représentent, et combien ce temps d'exécution est volatile sur un nouveau jeu de données que nous mettons à disposition de la communauté, composé de 116k workflow CI/CD répartis dans 28k répertoires de code à travers 20 langages de programmation. Nous découvrons que même dans GHA, où des actions réutilisables sont disponibles, 56% des étapes des flux de travail sont des commandes shell, et elles représentent 82% du temps d'exécution. De plus, nous constatons que certaines commandes shell ont un temps d'exécution très volatile, soulevant des questions sur les opportunités d'optimisation dans leur configuration.

A.4 Classification des échecs de construction de CI/CD

La catégorisation des échecs de construction est une méthode efficace pour aider les opérateurs à diagnostiquer les pannes. De plus, la mise en place de systèmes automatisés pour classifier ces échecs joue un rôle significatif dans l'identification des motifs de défaillance à grande échelle ou dans le déclenchement de mesures correctives automatiques. Pour réaliser une telle classification, il est impératif de délimiter des classes distinctes. Les critères de définition de ces classes varient largement, allant d'une catégorisation générale à une approche plus détaillée et fine.

Dans ce chapitre, l'examen se concentre sur deux méthodologies pour séparer les échecs CI/CD en groupes déterministes et non déterministes. Il a été démontré qu'en utilisant une technique de traitement du langage naturel simple mais efficace, le TF-IDF combiné à l'algorithme Random Forest, cela permet la classification des échecs avec un taux de précision de 78% sur un ensemble de test équilibré. Étant donné que les journaux de construction représentent seulement une fraction des données disponibles sur les constructions CI/CD, l'utilisation de graphes de connaissances a été explorée [119]. Une ontologie conçue pour l'écosystème Jenkins a été développée. Ensuite, les constructions CI/CD ont été intégrées dans des graphes de connaissances. Par la suite, les plongements de graphes de connaissances ont été utilisés pour prédire les liens entre les entités, fournissant ainsi une analyse efficace des causes profondes des échecs de construction. Cette stratégie a atteint un taux de précision de 94%, surpassant la performance de l'approche reposant uniquement sur les journaux de construction.

A.5 Inspection du traffic TLS

Bien que les communications sur Internet aient été à l'origine entièrement non chiffrées, la dernière décennie a vu les protocoles sécurisés comme TLS devenir omniprésents, améliorant considérablement la sécurité sur Internet pour les individus et les entreprises. Cependant, le trafic chiffré soulève de nouveaux défis pour la détection des intrusions et la surveillance du réseau. Les solutions d'interception existantes telles que l'attaque de l'homme du milieu (Man-In-The-Middle) sont indésirables dans de nombreux contextes : elles tendent à réduire la sécurité globale ou sont difficiles à utiliser à grande échelle. Nous présentons X-Ray-TLS, une nouvelle méthode de déchiffrement TLS agnostique de la cible qui prend en charge TLS1.2, TLS1.3 et QUIC. Notre méthode repose uniquement sur les installations existantes du noyau et ne nécessite ni hyperviseur ni modification des programmes cibles, ce qui la rend facilement applicable à grande échelle. X-Ray-TLS fonctionne sur les principales bibliothèques TLS en extrayant les secrets TLS de la mémoire des processus en utilisant un algorithme pour reconstruire les modifications de la mémoire. X-Ray-TLS prend en charge les programmes utilisant des techniques de durcissement TLS, telles que le pinning de certificat et la confidentialité persistante (PFS). Nous évaluons X-Ray-TLS sur les principales bibliothèques TLS, des outils en ligne de commande et un navigateur web. Nous démontrons que X-Ray-TLS réduit considérablement l'effort manuel nécessaire pour déchiffrer le trafic TLS des programmes exécutés localement, simplifiant ainsi l'analyse de sécurité ou le reverse engineering. Nous avons identifié plusieurs cas d'utilisation pour X-Ray-TLS, tels que le déchiffrement TLS à grande échelle pour les pipelines CI/CD afin de soutenir l'analyse des causes profondes ou la détection des attaques sur la chaîne d'approvisionnement logicielle.

A.6 Localisation des fautes dans les logs de CI/CD

Lorsque les constructions échouent, ce qui arrive des milliers de fois par jour dans les grandes entreprises, les développeurs doivent enquêter pour trouver les causes profondes de l'échec. Le journal de construction est volumineux pour les pipelines CI complexes, rendant l'investigation des causes profondes chronophage et fastidieuse. Nous proposons ChangeMyMind, une nouvelle approche pour localiser les messages de cause profonde dans les journaux. Notre système ne nécessite pas un ensemble de données étiqueté de messages de cause profonde, le rendant simple à appliquer. Nous n'avons également pas besoin de parser et d'interpréter les journaux, rendant notre approche générique pour tout système qui produit des journaux de succès/échec (par exemple, journaux de test, journaux de transactions). D'abord, l'algorithme est entraîné pour classifier les journaux de succès des journaux d'échec comme une tâche prétexte. L'algorithme identifiera les motifs menant à l'échec, à savoir les messages de cause profonde. Ensuite, dans la phase d'analyse, notre approche tire parti de l'évolution des croyances de l'algorithme de réseau neuronal récurrent (RNN)

pendant la phase d'inférence (prédiction). Nous proposons différentes heuristiques pour extraire l'emplacement exact du message de cause profonde dans le journal en utilisant cette évolution de croyance du RNN. Nous évaluons notre approche sur un ensemble de données synthétique et des journaux de construction réels qui sont étiquetés manuellement. Nous démontrons empiriquement que ChangeMyMind permet de localiser les messages de cause profonde dans les journaux de construction CI/CD avec une erreur de prédiction inférieure à 5 lignes pour 54% des journaux et inférieure à 50 lignes pour 71% des journaux. Par conséquent, cela réduit considérablement le nombre de lignes à investiguer par les opérateurs en cas de défaillance, rendant ainsi le dépannage des échecs de construction plus facile et plus rapide.

A.7 Sécurité des pipelines CI/CD

De nos jours, les orchestrateurs CI/CD utilisent largement les conteneurs. En effet, les conteneurs permettent des environnements identiques entre un ordinateur de développeur, un environnement de construction de CI, et les serveurs de production. Lorsqu'un build CI commence (par exemple, suite à un changement de code), l'orchestrateur démarre un nouveau conteneur. Ce conteneur est basé sur une image de conteneur appelée l'image de build CI. Cette image est généralement commune à tous les builds d'une famille spécifique (par exemple, logiciels basés sur Java, logiciels basés sur Python). Elle doit contenir tous les outils nécessaires pour effectuer les étapes du CI : compilateurs, linters, analyseurs de code, orchestrateurs de test, outils spécifiques à l'entreprise, etc. Par conséquent, il est courant pour les développeurs de personnaliser cette image pour inclure les outils nécessaires. Cela signifie que la construction de l'image CI est considérée comme un autre logiciel ; donc, elle est souvent construite en utilisant le système CI lui-même. En conséquence, les systèmes CI sont considérés comme des architectures auto-hébergées [182]. Thompson [169] a montré que les architectures auto-hébergées soulèvent de nouvelles questions sur la sécurité.

Revisitant le célèbre backdoor de compilateur de Ken Thompson, nous montrons qu'un système CI basé sur des conteneurs peut être compromis sans laisser de trace dans le code source. Par conséquent, détecter un tel logiciel malveillant est difficile, voire impossible, avec des pratiques courantes telles que la revue par les pairs ou l'analyse de code statique. Nous détaillons plusieurs façons de réaliser le processus d'infection initial. Ensuite, nous montrons comment persister lors des mises à jour du système CI, permettant une compromission à long terme. Nous détaillons les charges utiles d'attaque malicieuses possibles telles que l'extraction de données sensibles ou l'installation de portes dérobées dans le logiciel de production. Nous montrons que les systèmes CI infectés peuvent être contrôlés à distance en utilisant des canaux cachés pour mettre à jour la charge utile d'attaque ou adapter le logiciel malveillant aux stratégies de mitigation. Enfin, nous proposons une implémentation d'une preuve de concept testée sur GitLab CI et applicable aux principaux fournisseurs de CI.

A.8 Conclusion

Les pratiques d'Intégration Continue/Déploiement Continu (CI/CD) se sont imposées comme des composantes cruciales du cycle de vie du développement logiciel à l'ère moderne. À travers cette thèse, nous avons exploré plusieurs domaines de recherche, allant de la collecte et l'analyse des données de logs CI/CD, à l'automatisation de la classification des échecs CI/CD, la localisation des messages de cause racine dans les logs, la sécurité réseau, et la sécurité des systèmes de CI/CD.

Notre première question de recherche se concentre sur le potentiel d'apprentissage à partir d'un grand corpus de logs de build CI/CD, en identifiant particulièrement les motifs d'échec de build. Dans le Chapitre 3, nous avons collecté un large corpus d'exécutions CI/CD dans un contexte industriel. Nous avons proposé une méthode pour identifier automatiquement les raisons des échecs en exploitant l'action corrective effectuée par les développeurs (par exemple, un patch de code) entre deux builds successifs. Cette méthode permet de labelliser environ 10% des builds historiques ayant échoué. Notamment, nous avons identifié que de nombreux builds sont corrigés sans changement de code, suggérant une certaine instabilité. Considérant le manque de datasets de logs CI/CD publics, nous avons construit un dataset à partir de dépôts de code publics utilisant GitHub Actions. Le dataset public comprend 116k workflows GitHub Actions répartis dans 28k répertoires de code à travers 20 langages de programmation. Ensuite, nous analysons les étapes de pipeline définies dans ces workflows. Nous découvrons que même si Github Actions fournit des actions réutilisables, 56% des étapes des workflows sont des commandes shell, et elles représentent 82% du temps d'exécution. Par conséquent, cela encourage les études sur les pipelines CI/CD à se concentrer sur les étapes shell, souvent ignorées dans la littérature.

Notre deuxième question de recherche se concentre sur la classification automatisée des échecs de build CI/CD. Le Chapitre 4 explore l'utilisation de techniques de traitement automatique du langage sur les logs de build pour classer les échecs de build entre échecs non-déterministes (par exemple, instabilités) et échecs déterministes (par exemple, erreurs de code). Nous avons obtenu une précision de classification de 78% sur un dataset industriel équilibré. Considérant que les logs de build représentent seulement une proportion des données de build globales, nous avons tenté la même tâche de classification en utilisant des graphes de connaissances. D'abord, nous avons proposé une ontologie pour rendre explicite la sémantique de l'écosystème CI/CD. Ensuite, nous utilisons des techniques d'Embedding de Graphe de Connaissances pour faire de la prédiction de liens, ce qui équivaut à classer les échecs de build. Cette approche atteint une précision de 94% sur un dataset industriel équilibré, surpassant ainsi l'approche basée sur le traitement du langage naturel. Ce travail a mené à une publication [119] pour montrer l'utilisation des graphes de connaissances pour la classification des builds CI/CD.

Les builds CI/CD interagissent fortement avec leur environnement, notamment à travers le réseau. Ainsi, analyser les interactions réseau peut aider à identifier les raisons des échecs. Cependant, le trafic réseau est presque entièrement chiffré en TLS, et les méthodes existantes pour l'inspection du TLS sont difficiles à utiliser dans les environnements CI/CD, par exemple en raison de la grande versatilité des logiciels exécutés dans les pipelines. Le Chapitre 5 examine les stratégies pour le déchiffrement transparent et universel du trafic réseau chiffré en TLS avec une application aux environnements de build CI/CD. Ce travail a mené à une publication [117] pour décrire X-Ray-TLS, une nouvelle méthode basée sur l'introspection de la mémoire pour déchiffrer de façon transparente le trafic TLS.

Contrairement à la classification des builds, nous avons également exploré comment définir et localiser les messages de cause racine dans les logs de build CI/CD. En effet, prédire la localisation de la cause racine est une autre méthode qui permet de faciliter le dépannage des échecs de build. Dans le Chapitre 6, nous proposons ChangeMyMind, une méthode pour localiser les messages de cause racine dans les logs de build CI/CD. D'abord, nous proposons une définition des messages de cause racine. Ensuite, nous avons entraîné un algorithme de réseau de neurones pour classer les logs réussis et échoués. Enfin, en surveillant la phase d'inférence du réseau de neurones, nous pouvons identifier la zone des logs de build qui contient probablement le message de cause racine. La méthode ne nécessite pas un dataset étiqueté de messages de cause racine, ce qui est souvent difficile à construire en pratique, ce qui marque un changement majeur par rapport aux approches existantes qui se basent sur des connaissances existantes sur les motifs d'échec. Nous démontrons empiriquement que ChangeMyMind permet la localisation des messages de cause racine dans les logs de build CI/CD avec une erreur de prédiction inférieure à 5 lignes pour 54% des logs et inférieure à 50 lignes pour 71% des logs. Par conséquent, cela réduit significativement le nombre de lignes à investiguer par les opérateurs en cas d'échec, rendant ainsi le dépannage des échecs de build plus facile et plus rapide.

Les logs de build fournissent des aperçus sur les processus et actions dans les pipelines CI/CD. Cependant, les logs sont-ils suffisants pour détecter les compromissions de sécurité CI/CD ? Dans le Chapitre 7, nous avons exploré les propriétés de sécurité des systèmes CI/CD auto-hébergés. En revisitant le célèbre backdoor de compilateur de Ken Thompson [169], nous montrons qu'un système CI/CD basé sur des conteneurs peut être compromis à long terme sans laisser de trace dans les dépôts de code ou les logs de build. Par conséquent, détecter de tels malwares est un défi, voire impossible, avec les pratiques courantes comme la revue par les pairs, l'analyse de code statique, ou l'analyse des logs de build. Ce travail a mené à une publication [118] qui présente un concept de preuve ciblant Gitlab CI.

En conclusion, CI/CD a gagné en popularité ces dernières années, entraînant un changement dans le développement logiciel pour les projets open-source et commerciaux. De nombreux

défis demeurent pour assurer une utilisation sûre et à grande échelle du CI/CD. Cela suggère plus de recherches pour définir correctement et proposer des solutions à ces défis.