HAW HAMBURG

BACHELOR THESIS

Evaluating Supply Chain Security Risks in CI/CD-Pipelines:

A Case Study on Mutable GitHub Actions

submitted on 31 March 2025

Ronny Manjang

First examiner: Prof. Dr. Larissa Putzar
Second examiner: Florian Junge

HAMBURG UNIVERSITY OF APPLIED SCIENCES
Faculty of Engineering and Computer Science
Department of Computer Science and Digital Societies
Finkenau 35
22081 Hamburg

Abstract

This bachelor thesis investigates supply chain security risks in GitHub Actions-based CI/CD environments, with a critical focus on the often-overlooked dangers of mutable and transitive dependencies. While modern automated workflows rely heavily on third-party code, the lack of strict governance and native lockfile mechanisms in GitHub Actions creates significant blind spots. To address this, the thesis introduces the Mutable Dependency Risk Score (MDRS) model, a structured analytical framework designed to identify and quantify high-risk dependency patterns.

The theoretical model is being put into practice through the development of the MDRS Checker, a software tool capable of performing recursive dependency resolution across multiple ecosystems. By applying the tool to real-world case studies, including a simulated environment of the devastating *ua-parser-js* supply chain attack, the evaluation demonstrates the models efficacy. The results show that the implementation of a severity override mechanism successfully surfaces deeply hidden transitive risks. Ultimately, this thesis provides software engineering teams with both the theoretical understanding and the practical tooling necessary to expose invisible weaknesses and strategically prioritize their mitigation efforts.

Contents

1 Introduction

1.1 Motivation

Organizations rely on automation to distribute and integrate their software products. This process, called Continuous Integration / Continuous Deployment CI/CD, is a common practice in the realm of software development. [40]
Many organizations create complex software supply chains to fulfill this automation requirement. These supply chains rely on open source components that are often developed and maintained by hobbyists. [7]
This dependency on open source components in supply chains wasn't always the norm. With the introduction of GitHub Actions (GHA) in 2019, organizations opted for a more platform integrated approach and slowly migrated their CI/CD pipelines with said platforms. Delicheh et al. [12] clearly shows how a large-scale shift followed, after the introduction of GHA in the CI/CD landscape.
While this community driven open source approach helps with a more diverse and open environment, it also introduces new risks.
With this research, I will focus on GitHub, as it is one of the most widely used software development platforms and offers the functionality to build fully integrated supply chain pipelines through GitHub Actions.

1.2 Research Goals

Modern software development has scaled rapidly by building upon a foundational ecosystem of open-source components. However, this growth has created a significant

security gap [43], as the integrity of these external dependencies often remains entirely outside the direct control of project maintainers. In recent years, empirical evidence from security incidents shows a strategic shift toward complex attack chains that exploit the layered and interconnected nature of these supply chains [15] .

Research suggests that security risk in these environments scales proportionally to the total number of components within a system [31] [8] . Because GitHub Actions (GHA) often rely on a high volume of "hidden" or transitive components, they represent a substantial, yet rarely quantified, risk surface [6]. Even when developers follow best practices by pinning direct components, they face an *invisible risk*: the transitive dependencies deeper in the graph often remain mutable and unprotected.

To address this lack of control over external dependencies, the concept of *visibility* becomes essential. By actively monitoring the inventory of used dependencies and their derived transitives, maintainers can gain necessary insights into the actual attack surface and inherent risks of a project. Consequently, the main thesis of this work is that the pinning of root components in GitHub Actions is insufficient on its own, as it masks the pervasive and often unaddressed risk of mutable transitive dependencies.

This research focuses on the tools and strategies used to gain more visibility in GitHub Actions based CI/CD pipelines. Following the main thesis, the research questions are as follows:

- **RQ1:** What specific attack vectors and security risks arise from mutable transitive dependencies in pinned GitHub Actions workflows?
- **RQ2:** How can organizations effectively assess and prioritize the risks posed by mutable dependencies in their CI/CD pipelines?

1.2.1 Related Work

Related work in the field comprises of large scale empirical studies on various security aspects of GitHub Actions, such as the study by Delicheh et al. [12] that analyzes the depth of transitive dependencies in GHA. The study by Benedetti et al. [7] proposes a automated method for assessing workflow security, based on multiple metrics with

the more general goal to strengthening the security targets defined by the CIA Triad (see Section 2.4.1). He et al. [25] provides evidence with their study that the practice of dependency pinning is not sufficient to mitigate the risks of mutable transitive dependencies in the *npm* ecosystem. This ties into RQ1 and provides a strong motivation for this research, as it suggests that the same problem may exist in the context of GitHub Actions. A key takeaway from He et al. is that the implementation of lockfiles in the npm ecosystem could have mitigated the risks of mutable transitive dependencies, which is a potential mitigation strategy that will be discussed in more detail in Chapter 5.4.
The table below compares the related work based on selected criteria and key differences:

Table 1: Comparison of related work based on selected criteria and key differences.

Work	**Transitive vs. Direct**	**Ecosystem**	**Key Difference**
Pinning is Futile, He et al. [25]	Transitive	npm	A npm focused impact analysis without tool evaluation.
A Preliminary Study of GitHub Actions Dependencies [12]	Transitive	GitHub Actions	Inspects transitive dependency depth but not the mutability.
Automatic Security Assessment of GitHub Actions Workflows [7]	Direct	GitHub Actions	Proposes automated methods for assessing workflow security without considering transitive dependencies.
Ours	113 Repos	GitHub Actions	Provides an automatic assessment strategy for visibility and risk quantification of transitive dependencies in GHA.

2 Background

Before we can analyze the security risks of using GHA in CI/CD pipelines, it is important to understand the background and context of these technologies. In this chapter, I will provide an overview of the key technological concepts. This background information will provide a foundation for understanding the security risks associated with mutable dependencies in GHA and inform the analysis and mitigation strategies discussed in later chapters.

2.1 CI/CD Pipelines and Software Supply Chains

In software development organizations, the CI/CD process is often realized by constructing complex pipelines that combine build systems, distribution components, and management automations. These pipelines can span different development platforms and interact with numerous live production systems, such as databases and APIs. For these pipelines to act autonomously, privileged credentials are necessary. When these credentials are injected into pipeline components, the risk of credential theft or hijacking of high-privilege components increases. [12] The software supply chain encompasses all the processes and components involved in the development, distribution, and deployment of software. This includes source code repositories, build systems, package managers, and deployment tools. The security of the software supply chain is critical, as weaknesses or malicious code introduced at any stage can have far-reaching consequences. In the context of CI/CD pipelines, the software supply chain includes all the dependencies and tools used to build and deploy applications, making it essential to understand and manage the risks associated with these components.

2.2 GitHub

We will talk extensively about GitHub in this thesis and therefore need to establish a understanding of the platform and its features.

2.2.1 GitHub's role as a software-development platform

Organizations use GitHub as a platform to host and organize their source code. A wide array of implemented features allows to distribute and manage their software products. The concepts of GitHub organizations, teams and user based access control allows to collaborate and structure software projects. Architecting a closed source software project inside GitHub requires many security critical decisions to be made.

2.2.2 GitHub Organization

A GitHub organization is a shared account that allows multiple users to collaborate on projects. It provides a centralized space for managing repositories, teams, and access permissions. GitHub organizations can be used by organizations, open source projects, or any group of individuals who want to work together on software development. Within a GitHub organization, teams can be created to manage access to specific repositories and resources, allowing for fine-grained control over who can view or modify code. [19]

2.2.3 Security Guardrails

Guardrails are security measures implemented to prevent unauthorized access or actions within a system. In the context of GitHub, guardrails can include branch protection rules, required code reviews, and other policies that help ensure the integrity and security of the codebase. These measures can help prevent accidental or malicious changes to critical branches, enforce best practices for code quality and security, and provide an additional layer of defense against potential threats. [2]

2.2.4 GitHub Actions

GitHub Actions (GHA) is a platform integrated ecosystem within GitHub that allows developers to automate workflows directly in their code repositories. It enables users to create custom workflows that can be triggered by various events, such as code pushes, pull requests, or scheduled intervals. A workflow consist of one or more jobs, which are made up of steps that can run commands or use pre-built actions from the public GitHub Marketplace or custom developed actions that are stored privately. It is important to define the term GitHub Action (GHA) in the context of this thesis. A GHA is a reusable component that performs a specific task within a workflow. It can be thought of as a building block that can be combined with other actions to create complex workflows. GHA are developed in different ecosystems, but notably few ecosystems. Empirical studies show that most GHA are implemented in JavaScript, with a large share also being Docker-based Actions, and a smaller but still significant presence of Python implementations. [33]

2.2.5 GitHub Marketplace

The GitHub Marketplace is a platform that offers a wide range of third-party applications and tools that integrate with GitHub. It provides developers with access to various tools, including continuous integration, code quality analysis, security scanning, and more. The marketplace allows users to discover and manage these tools directly from their GitHub repositories, enhancing their development workflows.

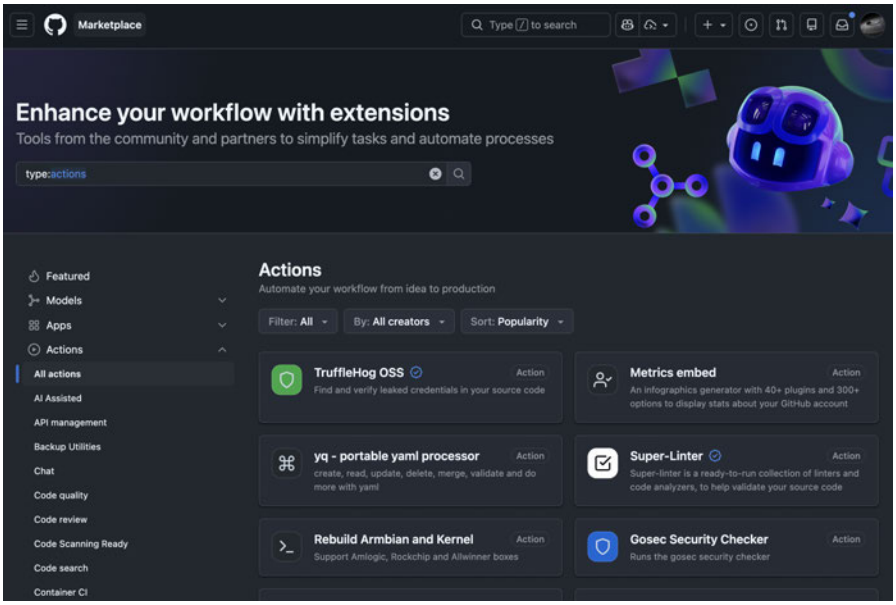

Figure 1: Screenshot of the GitHub Action browsing page from the Marketplace (accessed 23.02.2026, 18:44) [20]

2.2.6 The role of GitHub Actions in supply chains

GHA can be used to automate various stages of the software development lifecycle, including building, testing, and deploying applications. They can also be used to manage dependencies, perform security scans, and integrate with other tools and services. However, the use of GHA in supply chains introduces potential risks, especially when it comes to mutable dependencies. If a GHA is referenced in a mutable way (e.g., via a branch or tag), it can be exploited by attackers to introduce malicious code into the supply chain. This is particularly concerning when transitive dependencies are involved, as they may not be directly visible to developers and can be easily overlooked in security assessments. [7] [12]

2.3 Common Supply Chain Attack Vectors

2.3.1 Typosquatting

Typosquatting is a form of cyber attack where attackers register names (like Domain names or Package names) that are similar to legitimate ones, often by making common spelling errors. In the context of software development and supply chain security, typosquatting can happen when attackers create malicious repositories or packages with names that closely resemble popular libraries or tools. Developers who accidentally reference these malicious components instead of the intended ones can mistakenly introduce security risks into their projects, leading to potential security breaches or data theft. This type of attack has been exploited in the past on the GitHub Marketplace by scanning public repositorys for misspellings in GitHub Workflow definitions and then publishing corresponding projects on the Marketplace with malicious code. [47]

2.3.2 Repository takeover

A repository takeover is a security incident where an attacker gains unauthorized access to a software repository, often through compromised credentials or vulnerabilities in the hosting platform. Once the attacker has control over the repository, they can inject malicious code, modify existing code, or disrupt the development process. This can lead to significant security risks, especially if the compromised repository is widely used as a dependency in other projects, potentially resulting in a supply chain attack. [48]

2.4 Security Management Concepts

For our later analysis and discussion of mitigation strategies, it is important to understand some fundamental security management concepts. These concepts will help us evaluate the effectiveness of different mitigation strategies and understand how they can be applied in the context of CI/CD pipelines and supply chain security.

2.4.1 CIA Triad

The CIA Triad is a fundamental model in information security that represents the three core principles of security:

(i) *Confidentiality* refers to the protection of information from unauthorized access and disclosure.

(ii) *Integrity* ensures that information is accurate and unaltered, preventing unauthorized modifications.

(iii) *Availability* guarantees that information and resources are accessible to authorized users when needed.

In the context of software supply chain security, the CIA Triad can be used to evaluate and prioritize security measures to protect against risks associated with mutable dependencies and supply chain attacks. [44] [16]

2.4.2 Security Maturity Model

The security maturity model is a framework that helps organizations assess and improve their security posture continuously. It typically consists of multiple levels or stages, each representing a different level of security maturity. Organizations can use these models to identify gaps in their security practices, prioritize improvements, and track progress towards achieving a more secure environment.
Assesing the maturity level of an organization is a manual process that can be structured by using exisitng guidelines from like the PRISMA developed by NIST. [5] [11] In the context of CI/CD pipelines and supply chain security, a security maturity model can guide organizations in implementing best practices for dependency management, access control, and monitoring to mitigate risks associated with mutable dependencies and supply chain attacks.

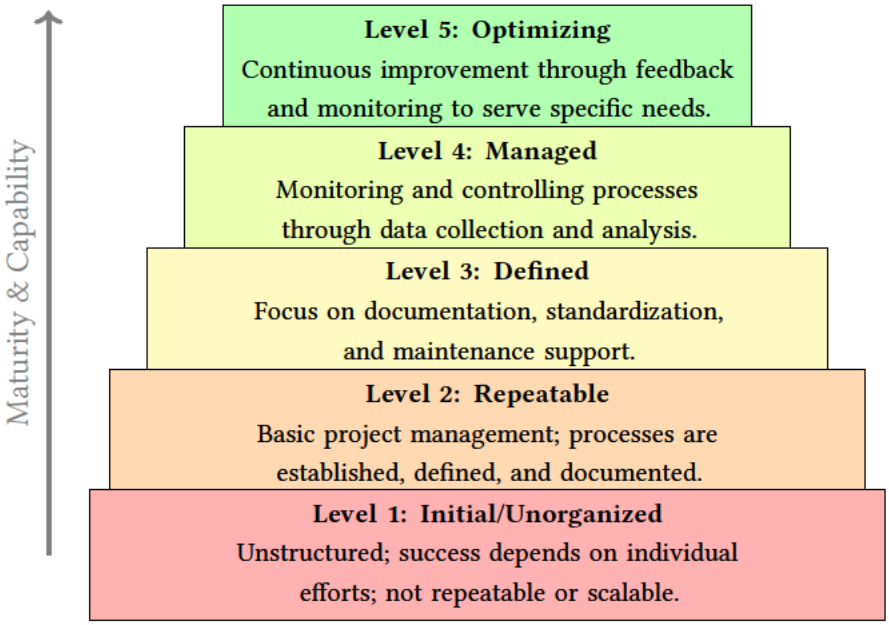

Figure 2: Security Maturity Hierarchy [46]

2.5 Transitive Dependencies

A transitive dependency is a dependency that is not directly referenced by a project but is required by one of its direct dependencies. In the context of software development, when a project depends on a library (direct dependency), and that library itself depends on another library (transitive dependency), the project indirectly relies on the transitive dependency. This can create complex dependency trees, where a project may have multiple layers of dependencies, making it challenging to manage and secure the software supply chain. [29]

2.5.1 Defining mutability

Mutability of depedencies is a critical factor in supply chain security. A mutable dependency is one that can change over time without a specific version or commit being referenced. This means that if a workflow references a dependency in a mutable

way (e.g., using a branch name or tag), it can be updated by the maintainer of that dependency, potentially introducing breaking changes or security weaknesses without the knowledge of the users of that workflow. In contrast, an immutable dependency is one that is fixed to a specific version or commit, ensuring that it cannot change unexpectedly and providing stability and security for the workflow. [39]

2.5.2 Semantic Versioning / semver

Semantic Versioning, commonly referred to as semver, is a versioning scheme for software that conveys meaning about the underlying changes in a release. It uses a three-part version number format: MAJOR.MINOR.PATCH. The MAJOR version is incremented for incompatible API changes, the MINOR version is incremented for backward-compatible functionality additions, and the PATCH version is incremented for backward-compatible bug fixes. This system helps developers and users understand the nature of changes in new releases and manage dependencies effectively. [36]

2.5.3 Dependency Pinning

Dependency pinning is a practice in software development where specific versions of dependencies are explicitly defined and locked in a project. This is done to ensure that the project uses a known and tested version of a dependency, rather than automatically pulling in the latest version, which may introduce breaking changes or security weaknesses. In the context of GitHub Actions, pinning can be achieved by referencing a specific commit SHA, which ensures immutability and stability of the action being used. [39]

2.5.4 How dependency pinning works in different ecosystems

Dependency pinning works differently in various ecosystems. As GHA can execute code in different languages and environments, the pinning mechanisms also differ. Below is a comparison of how dependency pinning is implemented across major software ecosystems:

Ecosystem	Locking Mechanism	Immutable Pinning Strategy
GitHub Actions [39]	None (Native)	Pin to Full 40-char **Commit SHA**
Docker [14]	docker-lock (External)	Pin to **Image Digest** (sha256:...)
Node.js (npm) [34]	package-lock.json	Integrity **Hashes** in lockfile + npm ci
Python (pip) [41] [27]	poetry.lock / uv.lock	**Lockfiles** with per-package hashes
Rust (Cargo) [4]	Cargo.lock	Automatic **Checksumming** of crates
Go (Modules) [21]	go.sum	**Transparency Log** (Sum DB) + Hashes
Java (Maven) [22]	gradle.lockfile (Gradle)	**GAV** coordinates + Checksum
Terraform [13]	.terraform.lock.hcl	**Provider Signatures** and checksums

Table 2: Dependency Pinning Mechanisms Across Major Software Ecosystems

As seen in the table the locking mechanisms usually depend on file based artifacts that are generated by the respective package manager. These artifacts contain the exact version and integrity information of the dependencies used in a project. In contrast, GitHub Actions implement locking by referencing a specific commit SHA, which is a unique identifier for a specific state of the codebase. This means that while other ecosystems rely on lockfiles to manage dependencies, GitHub Actions can achieve immutability by simply referencing the exact commit, ensuring that the action being used cannot change unexpectedly. Determening if a component is mutable therefore requires understanding the ecosystem specific pinning mechanisms and analyzing the workflow definitions accordingly.

3 Data Collection

For the analysis of the transitive dependency problem I collected data of 113 active repositories from inside a GitHub organization over a timespan of 58 days (17.12.2025 - 13.02.2026).
The intention of the data collection is to analyze the overall landscape inside an organization and create a baseline to evaluate the security risks of mutable dependencies in GHA usage. In the following security analysis chapter we can then analyze the collected data for security risks and examine two case studies derived from the collected data.

3.1 Methodology

The data collection was conducted by invoking the GitHub API via a peridocally scheduled AWS Lambda function. The Lambda function was configured to run every 24 hours and collect data on all workflow definitions and their transitive dependencies (which were compiled to a tree structure with with the edges representing the dependency relationships) of the GHA used in those workflows. The collected data was then stored in an AWS S3 bucket for further analysis.

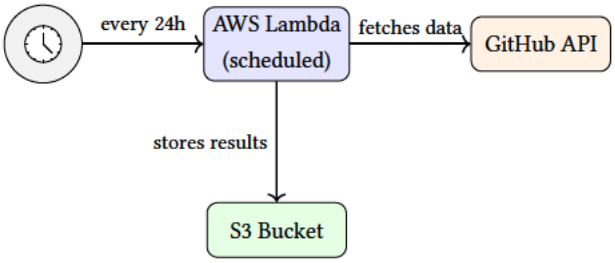

Figure 3: Automated data collection setup

3.2 Organization Setup

The GitHub organization from which the data was collected represents a medium-sized organization with around 400 members. Its platform technology stack is primarily structured around AWS and GitHub, with a strong focus on the Infrastructure as Code (IaC) paradigm. Teams own multiple repositories and projects, using GitHub Actions as part of their CI/CD pipelines. Some teams maintain internal services and projects, such as the cloud platform or device management. During the period in which I collected data, I had access to the repositories as a member of the platform team, which is primarily responsible for managing the cloud infrastructure.

3.2.1 GitHub Organization

The organization has two large GitHub organizations in total, one for the active core projects (*dev*) and the other for testing and experimentation (*playground*). The data collection was performed in the *playground* GitHub organization, which contains around 113 active repositories. There are multiple guardrails and policies in place from the platform team, such as branch protection rules or required code reviews. These rules are applied in order to provide a secure-by-default environment, an environment where developer teams can rely on secure defaults. In the following analysis, I won't take into account the GitHub organization policies and guardrails, as they are not directly related to the pinning problem and the security risks of mutable dependencies.

3.2.2 Amazon Web Services

Amazon Web Services (AWS) is the primary cloud provider used by the organization. Most of the infrastructure and the operations are running inside AWS. Software like Terraform is being used to manage the infrastructure as code and GHA then trigger Terraform deployments amongst other operations. Most of the code that is stored in the repositories is deployed through GHA into various AWS services.
Since GHA are being executed inside GitHub runners, there needs to a way to provide the necessary AWS credentials to those runners. There are multiple ways to approach

this, such as using GitHub Secrets or using OpenID Connect (OIDC) to assume AWS roles.

3.3 Collection Results

The data collection resulted in a comprehensive dataset comprising 6,038 action run records, which were aggregated into 3,752 depth summary records and 30,065 transitive edges across the 113 repositories. A complete, detailed tabular breakdown of the collection scope, dataset size, and execution statistics is provided in Table 9 in the appendix.

Crucially, the data highlights a severe lack of direct dependency pinning: the direct pinning rate was found to be only 3.61%, leaving an overwhelming 96.39% of direct dependencies unpinned and highly mutable. In contrast, transitive mutable edges accounted for a lower share of 13.88% of the total edges. The ecosystem analysis further revealed that the analyzed workflows rely heavily on the npm ecosystem, which makes up 92.65% of all transitive dependencies, while GitHub Actions themselves only account for 6.75% of transitives. Finally, the execution statistics indicate a relatively shallow dependency tree for CI/CD pipelines, with a mean transitive depth of 0.52 and a maximum depth of 2 levels.

With npm packages being the most prevalent type of transitive dependency, the relatively low rate of mutable transitive edges (13.88%) can be attributed to the native ecosystem defaults. Unlike GitHub Actions, where pinning is a manual opt-in process, npm projects automatically generate lockfiles *package-lock.json*. These files enforce strict versioning and record hashes for every transitive dependency, effectively 'freezing' the dependency graph by default. [34]

Study / Source	Population Size	Pinning Rate	Key Takeaway
This Study	113 Repos	**3.61%**	Results are consistent with established high-risk ecosystem trends.
Alvarez (2025) [3]	100 Repos	3.0%	Even security-focused projects fail to implement full SHA pinning.
Pan et al. (2024) [35]	324,672 Repos	1.56%	Large-scale analysis indicates widespread pinning issues.

Table 3: Comparison of Dependency Pinning Rates Across Recent Studies

The pinning rate of 3.61% is consistent with other studies that have analyzed the pinning problem in GitHub Actions, as shown in a comparison table above. The study by Pan et al. (2024) [35] analyzed a much larger dataset of 324,672 repositories and found a pinning rate of 1.56%, which indicates that the problem is widespread and not limited to a specific organization or set of repositories. The study by Alvarez (2025) [3] focused on security-focused projects and found a pinning rate of 3.0%, which suggests that even projects that are more aware of security issues still struggle with implementing proper pinning practices.

4 Threat Landscape and Case Studies

> Risk is positively correlated to the number of dependencies and complexity hides those dependencies. [18, 49]

In this chapter, I will analyze the discussed security risks of using GitHub Actions (GHA) in CI/CD pipelines, with first defining the security goals and then compiling a set of the most relevant security risks from the OWASP CI/CD Top 10 list. Finally, I will present two case studies that illustrate the best and worst-case scenarios of how supply chain attacks make use of those identified risks. The case studies are based on real-world incidents and gathered data from the organization.
This will provide a comprehensive understanding of the security implications and will set the stage for the subsequent discussion of mitigation strategies in Chapter 5 and it also ties back to our first Research Question (RQ1): "How do mutable and transitive dependencies in GitHub Actions impact the security of CI/CD pipelines?".

4.1 Definition of Security Goals

A clear definition of security goals is essential for a structured analysis of the case studies. The CIA Triad (Confidentiality, Integrity, Availability) as described in Section 2.4.1 is a widely accepted model for defining security goals in the context of information security. We can use that to systematically evaluate the security risks found in our case studies.

4.2 OWASP CI/CD Top 10 Security Risks

The Open Worldwide Application Security Project (OWASP) plays a crucial role in application security by providing standardized frameworks [45, 1]. Their "Top 10 CI/CD Security Risks" initiative identifies the most critical attack vectors in modern deployment environments.

Rather than discussing the entire list, this analysis focuses strictly on the risks that correlate directly with the use of mutable and transitive dependencies in GitHub Actions. By mapping these specific OWASP risks to the previously defined security goals, we create a lens through which the later case studies can be evaluated.

4.2.1 CICD-SEC-03: Dependency Chain Abuse

Depdendency Chain Abuse [9] refers to the exploitation of how systems fetch code dependencies. Attackers can compromise a dependency in the chain, which may be a direct or transitive dependency, to inject malicious code into the build process. This can lead to a loss of *integrity* if the compromised dependency is used in the build or deployment process, and it can also lead to a loss of *confidentiality* if sensitive information is exposed through the compromised dependency. Finally it can lead to a loss of *availability* if the compromised dependency causes the build or deployment process to fail. This risk identifies four main attack vectors:

(i) Dependency Confusion

(ii) Dependency Hijacking

(iii) Typosquatting

(iv) Brandjacking

Especially the first two attack vectors are relevant in the context of transitive dependencies, as they allow attackers to inject malicious code into the build process without altering the repository's source code.

4.2.2 CICD-SEC-08: Ungoverned Usage of Third-Party Services

The ungoverned usage of third-party services [10] significantly expands the CI/CD attack surface by the ease with which third-party services, such as GitHub Applications or marketplace plugins, can be granted access to repositories. Integrating these services often requires adding just one or two lines of code to a pipeline configuration file. While providing immediate value, this creates a governance challenge. Organizations frequently struggle to maintain continuous visibility over the permissions these third parties actually utilize, effectively executing unvetted, external code with highly privileged access to sensitive pipeline resources.

In the context of GitHub Actions, this ungoverned usage is primarily realized through mutable references (e.g., *@main* or floating tags) and their underlying, invisible transitive dependency trees. By bypassing strict pinning procedures, developers implicitly trust external publishers. If a orgnaization doesn't have a robust monitoring and auditing strategy in place, this could lead to unsecure code being executed in the CI/CD pipeline, which can result in a loss of *confidentiality*, *integrity* and *availability*.

4.3 Case Studies

With the previous sections providing a comprehensive overview of the security goals and risks associated with mutable dependencies in GitHub Actions, we can now inspect two case studies that illustrate the best and worst-case scenarios of how supply chain attacks can exploit these risks. These best-case is a collected example from the collected data and the worst-case is based on a real-world publically disclosed supply chain incident.

4.3.1 Best Case

If the pinning on a GHA is done correctly and all transitive dependencies are pinned as well, the risk of a supply chain attack is significantly reduced. The intention of the risk minimizing efforts were effective and the organization is protected against

the discussed attack vectors. Even if a transitive dependency gets compromised, the impact is limited to the specific version that is pinned and the attack can be mitigated by updating the pinning to a secure version.

```yaml
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@b4ffd... # v4.1.1

      - name: Run Android Emulator
        uses: reactivecircus/android-emulator-runner@1dcd0... #2.34.0
```

Figure 4: Best-case workflow snippet

We can see in the workflow snippet in Figure 4 that all dependencies are pinned to a specific commit SHA, which ensures immutability. When inspecting the transitive dependencies of the used actions, we can see that they are also immutable via the usage of lockfiles, that are native to the *npm* ecosystem.

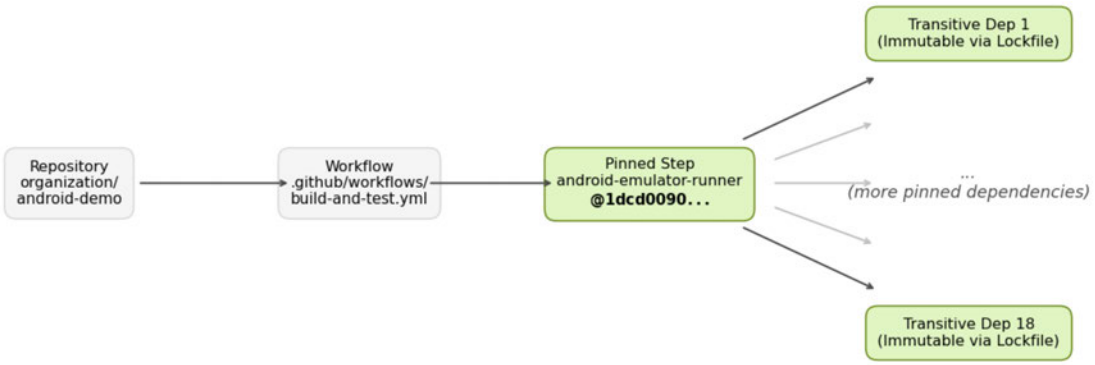

Figure 5: Best-case dependency graph

4.3.2 Worst Case

In the worst-case scenario, the GHA workflow contains immutable references to dependencies, and these dependencies have a large number of transitive dependencies that are in turn mutable. This introduces an invisible risk for supply chain attacks, as attackers can compromise any of the mutable dependencies in the chain to inject malicious code into the build process. The impact of such an attack can be severe, leading to a loss of *confidentiality*, *integrity* and *availability*.

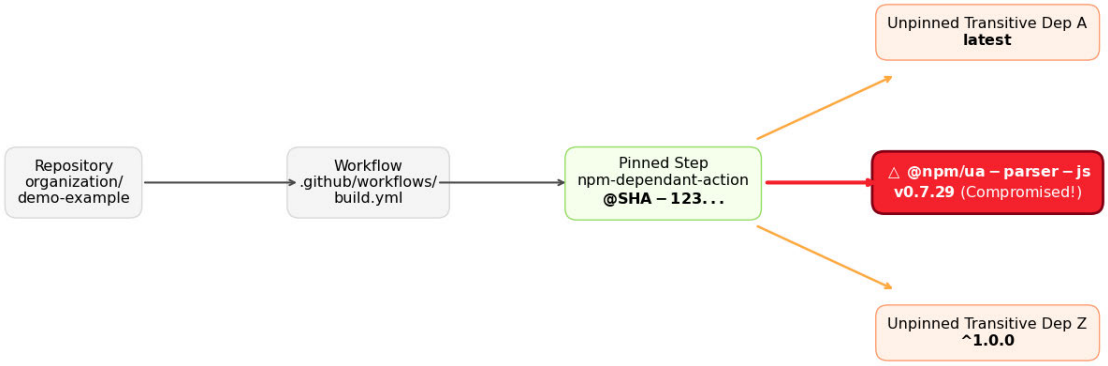

Figure 6: Worst-case dependency graph

In the dependency graph in Figure 6, we can see that the root node is being pinned to a specific version, but it has a large number of transitive dependencies that are mutable, because of a missing lockfile. This leads to the usage of a transitive dependency (*ua-parser-js, version 0.7.29* [32]) that is vulnerable to a known exploit. Looking at our risks from Section 4.2, this scenario is a textbook example of Dependency Chain Abuse, as the attacker can compromise the transitive dependency to inject malicious code into the build process, which can lead to a loss of *confidentiality*, *integrity* and *availability*. This scenario also illustrates the ungoverned usage of third-party services, as the mutable reference at the level of the transitive npm dependency allows for unvetted, external code to be executed with highly privileged access to sensitive pipeline resources.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@main

      - name: Execute NPM Dependent Action
        uses: organization/npm-dependant-action@v1 # Mutable tag, lacks lockfile
```

Figure 7: Worst-case workflow snippet

5 Mitigation

Upon analyzing the security risks of using GHA in CI/CD pipelines, the need for mitigation became apparent. I would like to present and discuss different mitigation strategies. Several strategies can be used in combination, but note that there may be redundancy in the effects since they all aim to minimize risk. Using multiple strategies and striving for holistic implementation correlates with a higher security maturity level, but it also creates a more restrictive environment for development teams. [37] Alot of this chapter discusses relevant topics that are related to RQ2 and we will find that there are many ways that users can mitigate the risks of mutable transitive dependencies, but there is no one-size-fits-all solution. The best approach depends on the specific context of the project, the resources available, and the risk tolerance of the organization.

5.1 Pin all transitive dependencies

Depending on the workflow structure, it may be possible to pin all referenced dependencies in the workflow definition. As this may only rarely the case, because most transitives are inherited further down call tree [23] [33], it may need an extra effort to make changes to the code repositories of the transitive dependencies.
Another possible solution would be to clone the original workflow, re-reference all transitives to either cloned or locally hosted instances of those transitives and enforce pinning along the way. This solution bears multiple problems.
(i) Some transitives may not be cloneable.
(ii) The process of mirroring an entire workflows dependency infrastructure can quickly develop to be a costly task in regards to storage and distribution.
(iii) In order to keep all mirrors updated, there need to be reliant mechanisms in place

that are ecosystem agnostic and enforce CI/CD security policies.
This approach can be very effective in a heavily managed system and might be the most suitable solution for all projects inside an organization. After evaluating critical pipelines, it might be a good option to implement this strategy for those pipelines and accept risk in others.

5.2 Run custom pinned GHA from private artifact store

Mirror all GHA that are used in a pipeline and validate all version bumps. If this path is chosen it needs to be verified that all transitive dependencies are either pinned or also mirrored and then pulled from respective mirror. This approach is similar to the previous one, though it could provide more flexibility and control. However, it also requires a significant effort to set up and maintain the artifact store and the mirroring process. Especially license compliance can be a challenge, as many open source components have specific licensing requirements that need to be adhered to when mirroring and redistributing them. This approach implies a significant amount of overhead and development teams might prefer to work with the readily available GHA on GitHub.

5.3 Introduce dependency cooldowns

A main attack vector is the automated updating of dependencies in workflows. Many supply chain attacks have been detected, reported and mitigated in less than a week.[48] [38] When enforcing a cooldown of for example 7 days for new updates to be pulled, it could mitigate a popular exploited mechanism.
Dependabot and Rennovate are two popular technologies used on GitHub for keeping versions up-to-date and both already implement the concept of dependency cooldown. This mitigation strategy seems easy to implement and effective for a specific purpose.

Though it won't address the structural weaknesses introduced by mutable transitive dependencies and there is the non-trivial requirement of pinning all actions.

5.4 Generate Lockfiles for GitHub Actions

Lockfiles are a common mechanism in software development to ensure immutability of dependencies. They record the exact version and often the hash of each dependency, including transitive ones. This allows for reproducible builds and mitigates the risk of unintentional updates.
He et al. [25] suggest that the implementation of lockfiles in the npm ecosystem could have mitigated the risks of mutable transitive dependencies. If a similar mechanism would be implemented for GHA, it could significantly reduce the risk of mutable dependencies. However, this would require efforts from the platform provider and might not be feasible in the short term for individual organizations, especially considering the associated costs and complexity. [42] [26]

5.5 Enforce deep-level pinning

The optional GitHub advanced security policies allow to enforce the pinning when referencing 3rd party actions, but these policies cannot enforce pinning for the transitive dependencies. If an organization sets up custom rules to scan all GHA used in critical pipelines for mutable transitive dependencies, it could strategize for dependencies on the deeper levels and ensure immutability for a significantly larger part of the dependency chain.

5.6 Introduction of a Scoring Model

Another complementary approach would be to conduct a static analysis of the Workflow definitions and summarize the outcome of that analysis in a score type deliverable. This is a common practice in the security domain [28] with frequently used scoring

systems like CVSS, EPSS or SLSA. Here i will introduce a scoring model that focuses on discussed risk vectors of transitive dependencies and their derived mutability.

6 Scoring Model

To effectively mitigate the supply chain risks discussed in the previous chapters, organizations require a systematic approach to quantify and evaluate their exposure. While identifying unpinned root dependencies is a straightforward process, uncovering the hidden risks of mutable transitive dependencies remains a complex challenge. To address this visibility gap, this chapter introduces the **Mutable Dependency Risk Score (MDRS)**.

Designed as a comprehensive evaluation framework, the MDRS enables a simpler assessment of CI/CD pipeline vulnerabilities through a single numeric value. By analyzing the dependency tree, the model focusses on revealing the cases where there is a discrepancy between pinned immutable root actions and their potentially mutable underlying components. Ultimately, the MDRS aids development and security teams in prioritizing problematic workflow usage and streamlining remediation efforts across entire repositories. To ensure seamless integration into existing developer workflows, the corresponding assessment tool implementing this model can be natively executed as a GitHub Action and provide automated feedback.

6.1 Goal

The theoretical foundation of the MDRS is built on a penalty-based evaluation mechanism. Instead of rewarding secure configurations, the model assumes a baseline risk of zero and penalizes deviations from immutability. Every identified dependency within a workflow definition is systematically evaluated against specific security metrics. High-risk practices—such as referencing a component via a highly mutable branch

or inheriting a mutable transitive dependency from an otherwise securely pinned root—incur distinct penalty values.

This approach produces a cumulative risk score. By assessing the metrics for every individual dependency and aggregating their penalties, the model calculates a final risk score for the overall workflow. This cumulative design reflects the reality of supply chain attacks: a pipeline is only as secure as its weakest, most mutable link.

6.1.1 Metrics

This approach to metrics draws a lot of inspiration from the CVSS scoring system. The CVSS systems uses multiple metrics to calculate a risk score for vulnerabilities[30]. The definition of those metrics implements a qualitative analysis of the attack vector at hand and assigns values based on that. The metrics used for **MDRS** reflect this philosophy, as in that the chosen metrics can be determined by clear defined categories.

Mutability

The Mutability metric captures how strictly a dependency reference is locked to a specific state. Rather than depending on ecosystem-specific mechanisms, the metric abstracts the concept into three universally applicable classes of mutability:

- **Absolute Immutability (0.0):** The reference is cryptographically or strictly locked to an exact state (e.g., via checksums or digests). Modifications are impossible without changing the reference itself.
- **Bounded Mutability (0.5):** The dependency is mutable, but updates are tied to an explicit release process and restricted by version boundaries (e.g., semantic versioning). This provides controlled, predictable changes.
- **Unbounded Mutability (1.0):** The component's state can change at any time without warning or version increment (e.g., pulling from active branches, floating tags, or raw URLs). This uncontrolled state offers the highest attack surface.

This classification is deliberately ecosystem-agnostic. Whether a reference points to a commit SHA in a Git repository, a digest in a container registry, or a checksum in a lockfile, the decisive factor is the *degree of control* the reference provides over the resolved artifact. Table 4 summarizes the resulting classification.

Mutability Class	Value
Absolute Immutability	0.0
Bounded Mutability	0.5
Unbounded Mutability	1.0

Table 4: Mutability Metric Classification

Transitivity

This is the central focus of the scoring system. A significant risk exists between if a component or workflow reference is pinned to be immutable, but a transitive component stemming from that immutable parent object is mutable itself. If this happens, the

maintainers' efforts to mitigate the risk are relativized, and the newly introduced risk may be invisible to them. *For this reason, I choose a higher weighting value of 2 for this metric.*

Importantly, direct dependencies at Depth 1 that are referenced via a mutable mechanism (Branch or Semver) must never receive a Transitivity value of 0.0 (Fully Pinned). Since their own reference is already mutable, they inherently carry transitive volatility and must be assigned at least a value of 0.5 (Visible Risk).

Classification	Value
Fully Pinned (Transitive is pinned)	0.0
Visible Risk (Mutable Transitive from a Mutable Root)	0.5
Invisible Risk (Mutable Transitive from a Pinned Root)	1.0

Table 5: Transitivity Metric Classification

Dependency Source

The Dependency Source is relevant in regards of the source's capability to supply the consumer with relevant metadata about the component. If a component is directly downloaded from a URL with curl, without a Checksum available, it implies a higher risk then if a component is referenced in a package-lock.json with a corresponding checksum.

Source	Value
Local	0.0
Private / Internal Registry	0.2
Public Managed Registry (e.g., NPM, PyPI)	0.5
Public Git Repository	0.8
Direct Retrieval (e.g. *curl*)	1.0

Table 6: Dependency Source Metric Classification

6.1.2 Risk score calculation

After determining the values for every component in the dependency tree, the risk scores can be calculated and summed into a total risk score, that represents the workflow definition.

$$\text{BaseScore} = \frac{100}{N \cdot Metric_{\text{max}}} \cdot \sum_{i=1}^{N} R_i \tag{6.1}$$

$$R_i = (w_M \cdot M_i) + (w_T \cdot T_i) + (w_S \cdot S_i) \tag{6.2}$$

Where:

- R_i denotes the risk score of action or workflow i.
- N is the total number of dependencies analyzed in the workflow.
- $Metric_{\text{max}}$ is the maximum possible score for a single dependency, calculated as $w_M \cdot 1.0 + w_T \cdot 1.0 + w_S \cdot 1.0 \ = \mathbf{4}$ based on the highest values in each metric category.
- M_i represents the **Mutability** metric for action i, derived from Table 4.
- T_i represents the **Transitivity** dependency metric for action i, derived from Table 5.
- S_i represents the **Dependency Source** metric for action i, derived from Table 6.

Weighting factors:

- $w_M = 1$ Mutability weighting factor as discussed in 6.1.1.
- $w_T = 2$ Transitivity weighting factor as discussed in 6.1.1.
- $w_S = 1$ Dependency source weighting factor as discussed in 6.1.1.

Invisible Risk Override:
To prevent the dilution of critical risks in large dependency graphs, an override mechanism is applied. If any component i in the pipeline is classified as Invisible Risk ($T_i = 1.0$), the final score is lower bounded by a minimum threshold:

$$\text{TotalRiskScore} = \max\big(\text{BaseScore},\ 75\big) \quad \text{if } \exists\, i : T_i = 1.0 \tag{6.3}$$

Otherwise, TotalRiskScore $=$ BaseScore.

6.2 Risk Levels

Criticality levels are defined based on the Total Risk Score to provide an intuitive categorization of workflow security. Considering the qualitative value that these risk levels add to the model, they help us to follow the goal of RQ2.

The following thresholds are proposed for categorizing the Total Risk Score into discrete risk levels:

- **Low Risk (0-25):** The workflow has minimal exposure to mutable dependencies, with strong immutability controls in place.
- **Moderate Risk (26-50):** The workflow contains some mutable dependencies, but they are generally well-managed and do not pose an immediate threat.
- **High Risk (51-74):** The workflow has significant mutable dependencies, including some with transitive volatility, indicating a substantial attack surface that requires attention.
- **Critical Risk (75-100):** The workflow has multiple mutable dependencies and at least one instance of invisible risk, necessitating urgent remediation.

6.3 Model Calibration and Iterative Refinement

The theoretical formulation of the MDRS, as presented in the preceding sections, was subject to empirical validation during the practical implementation of the MDRS model. Applying the model to real-world repositories, including the gathered dataset of 113 repositories described in chapter 3, revealed two systematic flaws in the initial scoring logic. The iterative refinement process that followed exemplifies a feedback loop between theoretical modeling and practical application, whereby observed anomalies in scoring outputs prompted corrections to the underlying metric definitions.

6.3.1 Addressing the Dilution Effect

The first deficiency manifested when analyzing large dependency graphs. In repositories with a substantial number of dependencies, the averaging mechanism inherent in the BaseScore formula (see Equation 6.1) caused individual high-severity findings to be diluted by the surrounding mass of low-risk components. A workflow containing, for instance, 50 dependencies—49 of which are fully pinned and 1 of which constitutes an Invisible Risk ($T_i = 1.0$) would produce a misleadingly low total score, despite having a critical vulnerability.

To correct this, a *Invisible Risk Override* was introduced (see Equation 6.3). Whenever the MDRS model detects at least one dependency classified as Invisible Risk ($T_i = 1.0$), the TotalRiskScore is lower bounded to a minimum of 75, placing the workflow in the Critical Risk category. This lower bounding mechanism ensures that the most dangerous class of findings (mutable transitive dependencies) hidden behind pinned roots cannot be masked by dilutive averages. This approach mirrors the *highest watermark* principle widely established in vulnerability management frameworks [44]. When evaluating an entire asset using systems like CVSS, the overall risk is not averaged across all components. Instead, the presence of a single critical vulnerability dictates the maximum severity of the entire system.

6.3.2 Resolving the Leaf Node Fallacy

The second deficiency was identified during initial test runs against repositories with shallow dependency trees. Under the original model, a mutable root dependency (Depth 1) that did not itself declare any transitive children, a so-called *leaf node*, received a Transitivity score of 0.0 (Fully Pinned). This occurred because the absence of transitive dependencies was interpreted as the absence of transitive risk.

This classification proved to be a logical fallacy. A mutable reference at Depth 1, whether it currently resolves further dependencies or not, constitutes an inherently unstable foundation for the pipeline. Its referenced artifact can change at any time, potentially introducing new transitive dependencies in future resolutions that did not exist at the time of analysis. Assigning a Transitivity penalty of zero to such a component effectively rewards the lack of visibility rather than penalizing it.

The correction enforces the following constraint: any direct dependency (Depth 1) that is referenced via a mutable mechanism (Bounded or Unbounded Mutability) must receive a minimum Transitivity classification of 0.5 (Visible Risk). This reflects the assumption that the risk is at least *visible* to the developer, the mutable reference is explicitly declared in the workflow definition, even if no transitive children are currently observed. The mutability of the reference itself is sufficient to warrant a non-zero penalty, as discussed in Table 6.1.1.

Both corrections were validated against the full dataset and resulted in score distributions that more accurately reflect the observed risk profiles of the analyzed repositories.

6.4 Limitations of the Model

While the MDRS provides a structured framework for quantifying supply chain risks, it is important to acknowledge its limitations. The model relies on static analysis of workflow definitions and their declared dependencies, which may not capture dynamic behaviors or runtime conditions that could affect security. Additionally, the weighting factors and metric classifications, while grounded in theoretical rationale, are ultimately heuristic and may require further empirical validation across diverse ecosystems and organizational contexts.

Finally, the model does not currently account for the potential impact of vulnerabilities within dependencies, focusing instead on structural risk factors; integrating vulnerability severity assessments could enhance its predictive power.

Another limitation is that the model currently only evaluates dependencies up to a certain depth (e.g., Depth 2), which may overlook risks present in deeper levels of the dependency graph. Future iterations of the model could explore methods for efficiently assessing deeper transitive dependencies without incurring prohibitive computational overhead. The source metric in Table 6 is also not entirely free of ambiguity, as the security implications of a dependency source can vary widely based on specific contexts and configurations. For example, a public Git repository could be well-maintained and secure, while another could be highly volatile and risky. Therefore, while the source metric provides a general guideline, it should be interpreted with caution and supplemented with additional context-specific information when making security decisions.

Lastly, while the MDRS offers a valuable tool for assessing risk, it should not be viewed as a standalone solution. It is most effective when used in conjunction with other security practices, such as regular vulnerability scanning, code reviews, and adherence to secure development lifecycle principles. The MDRS can help prioritize areas of concern but does not replace the need for comprehensive security strategies.

7 Implementation

With the theoretical framework of the MDRS established, the next step was to translate this model into a practical tool that could be applied to real-world repositories. We developed the **MDRS Checker**, a TypeScript/Node.js application designed to analyze GitHub Actions workflows and compute the MDRS for each identified dependency. The goal with this tool is to gain insights into how the theoretical model performs in practice. The implementation process revealed several challenges and necessitated iterative refinements to the model, which will be discussed in detail in section 6.3.

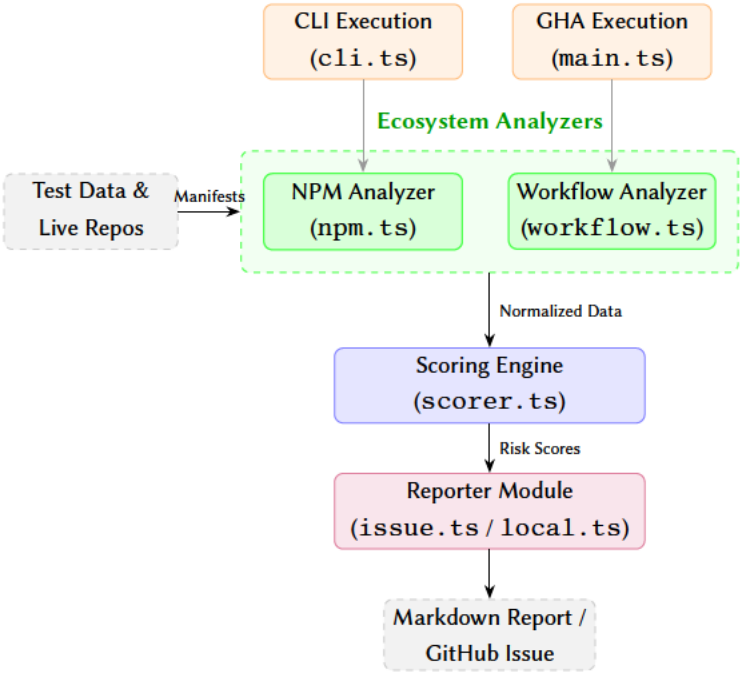

Figure 8: Software architecture of the MDRS Checker

7.1 Ecosystem Analyzers

To compute the MDRS we need to gather artifacts and evaluate them according to the defined metrics. The Ecosystem Analyzers are responsible for this task. They parse the relevant manifest files (e.g., ‘package.json‘ and ‘package-lock.json‘ for npm, ‘action.yml‘ for GitHub Actions), resolve the dependencies, and provide the necessary data to the Scoring Engine. The analyzers are designed to perform a recursive resolution of dependencies, meaning that they not only analyze the top-level dependencies but also resolve further down the dependency tree to identify transitive dependencies. This is crucial for accurately assessing the discussed risk. After parsing the manifest files, the analyzer then returns a object back to the Scoring Engine that contains all the relevant information about the dependencies, including their versions, whether they are pinned or mutable, and their position in the dependency tree. This information is then used by the Scoring Engine to calculate the MDRS for each dependency and ultimately for the entire workflow.

Code Block 7.1: Exemplary return object of an Ecosystem Analyzer

```
return {
  name: ref.repo,
  mutability,
  transitivity,
  source,
  depth: 1,
  ecosystem: "github-actions",
};
```

The current implementation includes two analyzers: one for npm packages and another for GitHub Actions. We chose these two ecosystems because they are widely used and allow us to analyze the case studies presented in chapter 4. We defaulted the analyzers to a search depth of 2, meaning that they will resolve dependencies up to two levels deep. This was a deliberate choice to balance the thoroughness of the analysis with computability, as deeper resolution can lead to significantly more transitive dependencies and therefore to more unimplemented ecosystems.

7.2 Dual Execution Modes

The MDRS Checker was designed to be flexible in its execution, allowing it to be used in different contexts. It can be executed as a GitHub Action (GHA) or as a Command Line Interface (CLI) tool.

7.2.1 GitHub Actions Integration (GHA)

Integrating the MDRS Checker as a GitHub Action allows it to be seamlessly incorporated into CI/CD pipelines. It can be configured to run automatically on desired triggers and then report its findings directly within the GitHub repository. Implmenting the MDRS Checker as a GHA helped to run large scale automated scans and gather performance data to calibrate the model further. The ability to integrate the MDRS model on a large scale also provide further answers to RQ2, as it allowed us to automate the analysis of a specified repositories and gather insights on the prevalence of high-risk patterns in real-world scenarios.

7.2.2 Command Line Interface (CLI)

The CLI tool allows users to run the MDRS Checker locally on their machines. This is particularly useful for testing and analyzing the output in a more interactive way. Running the MDRS Checker as a CLI tool also enables users to scan repositories that are not hosted on GitHub or to analyze specific branches or commits that may not be easily accessible through the GHA. Further, it allows scanning a whole GitHub organization by using the `scan-org` command and passing the flag `--org` and the organization name, as shown below.

```
1 mdrs-cli scan-org --org my-org --token ghp_... --max-depth 2
```

Code Block: Example CLI usage

7.3 Test Data Setup and Score Calibration

To validate the implementation of the MDRS Checker and to calibrate the scoring model, we created multiple sets of test data. Most notably, we created our best-case scenario and another representing our worst-case scenario from chapter 4. The best-case scenario includes a workflow with all dependencies as well as their transitively resolved dependencies pinned. The worst-case scenario includes a workflow with a pinned dependency but with a mutable transitive that is associated with a publically disclosed and exploited vulnerability. The worst case specifically aimed to trigger the invisible risk override. We also had test data that modeled a case where multiple ecosystems were involved, to validate the checker's ability to handle complex scenarios and a case which only included mutable dependencies.
Using these test cases, we were able to verify that the recursive resolution of dependencies is working correctly and that the calculated MDRS scores align with our expectations based on the defined model. This process of model calibration was crucial for ensuring that the MDRS provides meaningful insights into the security risks associated with mutable dependencies in GitHub Actions workflows.

7.4 Applying the Model to our Case Studies

To validate the theoretical scoring model and the recursive resolution logic of the MDRS Checker, the tool was executed against the two case studies defined in Chapter 4. The generated Markdown reports provide a transparent breakdown of the calculated risk scores, confirming that the tool accurately translates pipeline configurations into actionable security metrics.

7.4.1 Evaluating the Best-Case Scenario

When executed against the *best-case* repository, the MDRS Checker correctly identifies the strict governance applied to both the GitHub Actions and the underlying npm ecosystem. Because all top-level actions are cryptographically pinned via SHAs and all

transitive dependencies are secured via lockfiles, the tool assigns the lowest possible penalty values for mutability and transitivity.

As shown in Listing 7.4.1, the calculation results in a minimal Total Risk Score of **12.5 / 100** (Low Risk). The breakdown demonstrates that every dependency, regardless of its depth, achieves a perfect score for *Mutability* (0) and *Transitivity* (0). The remaining 12.5 points stem entirely from the baseline risk of fetching code from a public third-party registry (*Source* penalty of 0.5).

Metric	Value
Total Risk Score	12.5 / 100
Risk Level	Low

Package	Version	D.	Mutability	Transitivity
`actions/checkout`	`b4ffde6`	1	absolute-immutability (0)	fully-pinned (0)
`express`	`4.18.2`	2	absolute-immutability (0)	fully-pinned (0)

Table 7: Excerpt from the generated report for the best-case scenario

7.4.2 Evaluating the Worst-Case Scenario

The *worst-case* test data models the invisible risk of compromised transitive dependencies, mirroring the *ua-parser-js* [32] supply chain attack. In this scenario, the top-level GitHub Actions (*actions/checkout* and *actions/setup-node*) are perfectly pinned to a SHA. A superficial scanner would classify this workflow as secure.

However, the MDRS Checker's recursive analyzer detects that a transitive dependency (*ua-parser-js*) is imported using a mutable floating range (*^0.7.28*) without a locking mechanism. As seen in table subsection 7.4.2, this immediately triggers the invisible risk override mechanism. Instead of allowing the secure top-level actions to mathematically dilute the overall score, the tool applies the lower bounded penalty.

This elevates the Total Risk Score to **75 / 100** (Critical Risk). The report explicitly surfaces *ua-parser-js* at Depth 2 as the root cause, assigning it an *Invisible Risk* transitivity penalty of 1. This output successfully proves that the implemented tool and the underlying

MDRS model can detect supply chain weaknesses that derive from mutable transitive dependencies, even when the direct dependencies are securely pinned.

Invisible Risk Penalty Applied

Metric	Value
Total Risk Score	75 / 100
Risk Level	Critical

Package	Version	D.	Mutability	Transitivity
`ua-parser-js`	`^0.7.28`	2	bounded-mutability (0.5)	invisible-risk (1)

Table 8: Excerpt from the generated report for the worst-case scenario

7.5 Limitations and Future Work

As with any implementation, there are limitations to the current version of the MDRS Checker. One of the main limitations is the depth of dependency resolution. Currently, the analyzers are configured to resolve dependencies up to a depth of 2, which means that deeper transitive dependencies may not be analyzed. This was a deliberate choice to balance the thoroughness of the analysis with computability, but it does mean that some risks associated with deeper transitive dependencies may not be captured. Future work could involve optimizing the analyzers to allow for deeper resolution without significantly impacting performance.

As discussed in section 7.1, the MDRS Checker currently includes two ecosystem analyzers: one for npm packages and another for GitHub Actions. Many more ecosystems aren't taken into account, so a broader validation of the models robustness across different contexts is still pending. Finally, while the MDRS Checker provides a score that reflects the risk associated with mutable dependencies, it does not currently provide specific recommendations for mitigation. Future work could involve integrating mitigation strategies into the tool, providing users with actionable insights on how to reduce their MDRS scores and improve their security posture.

8 Summary

When using GitHub Actions for automating software development processes, it is crucial to understand the security implications of the dependencies involved. We have identified that mutable and transitive dependencies pose significant risks, as they can be exploited by attackers to compromise the toolchain whilst taking cloak in the complexity of the dependency graph.
To analyze this problem, we collected GitHub Action usage data from inside a organization and evaluated the prevalence of mutable and transitive dependencies. The mutability characteristics of our dataset was consistent with findings in related studies. We then took a sample from the collected data and added another sample of a known reported vulnerability, to conduct a case study with defined security goals and risk vectors. Based on the insights from this analysis, we proposed several mitigation strategies and introduced a scoring model to assess the discussed risks of transitive dependencies in GitHub Actions.
With the introduction of the scoring model, we developed a assesment strategy that can be used to identify high-risk dependency patterns and prioritize mitigation efforts. The models focus was directed towards identifying mutable transitive dependencies, that were introduced despite pinning efforts from the users. This focus was chosen because of the hypothesis that dependency pinning can lead to a false sense of security with transitive dependencies being the main attack vector. After we established the theoretical foundation of the scoring model, we implemented a prototype that can be used to conduct static analysis of GitHub Actions and compute the corresponding scores. Using this tool we could validate that the scoring model can effectively identify risks in real-world scenarios, as we applied the tool to the previously analyzed case studies. Whilst the scoring model and the implemented tool can be used to identify high-risk dependency patterns, it is important to note that there is no one-size-fits-all

solution for mitigating the risks of mutable transitive dependencies in GitHub Actions. The best approach depends on the specific context of the project, the resources available, and the risk tolerance of the organization.
Future work could explore implementations of additional ecosystems, as well as the development of more comprehensive mitigation strategies like lockfiles for GitHub Actions. Additionally, further research could be conducted to evaluate the effectiveness of the proposed scoring model in different contexts and with larger datasets.

Bibliography

[1] *About the OWASP Foundation | OWASP Foundation*. https://owasp.org/about/. (Visited on Mar. 17, 2026).

[2] *Actions Pull_request_target and Environment Branch Protections Changes - GitHub Changelog*. Nov. 2025. (Visited on Dec. 3, 2025).

[3] Adan Alvarez. *Medium - GitHub Actions and the Pinning Problem: What 100 Security Projects Reveal*. Mar. 2025. (Visited on Oct. 9, 2025).

[4] *Appendix: Glossary - The Cargo Book*. https://doc.rust-lang.org/cargo/appendix/glossary.html?highlight=lock#lock-file. (Visited on Feb. 19, 2026).

[5] L&Co Staff Auditors. *Security Maturity Models: Levels, Assessment, and Benefits*. Jan. 2023. (Visited on Feb. 16, 2026).

[6] Yaron Avital. *Unpinnable Actions: How Malicious Code Can Sneak into Your GitHub Actions Workflows*. Aug. 2023. (Visited on Feb. 24, 2026).

[7] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. "Paper - Automatic Security Assessment of GitHub Actions Workflows". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED'22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 37–45. ISBN: 978-1-4503-9885-5. DOI: 10.1145/3560835.3564554. (Visited on Aug. 22, 2025).

[8] Marc Campbell. *Keep Your Dependencies Secure and Up-to-Date with GitHub and Dependabot*. Jan. 2019. (Visited on Feb. 26, 2026).

[9] *CICD-SEC-3: Dependency Chain Abuse | OWASP Foundation*. https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-03-Dependency-Chain-Abuse. (Visited on Mar. 17, 2026).

[10] *CICD-SEC-8: Ungoverned Usage of 3rd Party Services | OWASP Foundation*. https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-08-Ungoverned-Usage-of-3rd-Party-Services. (Visited on Mar. 19, 2026).

[11] Information Technology Laboratory Computer Security Division. *Security Maturity Levels - Program Review for Information Security Assistance | CSRC | CSRC*. https://csrc.nist.gov/Projects/program-review-for-information-security-assistance/security-maturity-levels. Dec. 2016. (Visited on Feb. 16, 2026).

[12] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. "A Preliminary Study of GitHub Actions Dependencies". In: (). (Visited on Nov. 15, 2025).

[13] *Dependency Lock File (.Terraform.Lock.Hcl) - Configuration Language | Terraform | HashiCorp Developer*. https://developer.hashicorp.com/terraform/language/files/dependency-lock. (Visited on Feb. 19, 2026).

[14] *Docker-Lock Module - Github.Com/Safe-Waters/Docker-Lock - Go Packages*. https://pkg.go.dev/github.com/safe-waters/docker-lock#section-readme. (Visited on Feb. 19, 2026).

[15] ENISA. *Threat Landscape for Supply Chain Attacks | ENISA*. https://www.enisa.europa.eu/publications/threat-landscape-for-supply-chain-attacks. Nov. 2025. (Visited on Feb. 26, 2026).

[16] *Executive Summary — NIST SP 1800-26 Documentation*. https://www.nccoe.nist.gov/publication/1800-26/VolA/index.html. (Visited on Mar. 17, 2026).

[17] Göthe Universität Frankfurt. *Kennzeichnung Der Nutzung von KI-Tools*. https://www.starkerstart.uni-frankfurt.de/177578922.pdf. Oct. 2025. (Visited on Mar. 5, 2026).

[18] Dan Geer. *A Rubicon*. https://www.hoover.org/research/rubicon. Feb. 2018. (Visited on Feb. 11, 2026).

[19] GitHub. *GitHub Docs - About Organizations*. https://docs-internal.github.com/en/organizations/collaborating-with-groups-in-organizations/about-organizations. (Visited on Feb. 25, 2026).

[20] GitHub. *GitHub Marketplace - Actions*. https://github.com/marketplace?type=actions. (Visited on Feb. 23, 2026).

[21] *Go Modules Reference - The Go Programming Language*. https://go.dev/ref/mod. (Visited on Feb. 19, 2026).

[22] *Gradle Locking Versions (Gradle.Lockfile)*. https://docs.gradle.org/current/userguide/dependency_locking.html. (Visited on Feb. 19, 2026).

[23] Asi Greenholts. *Palo Alto - The GitHub Actions Worm: Compromising GitHub Repositories Through the Actions Dependency Tree*. Sept. 2023. (Visited on Oct. 9, 2025).

[24] HAW Hamburg. *Leitfaden Zum Einsatz von KI-Tools in Lehre*. https://www.haw-hamburg.de/fileadmin/ASD/ZLL-KI/Leitfaden_zum_Einsatz_von_KI-Tools_in_Lehre__Froschung_und_Verwaltung_2025_07_28.pdf. June 2025. (Visited on Mar. 5, 2026).

[25] Hao He, Bogdan Vasilescu, and Christian Kästner. "Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks". In: *Replication Package for "Pinning is Futile: You Need More Than Local Dependency Versioning to Defend Against Supply Chain Attacks"* 2.FSE (June 2025), FSE013:266–FSE013:289. DOI: 10.1145/3715728. (Visited on Feb. 26, 2026).

[26] *Lack of "Lock File" for Github Actions Is Making the Ecosystem Vulnerable to Supply Chain Attacks · Community · Discussion #154112*. https://github.com/orgs/community/discussions/154112. (Visited on Feb. 26, 2026).

[27] *Libraries | Main | Documentation | Poetry - Python Dependency Management and Packaging Made Easy*. https://python-poetry.org/docs/main/libraries#lock-file. (Visited on Feb. 19, 2026).

[28] Engla Ling, Robert Lagerström, and Mathias Ekstedt. "A Systematic Literature Review of Information Sources for Threat Modeling in the Power Systems Domain". In: *Critical Information Infrastructures Security*. Ed. by Awais Rashid and Peter Popov. Cham: Springer International Publishing, 2020, pp. 47–58. ISBN:

978-3-030-58295-1. DOI: 10.1007/978-3-030-58295-1_4. (Visited on Jan. 22, 2026).

[29] Darren Meyer. *Demystifying Transitive Dependency Vulnerabilities | Blog | Endor Labs*. https://www.endorlabs.com/learn/demystifying-transitive-dependency-vulnerabilities-sca. May 2024. (Visited on Feb. 11, 2026).

[30] *NVD - Vulnerability Metrics*. https://nvd.nist.gov/vuln-metrics/cvss#. (Visited on Mar. 11, 2026).

[31] Marc Ohm et al. *Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks*. May 2020. DOI: 10.48550/arXiv.2005.09535. arXiv: 2005.09535 [cs]. (Visited on Feb. 25, 2026).

[32] Sebastian Olsson. *The Supply Chain Attack of UAParser.Js Npm Package*. Oct. 2021. (Visited on Mar. 19, 2026).

[33] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. "Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. Lisbon Portugal: ACM, Apr. 2024, pp. 692–703. ISBN: 979-8-4007-0587-8. DOI: 10.1145/3643991.3644899. (Visited on Nov. 24, 2025).

[34] *Package-Lock.Json | Npm Docs*. https://docs.npmjs.com/cli/v9/configuring-npm/package-lock-json. (Visited on Feb. 19, 2026).

[35] Ziyue Pan et al. "Paper - Ambush from All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines". In: *IEEE Transactions on Dependable and Secure Computing* 21.1 (Jan. 2024), pp. 403–418. ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/TDSC.2023.3253572. arXiv: 2401.17606 [cs]. (Visited on Oct. 6, 2025).

[36] Tom Preston-Werner. *Semantic Versioning 2.0.0*. https://semver.org/. (Visited on Feb. 16, 2026).

[37] Sk Golam Saroar and Maleknaz Nayebi. “Paper - Developers’ Perception of GitHub Actions: A Survey Analysis”. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’23. New York, NY, USA: Association for Computing Machinery, June 2023, pp. 121–130. ISBN: 979-8-4007-0044-6. DOI: 10.1145/3593434.3593475. (Visited on Aug. 22, 2025).

[38] Christian Schneider. *Dependency Cooldowns: A Simple Supply Chain Fix.* https://christian-schneider.net/blog/dependency-cooldowns-supply-chain-defense/. (Visited on Feb. 11, 2026).

[39] *Secure Use Reference.* https://docs.github.com/en/actions/reference/security/secure-use. (Visited on Feb. 16, 2026).

[40] Laura Shiff. *BMC Software Blog - 2022 DevOps Trends.* https://www.bmc.com/blogs/devops-trends/. Oct. 2021. (Visited on Aug. 23, 2025).

[41] *Structure and Files | Uv.* https://docs.astral.sh/uv/concepts/projects/layout/#the-lockfile. (Visited on Feb. 19, 2026).

[42] *Support "Lock File" Equivalent for GitHub Actions · Issue #2195 · Actions/Runner.* https://github.com/actions/runner/issues/2195. (Visited on Feb. 26, 2026).

[43] Synopsis. *New Synopsys Report Finds 74% of Codebases Contained High-Risk Open Source Vulnerabilities, Surging 54% Since Last Year.* https://investor.synopsys.com/news/news-details/2024/New-Synopsys-Report-Finds-74-of-Codebases-Contained-High-Risk-Open-Source-Vulnerabilities-Surging-54-Since-Last-Year/default.aspx. 2024. (Visited on Feb. 26, 2026).

[44] National Institute of Standards and Technology. *Standards for Security Categorization of Federal Information and Information Systems.* Tech. rep. Federal Information Processing Standard (FIPS) 199. U.S. Department of Commerce, Feb. 2004. DOI: 10.6028/NIST.FIPS.199. (Visited on Mar. 20, 2026).

[45] *What Is OWASP? What Is the OWASP Top 10?* https://www.cloudflare.com/learning/security/threats/owasp-top-10/. (Visited on Mar. 17, 2026).

[46] *What's Your Security Maturity Level? – Krebs on Security*. Apr. 2015. (Visited on Feb. 18, 2026).

[47] Ofir Yakobi. *Watch the Typo: Our PoC Exploit for Typosquatting in GitHub Actions*. Sept. 2024. (Visited on Feb. 16, 2026).

[48] yossarian. *We Should All Be Using Dependency Cooldowns*. Reddit Post. Nov. 2025. (Visited on Jan. 6, 2026).

[49] Markus Zimmermann et al. "Small World with High Risks: A Study of Security Threats in the Npm Ecosystem". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 995–1010. ISBN: 978-1-939133-06-9. (Visited on Mar. 19, 2026).

List of Figures

List of Tables

Metric	Value	Unit
Collection Scope		
Snapshots	60	n
Observation window	2025-12-17 to 2026-02-13	
Unique repositories	113	repos
Unique workflows	28	workflows
Unique root actions	38	actions
Distinct action owners	14	owners
Dataset Size		
Action run records	6,038	rows
Depth summary records	3,752	rows
Transitive edges	30,065	edges
Pinning & Mutability		
Direct pinning rate	3.61	%
Direct unpinned rate	96.39	%
Transitive mutable edges	13.88	%
Transitive Dependency Types		
npm packages	92.65	%
GitHub Actions	6.75	%
Local actions	0.40	%
Docker images	0.20	%
Execution & Depth Statistics		
Total workflow runs	42,513	runs
Mean runs per record	7.04	runs
Max runs per record	50	runs
Mean transitive depth	0.52	levels
Max transitive depth	2	levels

Table 9: Overview of the aggregated GitHub Actions dependency dataset

9 AI Transparency Statement

The following table lists the AI tools and their application areas used during the preparation of this thesis. The disclosure is based on the guideline for labeling the use of AI tools by Goethe University Frankfurt [17] and considers the guideline on the use of AI tools in teaching by Hamburg University of Applied Sciences [24].

AI Tool	**Work Steps**	**Usage**
GitHub Copilot	Drafting and editing text	LaTeX inline suggestions, formatting of text paragraphs, figures, and tables
Google Gemini	Ideation and topic scoping	Preparation of outline points and content brainstorming
DeepL	Translation and language assistance	Translation of selected passages and language refinement
Semantic Scholar	Literature research	Identification of scientific sources

Table 10: AI tools used and their application areas

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel:

Evaluating Supply Chain Security Risks in CI/CD Pipelines: A Case Study on Mutable GitHub Actions

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Datum [redacted] Unterschrift

Änderungshinweis

Am 30.03.2026 habe ich eine nachträgliche Korrektur an der gedruckten Bachelorarbeit vorgenommen. Die Korrektur betrifft nur den Titel und das Abgabedatum auf der ersten Seite und die Eigenständigkeitserklärung auf der letzten Seite. Den Titel habe ich im Verlauf der Ausarbeitung abgeändert, im Irrtum dass der beantragte Titel nur vorläufig ist. Die Korrektur behebt den durch dieses Irrtum entstandenen Fehler.

Datum [redacted] Unterschrift