CfgNet: A Framework for Tracking Equality-Based Configuration Dependencies Across a Software Project

Sebastian Simon, Nicolai Ruckel, and Norbert Siegmund

Abstract—Modern software development incorporates various technologies, such as containerization, CI/CD pipelines, and build tools, which have to be jointly configured to enable building, testing, deployment, and execution of software systems. The vast configuration space spans several different configuration artifacts with their own syntax and semantics, encoding hundreds of configuration options and their values. The interplay of these technologies requires some level of coordination, which is realized by matching configurations. That is, configuration options and their according values may depend on other options and values from entirely different technologies and artifacts. This creates non-obvious configuration dependencies that are hard to track. The missing awareness and overview of such configuration dependencies across diverse configuration artifacts, tools, and frameworks can lead to dependency conflicts and severe configuration errors. We propose CFGNET, a framework that models the configuration landscape of a software project as a configuration network in an extensible and artifact-independent way. This way, we enable the early detection of possible dependency violations and proactively prevent misconfigurations during software development and maintenance. In a literature study, we found that the most common form of dependencies is the equality of values of different options. Based on this result, we developed an equality-based linker to determine dependent options across different artifacts. To demonstrate the extensibility of our framework, we also implemented nine plugins for popular technologies, such as Maven and Docker. To evaluate our approach, we injected and violated five real-world configuration dependencies extracted from Stack Overflow, which we support with our technology plugins, in five subject systems. CFGNET found all injected dependency violations and four additional ones already present in these systems. Moreover, we applied CFGNET to the commit history of 50 repositories selected from GitHub and found dependency conflicts in about two thirds of these repositories. We manually inspected 883 conflicts, with about 89 % true positives, demonstrating the need to reliably track cross-technology configuration dependencies and prevent their misconfiguration.

Index Terms—Configuration Dependencies, Configuration Conflicts, Services and Components.

1 INTRODUCTION

Modern software development often incorporates various technologies, such as containerization, build tools, and continuous integration and delivery (CI/CD) pipelines. Developers need to configure these technologies to build, test, deploy, and execute software systems. On top of that joint configuration space, software systems are often not singular and independent entities, but interact with other software systems, such as databases, operating systems, or external services that all need to be jointly configured. Therefore, the vast configuration space spans over different types of configuration artifacts (e.g., YAML files, Dockerfiles, and build files) with their own syntax and semantic [1]. To ensure the interplay and interoperability of all technologies, configuration options and their according values may depend on other options and values from entirely different technologies. These dependencies among different artifacts, frameworks, and tools create non-obvious and hard to track configuration dependencies.

The missing awareness of such configuration dependencies can lead to system failures, security vulnerabilities, or performance degradation [2], [3], [4]. Specifically, in the worst case, a single change of a configuration in the technology stack may cause another framework, or even another step in the CI/CD pipeline to fail. Fixing those cross-technology configuration-dependency errors can be cumbersome and time-consuming. Instead of fixing configuration bugs, we should concentrate on avoiding misconfigurations in the first place. However, developers miss an integrated overview of configuration dependencies across the diverse configuration artifacts, tools, and frameworks. Hence, all of these tools, which originally had the purpose of simplifying the development and delivery process, introduce further complexity and make it more challenging for developers instead.

```
1  <?xml version="1.0"?>
2  <project>
3      <artifactId>app</artifactId>
4      <version>1.0</version>
5  </project>
6
```

Listing 1: Maven build file defining the name of the generated JAR file.

```
1  FROM java:8
2  ADD target/app-1.0.jar app.jar
3  EXPOSE 8761
4  ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Listing 2: Exemplary Dockerfile deploying a JAR file.

As an example, we consider a typical Java project that is deployed in a Docker container. The application's build file is shown in Listing 1, in which we want to change the version number from 1.0 to 1.1, since we introduced a security fix. Now, we deploy the supposedly fixed software using our Dockerfile in Listing 2, possibly passing all tests of the CI server. Yet, we are still using the old version of the program

in production. In this small example, it is easy to see that we forgot to change the version number in the Dockerfile to grab the new JAR file. Unfortunately, this scenario is quite realistic, as the security developer may not be aware of the DevOps infrastructure of the software and may test the fix only locally.

This example demonstrates a dependency between two configuration options for which we need to keep both configuration values (version number and file name of the JAR) consistent. Otherwise, we will encounter a dependency conflict between Docker and Maven, which leads in the worst case to a configuration error. Although trivial, this exact example manifests in practice over and over again: We found exactly this configuration dependency in different Stack Overflow posts. Only when bringing both configuration files together makes the dependency obvious, but in a production setting the same dependency is hidden among dozens of configuration files, rendering it far from obvious. Therefore, even for such a trivial example, knowing this dependency is challenging, as both configuration values are encoded in different configuration artifacts and there is no obvious connection between them.

In practice, those dependency problems can be quite complicated, stretching over multiple artifacts, deployment stages, and environments. Sometimes there are even multiple places to adjust a setting, which could overwrite each other, and it is often unclear which of those values was applied last. For example, Sayagh et al. [5] provide a relevant example where the configuration of the maximum memory limit of a PHP script that a Web application is allowed to use can be specified in the configuration artifact of the Web server, PHP interpreter, or content management system.

There is a high demand for a good solution for these kinds of problems, as configuration-related problems have become one of the main causes for system failures in recent years [3]. To improve the tedious and error-prone manual configuration process of software projects, automation pipelines, and technology stacks, researchers have tried to develop mechanisms to detect and solve configuration problems automatically. Unfortunately, existing solutions are often restricted to isolated tool sets, for instance, Docker and Java, or PHP [6], or focus mainly on the diagnosis and detection of configuration errors. To the best of our knowledge, no previous approach attempts to address cross-technology configuration dependencies in a proactive, framework-, and language-independent way.

In this work, we address the problem of cross-technology configuration dependencies. As a first step, we conducted a systematic literature study to understand the manifold forms of configuration constraints and dependencies within software projects. Our results reveal a wide range of constraints on individual options as well as dependencies across multiple options. We also found that configuration dependencies are not limited to a single technology, but can span across the whole technology stack. A major finding is that value equality is an important form of cross-technology dependencies encountered in academic literature.

We propose CFGNET [7], a plugin-based framework for detecting and tracking dependencies among configuration options across different configuration artifacts. CFGNET represents configuration options of configuration artifacts as trees and connects them to build a network, where links between the leaf nodes of a tree correspond to the dependencies between configuration options. Our idea is to have a framework in which technology plugins analyze different configuration files (e.g., Dockerfiles or build scripts) and create nodes for configuration options with their user-defined values. Linker plugins then connect the nodes in case a dependency exists using a specific linker heuristic. Once a network is initialized, we can check whether changes to values of configuration options require changes of other options by traversing the configuration network. This approach enables consistency checks as Git hooks, suggestions of configuration values, and detections of potential violations of configuration dependencies. To demonstrate CFGNET's extensibility, we already implemented nine plugins for popular technologies that require cross-technology configuration. We also realized a general linker based on equality of configuration values, initially targeting one of the most encountered forms of cross-technology configuration dependencies.

To evaluate CFGNET's ability to model the configuration landscape of software projects and detect dependency conflicts using configuration networks, we followed state-of-the-art practice [4], [8], [9] by violating real-world configuration dependencies extracted from Stack Overflow in five software projects. In doing so, we simulated CFGNET's main application scenario and demonstrated that it is able to efficiently construct a configuration network and detect dependency conflicts. Moreover, we applied CFGNET to the commit history of 50 repositories selected from GitHub and found dependency conflicts in about two thirds of these repositories, indicating the practical relevance of cross-technology configuration dependencies. Finally, we manually reviewed 883 detected conflicts and annotated 89% of them with *yes* (i.e., true positives), showing the relevance as well as the applicability of our approach for real-world software projects. The evaluation scripts and further information can be found at our supplementary website [10].

To summarize, we make the following contributions:

- A systematic analysis of literature on configuration constraints and dependencies, classifying the manifold forms of configuration constraints and dependencies within software projects.
- CFGNET, a plugin-based framework that represents configuration options and code artifacts as nodes in a network of trees, where the links between the leaf nodes correspond to the dependencies between them.
- A technique for tracking equality-based configuration dependencies in an extensible and artifact-independent way using configuration networks, and thus a possibility to detect violation of configuration dependencies by evaluating the changes in the network.
- The ability to integrate diverse solutions, such as technology and linker plugins, into a common data structure and conduct analyses on it by leveraging the plugin-based architecture of CFGNET.
- An evaluation of CFGNET's main application scenario by purposefully violating dependencies in five subject systems, demonstrating CFGNET's ability to detect dependency conflicts and its potential for daily use.
- A comprehensive analysis of the commit history of 50 different repositories selected from GitHub, demonstrat-

ing the need to reliably track cross-technology configuration dependencies and prevent their misconfiguration.
- A supplementary code repository [7] and website [10], where the framework, data, and scripts are publicly available.

2 State of the Art

The goal of related approaches tackling the configuration landscape and related problems can be roughly divided into three categories: diagnosis of configuration errors, detection of configuration errors, and detection of configuration dependencies.

2.1 Diagnosis of Configuration Errors

Static and dynamic program analyses are employed to detect configuration errors. ConfAid uses dynamic taint analysis to identify the root cause of misconfigurations [11]. Rabkin and Katz [12] map each line of a program to a set of configuration dependencies using static program analysis. This way, users can query the table with an error message if they encounter a configuration error to find the option that fixes the error.

ConfDiagnoser by Zhang [13] uses a pre-built database of erroneous program executions to report a ranked list of suspicious configuration options, which deviate from correct execution profiles. Sayagh et al. [5] investigate Cross-stack Configuration Errors (CsCE) and propose a technique to recommend culprit configuration options in a LAMP stack. They take an error message as input and then leverage existing code slicing techniques, which are applied in each layer of the stack to generate a cross-stack slice dependency graph. By traversing the dependency graph, they recommend the most likely configuration options that caused a CsCE.

The main difference of the aforementioned approaches to CFGNET is that we provide a static as well as a language- and tool-independent approach that links configuration options outside a running application, crossing the boundaries of several tools and frameworks. In addition, we pursue a proactive (i.e., preventing configuration dependency violations) rather than a reactive (i.e., fixing configuration errors) approach.

2.2 Detection of Configuration Errors

Many approaches aim to detect configuration errors. A common approach relies on machine learning techniques to automatically extract configuration rules. For example, the frameworks Config [14], ConfigV [15], and EnCore [16] learn configuration rules by specification mining to ensure the correctness of configurations. We can make use of such configuration rules in our linker component.

Behrang et al. [17] propose SCIC to tackle inconsistencies in multi-language software projects. They use static analysis to create a unique set of preferences from the source code and compare it against preferences obtained from higher level APIs to report configuration inconsistencies.

Staccato by Toman and Grossman [18] focuses on evaluating runtime configurations by detecting stale data from old configurations in programs that run on the JVM to avoid configuration errors. Staccato requires developers to change their source code such that all configuration related data changes are delegated to the tool. Xu et al. [9] focus on latent configuration errors (LC errors), which occur due to configuration parameters that are neither used nor checked during normal operations. To this end, they propose the tool PCheck to analyze source code and generate configuration-checking code that emulates executions to check for illegal configuration options in Java and C programs.

While all the aforementioned approaches contribute valuable solutions to today's configuration problems, they are mostly language-dependent and require heavy in-depth analysis upfront in detecting configuration errors.

2.3 Detection of Configuration Dependencies

Closer to our work are approaches tackling configuration dependencies, which, in the worst case, may lead to configuration errors when violated. Chen at el. [4] study configuration dependencies within and across software components with the goal of deriving types of configuration dependencies with their common code patterns. Using the results from their study, they developed cDep, a tool that detects configuration dependencies from Java bytecode using static program analysis.

Ramachandran et al. [19] provide deeper insights about configuration-parameter dependencies between different component instances of software projects. They first analyze configuration data accessed via APIs to estimate candidates of configuration dependencies. Then, they compute weights for each dependency using a heuristic, and provide a ranked list of dependencies so that administrators can quickly identify true dependencies.

Lillack et al. [20] developed Lotrack to track configuration options from the moment they are loaded in the program using static taint analysis. Applying their tool results in a configuration map that explains which code fragments are affected by an option and where and how options may interact. Since their tool targets a different application scenario, it is limited to Java and cannot track configuration options across system boundaries.

With a light-weight static analysis, Metcalf et al. [21] extract control-flow and data-flow dependencies among configuration parameters. Their goal is to visualize interactions between configuration options at component level using an interaction graph.

With our approach, we complement the above approaches in the following aspects. First, related approaches concentrate mostly on a singular language or tool, whereas we pursue a plugin-based approach to be technology-agnostic. We believe that research should concentrate more on a framework-based development, since one-size-fits-all tools are often outdated, not maintained, and lack adoption in practice. Second, while previous approaches mainly aim at detecting configuration dependencies, they do not track configuration dependencies to enable the early detection of dependency violations. Third, CFGNET prevents possible violations of configuration dependencies by acting during the change of any configuration artifact (e.g., as a Git hook). Hence, we follow a proactive rather than a reactive (e.g., fixing an error) approach. Finally, we believe that existing work and their proposed techniques can be incorporated into CFGNET due to our plugin-based architecture (e.g., as alternative linker plugins), thereby establishing a common ground for a range of research approaches.

3 Constraints and Dependencies

Every single configuration option may not only have its own constraints, but can also introduce constraints between multiple options across the used technology stack. We therefore distinguish between unary and n-ary configuration constraints. The former describes the requirements that a value of a single configuration option should satisfy, such as the type, value range, or syntactic format [8]. For instance, a port is constrained by a data type and value range, which should be an integer in the range from 0 to 65535 [22]. N-ary constraints represent dependencies among multiple configuration options [23]. For instance, the port of an application introduces a constraint between multiple options, as the port is often specified in different configuration artifacts, such as in a Dockerfile or docker-compose.yml. Note that in the rest of the paper we refer to n-ary constraints as configuration dependencies (dependencies for short).

To obtain a large and robust overview about the diverse forms of unary constraints and dependencies, we conducted a literature study. Our methodology adopts the guidelines for performing a systematic literature review proposed by Kitchenham and Charters. [24] in the following steps: identification of research, selection of primary studies, and data extraction. In what follows, we describe our methodology in detail.

3.1 Methodology

First, we collected an initial set of papers by searching for keywords (i.e., configuration dependencies, configuration constraints, misconfiguration, configuration error) in three major academic libraries: IEEE Xplore, ACM Digital Library, and Google Scholar [25]. The rationale of having these keywords is to set the starting point of the literature study on relevant papers dealing with configuration constraints, dependencies, and their effects in software systems. Moreover, we selected the three major libraries to obtain a diverse set of initial papers. We applied our keywords to each library and extracted the top ten papers sorted by relevance, resulting in an initial set of 120 papers.

Then, two authors individually annotated the papers in the initial set either with *relevant*, *not-relevant*, and *duplicate* using our inclusion criteria. Specifically, we deemed a paper relevant if it addressed configuration constraints and dependencies in software systems, tackled the effects of their violations, or classified them. By reading the abstract, introduction, and conclusion for each paper in our initial set, we checked whether they met our inclusion criteria and annotated the papers accordingly. For the initial set of papers, we calculated the inter-annotator agreement using Cohens Kappa statistic [26], which was 0.87, indicating high agreement. However, we noticed that the initial inclusion was not clearly enough defined, since the two authors had different views on which papers to include and exclude. To this end, we sharpened the inclusion criteria and additionally defined exclusion criteria as follows: We also deem paper as relevant that address configuration constraints or dependencies in software-product-lines (SPL) and source code. By contrast, we deem paper as not-relevant that address hardware, non-technical, or ethical constraints and dependencies, or propose configuration modeling languages, technologies, or tools. Moreover, we dropped papers that deal with network, cyber-physical, or business process configuration. We then checked all papers in the initial set again using the final inclusion and exclusion criteria and annotated them accordingly, receiving a Cohens Kappa statistic of 1.0, which indicated perfect agreement among the annotators. With this approach, we finally obtained an initial set of 24 relevant paper.

In the next step, we extended our initial set of relevant papers, following a forward and backward snowballing approach [27]. That is, we first included papers that have been cited by the papers in our initial set and then added papers that referenced the papers in our initial set. This way, we received a set of 943 potentially relevant papers. Then, the main author applied again the same inclusion and exclusion criteria to all added papers to identify the relevant ones. Following this approach, we obtained a final set of 106 relevant papers. To measure the inter-annotator agreement, a second author randomly sampled 20 % of the final relevant and not-relevant papers and annotated them according to the inclusion and exclusion criteria. We received again a Cohen's kappa statistic of 1.0, indicating perfect agreement and a sound and reproducible criteria.

Finally, we read the full text of all relevant papers and extracted all types of configuration constraints and dependencies. Specifically, we extracted constraints and dependencies from concrete examples, but also from the proposed approaches in a paper. Two authors discussed the extracted configuration constraints and dependencies and assigned them to different categories. We based our categories with small modifications on the work of Xu et al. [8] and Chen et al. [4]. In addition to that, we also classified the extracted configuration dependencies either in intra- or cross-technology dependencies.

TABLE 1: Classification of unary configuration constraints.

Unary Constraints	#	Description	Example
Syntax	32	An option must correspond to a specific syntax pattern.	The *Duration* option in xml files must conform to the pattern: 00:00:[0-9]2+ [28].
Value Range	30	The value of an option must be within a specifc value range.	The *blocksize* option of mke2fs has a value range of 1024-65536 [23].
Data Type	30	An option must correspond to a specific data type.	In MySQL, the option *max_connections* must specify a numerical value [29].
Semantic	13	A semantic constraint refers to the sematic type of an option.	The port should not be occupied [30].

3.2 Results

The results of our literature study are shown in Table 1 and 2. We identified four types of unary constraints that restrict the value of options according to their syntax, value range, data type, and semantic. The majority of unary constraints refer to syntax, value range, and data type

TABLE 2: Classification of configuration dependencies. The columns #, #*Intra* and #*Cross* represent the number of total, intra-, and cross-technology configuration dependencies found for each type.

Dependencies	#	# Intra	# Cross	Description	Example
Value Equality	47	19	28	The value of an option must be equal to the value of another option.	In Apache, the options *NameVirtualHost* and *VirtualHost* should be set to the same name, otherwise the virtual host is loaded in a wrong order [2].
Control	29	22	7	The usage of an option depends on the value of another option.	In PostgreSQL, the option *commit_siblings* is enabled if *fsync* is not set to zero [8].
Value Inequality	19	12	7	The value of an option must differ from the value of another option.	In PHP, *mysql.max_persistent* must be smaller than the *max_connections* in MySQL [31].
Behavioral	11	5	6	Multiple options co-operate to influence the behavior of the systems.	In Hadoop Common, the host and port options are combined to create the IP address [4].
Overwrite	9	3	6	The value of an option is overwritten by the value of another option.	The option *dfs.client.retry.policy.spec* defines the timeouts and retries for HDFS clients. YARN overwrites this options with its own option [4].

constraints, followed by a few unary constraints referring to the semantic of an option. We also identified five types of configuration dependencies, which we additionally classified into intra- and cross-technology dependencies. Here, the majority of extracted dependencies points to value equality as cause of configuration dependencies. A similar picture results from the number of cross- and almost also of intra-technology dependencies, as the majority points again to value (in-) equality. Especially interesting for our use case are cross-technology constraints. Here, value equality has been reported in over 51 % of the cases. The second most extracted dependencies refer to control dependencies which often represent Boolean expressions involving multiple options. We also list value in-equality as a special case. Although it can be detected with similar technologies, this detection process is prone to produce many false positives. Furthermore, we found a few special cases that did not fit into our categories, such as structural or occurrence constraints, which affect the whole configuration artifact, and control- or data-flow dependencies of options in source code.

Overall, our results reveal clearly separable unary constraints and dependencies. While syntax, value range, and data type constraints are nearly equally prevalent among the identified unary constraints, value equality is the most frequent form of configuration dependencies found in our literature corpus. This may indicate that related work has primarily focused on value equality as cause of configuration dependencies, as it is relatively easy to detect. Although value inequality is also comparatively easy to detect, it quickly becomes impractical, as it would infer dependencies among all configuration options with different values, resulting in many false positives. Conversely, the other types of dependencies are often more complex than dependencies due to value relationships, and thus more challenging to identify unless explicitly documented. That is, these dependencies require an upfront specification in form of cross-technology variability models, which are not available. Moreover, we found that dependencies are not limited to single technologies, but can also span the entire the technology stack. This leads to manifold and complex cross-technology dependencies, which manifest among dozens of configuration artifacts, each with its own syntax and semantic, rendering them far from obvious.

Due to the diversity and complexity of cross-technology configuration dependencies, efficiently tracking these dependencies across a software project is a challenging but important task to proactively prevent misconfigurations. Unfortunately, there is a lack of approaches tackling cross-technology dependencies, since existing approaches are mostly language specific and do not track cross-technology dependencies within a software project. That is, they do not aim at proactively preventing the violations of configuration dependencies across the used technology stack. Our results clearly show that tracking configuration dependencies based on value equality represents an ideal starting point, as the majority of cross-technology dependencies point towards value equality as cause of the dependencies.

4 Configuration Networks

To model software artifacts, configuration options, and dependencies within a software project, we use a network of trees as the underlying data structure. A configuration network describes a software project and its surrounding ecosystem (i.e., CI/CD pipeline, containers, build tools, etc.), in which each tree corresponds to a part of the ecosystem in terms of nodes representing artifacts, options, and values. The leaf nodes of the trees can be connected by links, representing the dependencies among options. This representation allows us to detect and track configuration dependencies across the used technology stack. Configuration networks are built from different types of nodes as shown in the meta-model in Figure 1. Each used configuration artifact, option, and value is represented as a distinct node type in this network and is connected to all of its dependencies. The rational of having a hierarchy of different node types is to efficiently generate and compare networks. Moreover, this hierarchy

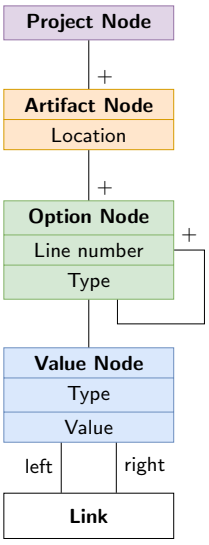


Fig. 1: Meta model of a configuration network.

allows us to model not only simple key-value configuration options, but also more complex configuration options, as demonstrated in our introductory example. Next, we discuss the different node types.

- **Project node:** The project node is the root of a configuration network and represents the whole software project.
- **Artifact nodes:** An artifact node models an existing configuration artifact, which can resemble different kinds of configuration files. A plugin of our framework usually covers a specific technology (e.g., Docker) and generates the corresponding subtrees when parsing configuration files for this technology (e.g., Dockerfiles).
- **Option nodes:** An option node represents a configuration option specified in the corresponding artifact node, such as a command, key, identifier, or a setting in general. It is also possible that a plugin developer creates additional option nodes, which are, for instance, composed of values of other options. Moreover, option nodes can be further specified with a configuration type (cf. Table 3). We base the type system with small modifications on the work of Li et al. [30]. To enable suggestions for fixing broken dependencies, we establish a tracing from the network to the physical place of the option by storing an option's location (i.e., line number in the configuration artifact).
- **Value nodes:** A value node represents an actual configuration value associated with their configuration option, such as a port or version number. Each value node v also points to its parent option node $v.parent$ and inherits its type.
- **Link:** A link l is a tuple $(l.left, l.right)$ of value nodes that represents a dependency between those nodes. We discuss whether and where we establish links in Section 5.2.

TABLE 3: Available configuration types.

From Li et al. [30]	time, port, version_number, memory, fraction, speed, permission, count, size, ip_address, username, mode, url, email, domain_name, boolean, password
Own additions	unknown, protocol, image, path, name, command, license number, id, pattern, environment, platform, language, type

Figure 2 illustrates the configuration network of our introductory example. To construct the network CFGNET iterates over all files of the software project, usually extracted from a version control system (VCS). Then, for each file, CFGNET selects the responsible plugin that has registered to handle the current file type. A plugin usually matches to a concept, parses the current file of the concept to construct the artifact's subtree, and adds it to the network. Here, CFGNET finds two configuration artifacts (i.e., the Dockerfile and pom.xml) and selects the responsible plugins (i.e., Docker and Maven plugin). That is, the Docker and Maven plugin parse the file of their concept, create the artifact's subtree with all option and values nodes, and add it to the network. Once all files are parsed, the linker manager selects enabled linker plugins, which finally link nodes based on their specific linker criterion. Note that for demonstration purposes, the network shows only nodes that are involved in the configuration dependency with Maven's executable name.

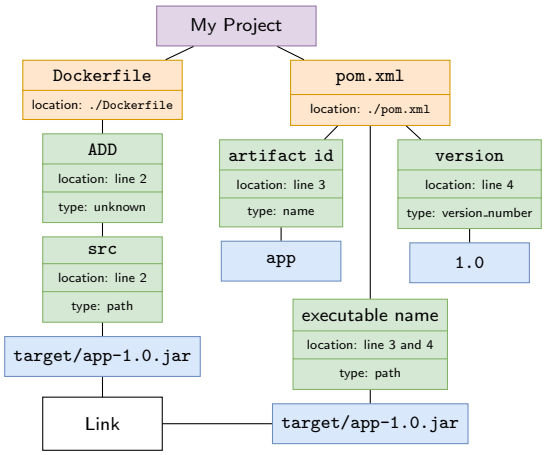


Fig. 2: Simplified configuration network of the example Java project with Docker.

5 ARCHITECTURE

CFGNET's architecture consists of three main parts: the plugin manager, the linker manager, and the conflict detector (see Figure 3). To create a network, CFGNET iterates over all configuration artifacts of a software project. The plugin manager chooses the right plugin for each artifact to update the network together with the linker manager's plugins, which detects possible dependencies. Once a network is created, the conflict detector can find potential dependency issues by using this network as a reference.

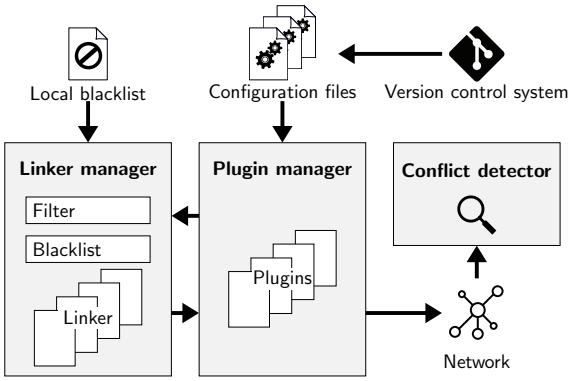


Fig. 3: CFGNET's architecture overview.

5.1 Plugin Manager

Plugins are the key success factor for our approach: they determine how much information a configuration network contains. Every plugin consists of two parts. First, a plugin specifies rules based on file extensions or naming conventions to select the configuration artifacts to be parsed. Second, a plugin parses the selected artifact to find configuration options and their values, which are then returned and added as nodes to the artifact's subtree. Plugins do not need to link their nodes to the nodes created by other plugins. Instead, they can solely rely on the linker plugins. All currently implemented plugins and their corresponding filter rules are shown in Table 4.

TABLE 4: Existing plugins with their type and filter rules.

Plugin	Filter rule
Maven	pom.xml
Docker	Dockerfile
Docker-Compose	docker-compose*.yml
Travis CI	.travis.yml
Node.js	package.json
Spring	application*.yml | application*.properties
Pyproject	pyproject.toml
TSconfig	tsconfig.json
Cpyress	cypress.json

Plugins are always technology-specific and incorporate domain knowledge when parsing configuration files. That is, plugin developers can encode their domain knowledge to create nodes for hidden or implicit configuration options and their values, such as aggregated option values, or to add types for configuration options according to our type system in Table 3. For example, in our introductory example, the Maven plugin creates an additional node for the executable name, which is computed by composing the project name with an optional version number and a packaging format, which defaults to *JAR* in this case. From the Maven specification, we also know that the default folder for the compiled files is *target/*, so we can prepend that to the executable name and add the corresponding type.

The process for writing a new technology plugin is to find a parser for the file type of the technology (usually existing) and translate its results into nodes, which is usually an easy task (e.g., traversing the parsed representation of the configuration file and outputting the data types of our network). More challenging is the inclusion of domain knowledge, which is benefical to extract more complex configuration options and add type information. Once a technology plugin is implemented, the plugin usually does not need to change, as it cover third-party technologies, whose implementation is project-independent. Here, changes are required only if the corresponding artifact changes its syntax, which is a rare case in practice as this is considered a breaking change. All existing plugins range from 63 to 385 lines of code and have been developed by the authors and undergraduate students, indicating the low complexity and effort of this task.

5.2 Linker Manager

We designed our framework to incorporate detected or already known dependencies as links in the configuration network. Links are created by linker plugins, which rely on a specific linker heuristic and incorporate type information if available. Since there a manifold forms of cross-technology configuration dependencies within software projects, we conducted the systematic literature study presented in Section 3. Based on our results and in line with related work [4], [5], we implemented a general linker plugin that relies on the equality of configuration values to target the most frequent form of cross-technology configuration dependencies found in our literature corpus. That is, our equality-based linker plugin assumes that configuration options depend on each another if their configuration values are equal.

We are aware that the equality-based linker bears the risk of creating false positives, as not all configuration option whose values are equal have to be dependent on each other. To mitigate such issues, we add two mechanisms. First, the equality-based linker incorporates type information if available in the linking process to prevent linking values whose types are different. Second, we conducted pre-studies to create a global blacklist β_g of values. This way, we avoid configuration values that bear the risk of being involved in too many links, although they do not represent an actual dependency between configuration options. For example, we excluded Boolean values, which are almost always false positives. Other entries in the global blacklist are *Null*, *None*, *Yes*, and *No*. In addition, developers can create a local blacklist β_l with project-specific entries similar to Git's gitignore file. This enables developers to adapt the linking algorithm to specific software projects in order to mitigate the risk of a linker generating false positives.

The general linking process is described as follows. Once a network is created, for each value node v, a linker iterates through all value nodes V included in the network and checks if they match the link criterion λ but not any of the global filter β_g and the local filter β_l. For all detected matches, the linker generates a link between the two value nodes represented as a tuple $l = (v_i, v_j)$ and adds that link to the network. Equation 1 shows the linking approach of how the equality-based linker obtains the set L of links between configuration values in a network.

$$L = \{(v_i, v_j) \mid v_i, v_j \in V \wedge \overbrace{\lambda(v_i, v_j)}^{\text{link criterion}} \wedge \overbrace{val(v_i) \notin \beta_g, \beta_l\}}^{\text{filter}} \quad (1)$$

The equality-based linker checks if two value nodes v_i, v_j have the same value (cf. Equation 2):

$$\lambda(v_i, v_j) \coloneqq val(v_i) = val(v_j) \wedge matching\_types(v_i, v_j) \quad (2)$$

In addition, the equality-based linker links value nodes only if they have the same type (cf. Equation 3):

$$matching\_types(v_i, v_j) \coloneqq type(v_i) = type(v_j) \quad (3)$$

Besides plugins for technologies, CFGNET can also be easily extended by adding other or more specialized linker plugins due to our plugin-based approach. Those linker plugins may draw their linking heuristics from repository mining or rely on specific rules. Note that the development of specialized linker plugins is not the goal of this work, as this would be an entirely different contribution in the research directions of mining configuration errors and their rules. However, our framework approach supports diverse solutions, such as technology and linker plugins, in a common data structure.

5.3 Conflict Detector

Storing links within a configuration network allows us to detect dependency conflicts of configurations across a software project. Those dependency conflicts occur when a developer makes a change to a configuration in a way that the values of two linked value nodes do not match anymore.

There are two causes for those conflicts. First, the values of the two nodes are not equal anymore. Second, an artifact or option node has been removed from an associated value node.

The conflict detection module detects possible dependency conflicts by creating a configuration network for the current version of the software project and comparing it with a reference network (e.g., from the last commit or from a blueprint project). First, we compute the missing links $L_{missing}$, that is, all links that were present in the reference network but are missing in the new one by computing the relative difference of the links in the new network N_{new} to the old network N_{old} (cf. Equation 4):

$$L_{missing} = \{l \mid l \in N_{old} \setminus N_{new}\} \tag{4}$$

The absent links do not tell us the cause of their absence. However, to provide suggestions on how to fix a missing link due to inconsistent value changes, we need to find both linked value nodes in the new network (cf. Equation 5):

$$\begin{aligned} L_{equiv} = \{l' \mid & l'.left = N_{new}.find(l.left) \\ & \wedge\, l'.right = N_{new}.find(l.right)\} \end{aligned} \tag{5}$$

To this end, we need to find a value node with the same parent as the original node in the new network (cf. Equation 6):

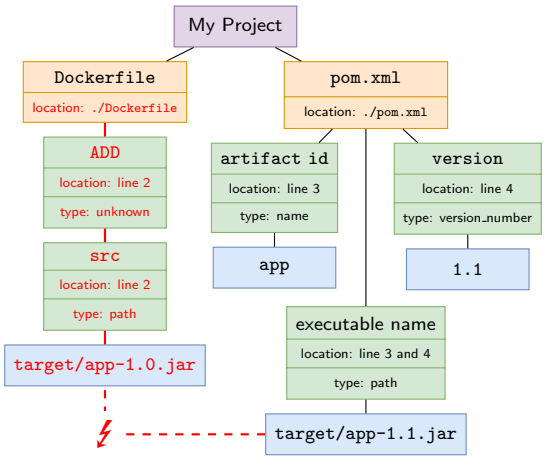
$$N.find(v) = \{v' \mid v' \in N \wedge v'.parent = v.parent\} \tag{6}$$

Equation 7 shows the definition of the set of potential conflicts C which is defined by all links from L_{equiv} where the values of the included value nodes are different from each other according to the link criterion (cf. Equation 2):

$$C = \{c \mid c \in L_{equiv} \wedge \neg\lambda(c.left, c.right)\} \tag{7}$$

We distinguish between three different types of conflicts. The first type is a *Modified Option Conflict*, which is the primary kind of conflict that CFGNET detects by default. A *Modified Option Conflict* occurs when one or both value nodes of a link changed their values in a way that they are not equal anymore. The second and third types are *Missing Artifact* and *Missing Option Conflicts*. Those conflicts are caused by a missing value in the current network due to the removal of the corresponding artifact or option. Removing an artifact or option does not necessarily result in configuration conflicts. However, faulty project setups, erroneous CI processes, or just human error may cause the removal of artifacts or options. Due to the configurability of CFGNET, the detection of those conflicts can be enabled by the developers if desired.

Figure 4 shows how the conflict in our introductory example is detected. First, CFGNET creates the reference network using the plugins for Maven and Docker. During that process, the linker creates a link between the two matching values for Docker's *ADD* and the executable name in the `pom.xml`. When we change the version number in the `pom.xml` and run CFGNET again, a new network is created and the links in both networks are compared. The conflict detector will then detect a missing link between the value nodes for the executable name and *ADD*. By traversing the tree from the unchanged value node to the corresponding artifact node, we get the corresponding file name and line number to suggest a possible fix using the new value (see Figure 4).




```
$ cfgnet validate /home/user/example
MODIFIED OPTION (b7407cb92b2fc55c57a8fad6461075b7af201ed0)

Modifed Option: "Executable Name" in artifact "pom.xml"
Value changed from "target/app-1.0.jar" to "target/app-1.1.jar"

Conflicts:
In file "/home/user/example/Dockerfile:2
Link with option "add::src" is missing
Modify option "add::src"
"target/app-1.0.jar" to "target/app-1.1.jar"
```

Fig. 4: By comparing the current network with the reference network (cf. Figure 2), we see that the link shown with a dotted red line is missing here. We can traverse the tree from the other value node up to the corresponding artifact node to inform the user how to fix the error.

6 Evaluation: Dependency Violation

To demonstrate CFGNET's ability to model a software project's surrounding ecosystem and detect real-world dependency conflicts, we evaluate our framework on real-world dependency violations. To this end, we first extract configuration dependencies from Stack Overflow (SO) posts and inject one intra- and four cross-technology configuration dependencies in five real-world software projects. Next, we analyze whether we can detect those dependency violations with CFGNET.

6.1 Stack Overflow Analysis

Although there are many approaches tackling configuration errors and dependencies, to the best of our knowledge, no readily available dataset of configuration dependencies and their violations exists. To this end, we analyzed SO posts to obtain a set of typical real-world intra- and cross-technology configuration dependencies.

Methodology. We downloaded the SO dataset from the Stack Exchange Data Dump on November 12, 2021 [32]. The dataset contains posts generated from 2008 to 2021 and provides for each post metadata, such as the identifier, post type, title, body, and tags. Moreover, a post contains access metrics, such as the view, answer, and favorite count, as well as a score determining its relevance.

Following previous SO analyses [33], [34], we created a vocabulary with our technology-specific plugin names (i.e., maven, docker, docker compose, travis, node.js, spring,

TABLE 5: Five real-world configuration dependencies extracted from SO, including four cross- and one intra-technology configuration dependency.

ID	Dependency Name	Count	Technologies	Dependency Description
1	Executable Name	68	Spring, Maven, Docker	Java applications that are built with Maven and deployed with Docker depend on the executable name, which is specified in the application's build and used in Docker to add the executable to the file system of the container.
2	Internal Docker Dep.	63	Docker	Docker's *ADD* and *ENTRYPOINT* commands often refer to the same configuration value, such as when both commands specify the executable name inside the container, and thus creating an internal dependency.
3	Port	58	Spring, Docker, Docker-Compose	Dockerized Spring Boot applications usually exhibit multiple configuration options, which specify the application's reachable port. Thus, the port is defined at different places, such as in a Dockerfile and docker-compose.yml.
4	Database Credentials	62	Spring, Docker-Compose	Multi-container applications often incorporate databases. To access those databases, credentials are specified at different configuration artifacts, such as in Spring Boot's Properties file and in the docker-compose.yml.
5	Configuration Artifact	94	Docker, Maven, Node.js, Poetry, TSconfig	Configuration files, such as the pom.xml and package.json are often copied from the local root directory to the host machine's file system to build and run the application in a Docker container, thereby creating a dependency.

pyproject, tsconfig, cypress) to extract only those configuration dependencies that we can cover with our currently supported plugins. Note that we are not conceptually limited to these plugins, but consider these as suitable search keys for finding configuration dependencies, as they often interact with other technologies in a software project.

To select relevant posts, we first filtered the dataset using each possible tag pair created from our vocabulary, then sorted the resulting posts in descending order based on their score, and finally extracted the top 50 posts for each technology pair. We focused on the most relevant posts for each tag pair using the score as metric indicating how much attention a post has received by the SO community to extract typical and relevant real-world intra- and cross-technology configuration dependencies. The number of posts varied among technology pairs. That is, we obtained 50 posts for 16 pairs, 0 posts for 13 pairs, and for the remaining pairs we found between one and 50 posts. Overall, we obtained 888 SO posts (i.e., 847 unique posts) for 36 different tag pairs, which were then manually reviewed by the main author.

Naturally, manually extracting configuration dependencies from SO posts is challenging, because configuration dependencies are not always obvious and explicitly declared. To this end, we manually reviewed each post taking all available information in the posts into account, including code snippets, figures, answers, and comments, to identify dependent options. Specifically, we focused on options that appeared together in the posts with the same value (e.g., in code snippets or answers). Moreover, we considered individual options if the questions, answers, or comments indicated that other technologies were involved, which included options with the same value. By using domain knowledge, we also identified complex options, such as hidden, nested, or composed options. In all cases, we finally decided if the configuration options represent real configuration dependencies, and if so, we extracted the configuration dependency. Unclear configuration dependencies had been discussed by two authors until an agreement was reached. Following this methodology, we identified and extracted relevant dependencies that rely on value equality.

Note that our selection does not include search terms for configurations, dependencies, or conflicts, since they may bias our search. Moreover, developers may be unaware that they have a dependency problem and not add such terms as tags. This approach even increases our validity, because if we find posts with configuration dependencies, we can be certain that these dependencies are relevant to a large number of developers, as the high score is compared to any development activity (i.e., any topic post) involving the corresponding technologies and not just configuration related.

Results. In total, we found 10 configuration dependencies that appeared in multiple SO post, indicating that we have extracted typical real-world configuration dependency examples. Among the extracted configuration dependencies, there are three intra- and seven cross-technology dependencies. Our approach detects cross-technology configuration dependencies by default, but is not conceptually limited to them. Instead, CFGNET also allows the detection of intra-technology configuration conflicts, so that we decided to keep one intra-technology configuration dependency. To this end, we selected the most common intra- and the four most common cross-technology configuration dependencies for the dependency violation injection. We show these five real-world configuration dependencies extracted from SO in Table 5. The remaining configuration dependencies can be found at our supplementary website [10].

6.2 Dependency Violation Injection

We selected five projects from GitHub that have been built with the technologies involved in the extracted configuration dependencies such that each project incorporated at least one of the configuration dependencies extracted from SO. Specifically, we first filtered GitHub repositories by the corresponding technologies (i.e., docker, docker-compose, spring/spring-boot, and maven) and then manually reviewed the best matches. Thereby, we looked into the configuration files of the repositories and selected these software repositories that incorporated a configuration dependency extracted from SO. We excluded all repositories

TABLE 6: Overview of subjects systems, including the number of technologies (|T|), configuration artifacts (|F|), total options (|O|), and average number of options per artifact (|Ø O|) as well as the results of the dependency violation. The columns *ID*, *Injected*, *Detected*, and *Unexpected* show the types of violated dependencies extracted from SO, the total number of dependency conflicts that we injected, the actually identified conflicts by CFGNET, and the number of additionally detected conflicts. The columns *Detected* and *Unexpected* under *Fix* show the number of fixes suggested by CFGNET for the detected and additionally detected conflicts.

Repository	Config. Statistics				ID	Dependency Conflicts			Fix		Performance (in seconds)	
	|T|	|F|	|O|	|Ø O|		Injected	Detected	Unexpected	Detected	Unexpected	Construction	Detection
spring-boot-blog	3	3	72	24	1, 3	2	2	0	2	0	0.02	0.02
Ward	2	2	55	27.5	5	1	1	0	1	0	0.02	0.02
piggymetrics	5	29	680	23.4	1, 3	4	4	1	4	1	0.32	0.33
netflix-oss-example	4	34	626	18.4	1, 2, 3	5	5	1	5	1	0.21	0.22
taskManagement	4	7	125	17.9	1, 2, 4	3	3	2	3	2	0.03	0.04

that did not incorporate the corresponding technologies and configuration dependencies. Moreover, we dropped software repositories that did not represent real software projects. We stopped our search after we found 5 software projects, so that the extracted configuration dependencies from SO were covered at least once. We show our subject systems with configuration-related statistics in Table 6

We then followed state-of-the-art practice [4], [8], [9] by purposefully violating one or more existing dependencies in a local fork of the selected software projects. Technically, we first applied CFGNET to a software project to initialize the reference network. Next, we injected the dependency violations in the software project by randomly changing one value of the configuration dependencies under test. Finally, we applied CFGNET again to the software project to create the modified version of the network, which was then compared to the reference network to detect dependency conflicts. Note that we knew the configuration dependencies in each subject systems, since we manually looked into each project and selected those that incorporated at least one configuration dependency extracted from SO.

6.3 Results

We report the results of our dependency violation in Table 6. As we will explain in more detail next, we detected 19 real-world dependency conflicts, demonstrating CFGNET's potential for daily use.

6.3.1 Detected Dependency Conflicts

In all subject systems, we purposefully violated one or more existing configuration dependencies by changing one of their values, respectively. This way, we injected a total of 15 known dependency violations into our five subject systems and applied CFGNET in order to detect the resulting dependency conflicts. From the 15 injected dependency conflicts, CFGNET could detect all of them and provided correct suggestions to fix them in all cases.

Interestingly, CFGNET additionally detected 4 unexpected dependency conflicts, for which it also provided correct fixes. We found two of them when we changed the name of a specific service to violate the executable name dependency (ID 1). Specifically, we renamed a certain service in Piggy Metrics and in Netflix OSS Example and expected to detect only a single dependency conflict, respectively. However, we detected one additional dependency conflict in each subject system, since other options also referred to the service name. We believe that these two unexpected conflicts are real dependency conflicts, because when the name of a service is changed, all other configuration options that refer to the service name should change as well.

The two remaining conflicts were detected when we changed the username for a database to violate the database credentials dependency (ID 4). Here, we renamed the username in Task Management and expected to detect only one dependency conflict, but we detected a total of three conflicts for this dependency violation. The problem here is that the project incorporates different Spring environment configurations file, since we found configuration files for the *dev* environment. Currently, we do not distinguish different Spring environments, however, we believe that when the credentials of a database are changed, the current environment does not matter. Hence, we additionally detected two dependency conflicts across different environment files that represent real dependency conflicts. So, interestingly, we found a security issue with our approach.

6.3.2 Performance

Since CFGNET's network construction and conflict detection are intended to be used in Git hooks, it needs to be reasonably fast. Hence, we measured the construction and conflict detection time for each software project during the dependency violation. The results in Table 6 show that CFGNET's performance in constructing a configuration network and detecting dependency conflicts is well below half a second, and thus reasonably fast. Therefore, a developer would get an immediate response when using our framework for detecting dependency conflicts, which represents an important aspect of its practicality.

6.3.3 Summary

In total, we injected 15 known dependency violations into five subject systems selected from GitHub. All known dependency violation were reliably detected by CFGNET. In addition, CFGNET detected four dependency conflicts caused by changing a certain service name and database credentials. With this evaluation, we demonstrated CFGNET's main application scenario and its potential for daily use. We also showed how CFGNET helps to prevent possible configuration dependency conflicts during software development and maintenance.

7 Evaluation: Commit History

To demonstrate the need to reliably track cross-technology configuration dependencies, we analyzed the commit history of 50 repositories selected from GitHub. In doing so, we simulated a generic application scenario of CFGNet, where we employed the technology-plugins and a general linker to detect dependency conflicts.

7.1 Setup

To not bias our evaluation, we selected the top 50 repositories from GitHub with the most stars (June 2022) that incorporate at least two of the technologies for which we have already implemented corresponding technology plugins. We replaced repositories that could not be analyzed due to errors caused by third-party libraries and that did not represent real software systems, for example, collections of best practices, frameworks, or link hubs. Based on these criteria, we ensured that we select relevant software systems that can be fully analyzed and potentially exhibit violations of cross-technology configuration dependencies in their commit history.

We then analyzed the commit history of our subject systems in an automated manner and tracked all detected dependency conflicts for each commit. Since commit histories are usually not linear, we cloned each subject system and applied Git's filter-branch command[1] to the local fork of each subject system in order to linearize their commit histories.

After the analyses, we sampled 50 *Modified Option Conflicts* for each subject systems and reviewed them. If there were less than 50 conflicts, we reviewed all of them. Note that we sampled only *Modified Option Conflicts* and excluded all other conflict types for three reasons. First, *Modified Option Conflicts* are the primary type of conflicts that CFGNet detects by default. Second, for *Modified Option Conflicts*, we were able to manually inspect and reason about the severeness of the reported conflict. Third, for *Missing Artifact* and *Missing Option Conflicts*, there is no way of telling whether the removal of artifacts and options was intended.

The review of dependency conflicts encompassed the following steps: First, one author went through all commits involving a conflict and checked the change that caused the conflict as well as the involved configuration options and values. Thereby, the author used his experience and domain knowledge to reason about whether the change actually leads to a configuration conflict. Finally, the author marked the conflicts accordingly as either clearly *yes*, *no*, or *borderline case*. As this methodology might be subjective, the second author reviewed all borderline cases and manually sampled clear cases for validity checks.

7.2 Results

We report the results of the commit history analyses and manual review of dependency conflicts in Table 7. We found dependency conflicts in 39 (78 %) subjects systems, while 32 of them (64 % of all subject systems) contained *Modified Option Conflicts*. From each of the 32 subject systems, we created sample sets for the manual review of the dependency conflicts. In total, we manually reviewed 883 *Modified Option Conflicts*, annotated 785 (89 %) conflicts with *yes*, and 98 (11 %) with *no*. To discuss the reviewed conflicts, we classified them into different categories based on their rating and the involved configuration types. We report the categories of conflicts in Table 8.

1. git filter-branch --parent-filter 'cut -f 2,3 -d " ""

7.2.1 Dependency Conflicts

The most reoccurring dependency conflicts were due to the version number of libraries and configuration files. Specifically, these conflicts frequently occurred in projects that incorporated Maven and Node.js. This is not surprising, as these projects often contain several packages that come with their own configuration files, in which the versions of libraries and configuration files are specified. Version inconsistencies can lead to a wide range of issues, including compatibility issues, security threats, broken builds, and maintenance issues [35], [36]. For example, Santolucito et al. [37] demonstrate that a version inconsistency caused a CI failure, which resulted in a broken build of the project. Moreover, these dependency conflicts are related to a common problem, which is also known as the dependency hell [38]. Usually, a dependency hell emerges in large and complex software systems due to a multitude of software libraries. These libraries introduce their own dependencies across different configuration artifacts, making it difficult and time-consuming to adequately set up, manage, and maintain all the intertwined dependencies. Consequently, keeping those dependencies consistent is challenging even for experts due to missing awareness and complexity of configuration dependencies across the used technology stack.

The second most reoccurring conflicts belong to the category *Unknown*, which also represents the category with most conflicts labeled with *no*. Taking a closer look at these conflicts showed that they were mainly caused by inconsistent Java and NPM version, compiler options, and base images in pom.xml, package.json, tsconfig.json, and Dockerfiles. Additionally, we found two dependency conflicts caused in the Spring configuration file, involving a Redis timeout and the Elasticsearch cluster-node option that points towards the Elasticsearch container. The severity of these dependency conflicts encompasses a wide range of issues, such as compatibility, connectivity, compilation, and maintenance issues. Moreover, conflicts in this category also reveal an issue of the technology plugins and the equality-based linker. Here, options without any type information were linked, and since they did not change consistently, we found conflicts. This means that a few plugins do not fully exploit domain knowledge (i.e., type information), and therefore the equality-based linker solely relied on his linker heuristic and linked options with the same values. Considering the distribution of labeled conflicts, it becomes apparent that incorporating type information significantly mitigates the problems of the equality-based linker, making the linker practicable.

Other reoccurring dependency conflicts were caused by the scripts field in package.json files, since different package.json files often specified the same scripts and entry points for a project, representing conflicts in the *Command* category. Basically, scripts are a set of commands, including built-in and custom scripts that are run at various times in the lifecycle of a project. If these options are not changed

TABLE 7: Results of the commit history analyses. The columns *MAC*, *MOC*, and *ModOC* represent *Missing Artifact*, *Missing Option*, and *Modified Options Conflicts*, respectively.

Name	Stars	Language	Analysis Statistics					Performance		Conflict Review		
			# Commits	# Conflicts	# MAC	# MOC	# ModOC	Ø Construction (in s)	Ø Detection (in ms)	Sample	yes	no
angular	82.0K	TypeScript	24105	126	41	57	28	0.2	0.08	28	16	12
animate	74.8K	CSS	378	1	0	0	1	0.01	0.01	1	1	0
ant-design	80.7K	TypeScript	11256	1	0	0	1	0.06	0.01	1	1	0
atom	58.2K	JavaScript	15563	24	3	13	8	0.09	0.19	8	8	0
axios	94.0K	JavaScript	947	0	0	0	0	0.02	0.01	0	0	0
Chart	57.3K	JavaScript	2594	3	0	3	0	0.05	0.01	0	0	0
code-server	54.1K	TypeScript	1632	11	6	3	2	0.14	0.02	2	1	1
core	53.4K	Python	41160	2	0	0	2	0.22	0.01	2	2	0
create-react-app	95.5K	JavaScript	2631	168	10	30	128	0.07	0.20	50	50	0
deno	83.0K	Rust	7223	5	5	0	0	0.03	0.01	0	0	0
developer-roadmap	197.1K	TypeScript	953	0	0	0	0	0.02	0.003	0	0	0
django	64.6K	Python	30004	0	0	0	0	0.14	0.01	0	0	0
echarts	51.4K	TypeScript	3262	0	0	0	0	0.02	0.003	0	0	0
elasticsearch	60.0K	Java	20502	66	5	59	2	2.85	0.75	2	2	0
electron	102.1K	C++	11861	18	7	1	10	0.05	0.11	10	2	8
element	52.2K	Vue	3097	182	42	10	130	0.05	0.37	50	50	0
fastapi	46.3K	Python	2174	2	2	0	0	0.03	0.02	0	0	0
FiraCode	64.4K	Clojure	513	0	0	0	0	0.01	0.01	0	0	0
freeCodeCamp	347.6K	TypeScript	21148	232	35	82	115	0.74	0.31	50	47	3
gatsby	53.1K	JavaScript	18810	8261	80	491	7690	4.77	95.75	50	50	0
go	100.7K	Go	49816	0	0	0	0	0.11	0.01	0	0	0
hugo	59.6K	Go	6870	0	0	0	0	0.03	0.003	0	0	0
ionic-framework	47.5K	TypeScript	9497	344	52	93	199	0.11	0.11	50	46	4
kubernetes	89.4K	Go	46764	55	32	12	11	16.19	0.01	11	8	3
mall	59.0K	Java	841	89	7	67	15	0.19	1.26	15	13	2
material-ui	79.1K	JavaScript	11469	665	27	107	531	1.37	1.01	50	50	0
moby	63.3K	Go	16129	1	1	0	0	0.11	0.004	0	0	0
moment	46.6K	JavaScript	1661	0	0	0	0	0.02	0.01	0	0	0
nest	47.8K	TypeScript	3816	856	62	48	746	0.35	1.05	50	50	0
netdata	59.7K	C	9337	3	1	2	0	0.16	0.01	0	0	0
next	88181	JavaScript	11089	2575	220	802	1553	9.47	252.29	50	50	0
node	88.3K	JavaScript	33558	3778	664	1311	1803	8.21	43.13	50	47	3
nvm	58.6K	Shell	1432	0	0	0	0	0.02	0.01	0	0	0
opencv	62.2K	C++	11224	2	0	0	2	0.11	0.02	2	2	0
protobuf	55.0K	C++	3663	268	11	174	83	0.33	1.23	50	34	16
puppeteer	78.5K	TypeScript	2853	58	12	23	23	0.05	0.07	23	21	2
rails	50.9K	Ruby	38425	12	0	9	3	0.07	0.02	3	3	0
redux	58.2K	TypeScript	1964	0	0	0	0	0.03	0.01	0	0	0
rust	67.7K	Rust	35625	58	33	3	22	0.19	0.10	22	18	4
spring-boot	61.7K	Java	21042	5899	586	4796	517	33.03	0.464	50	31	19
socket	56.0K	TypeScript	1326	26	1	4	21	0.03	0.15	21	20	1
storybook	71.7K	TypeScript	11448	9686	228	1591	7867	4.78	60.53	50	50	50
superset	46.6K	TypeScript	9654	9	7	2	0	0.09	0.01	0	0	0
svelte	59.5K	TypeScript	2608	6	1	0	5	0.10	0.01	5	5	0
transformers	65.3K	Python	8386	13	0	10	3	0.04	0.08	3	3	0
TypeScript	81.5K	TypeScript	11286	138	19	95	24	1.38	4.22	24	11	13
vscode	133.0K	TypeScript	40587	786	45	407	334	3.06	38.73	50	43	7
vue	196.9K	TypeScript	3297	243	6	17	220	0.06	0.07	50	50	0
vue-element-admin	76.7K	Vue	906	0	0	0	0	0.02	0.004	0	0	0
webpack	61.2K	JavaScript	5254	1	1	0	0	0.14	0.10	0	0	0
Total										883	785	98

consistently together, various errors can occur such as build errors, inconsistent system behavior, and even difficulty in managing and updating scripts. Since scripts are often used to automate repetitive tasks, such as running a linter tool on the source code, executing tests, building the project, the resulting conflicts are often detected in CI/CD pipelines.

We also found one of the dependency conflicts (ID 4) that we injected in Section 5. The username and password were changed for the Spring data source, whereas the same options were not changed in another configuration artifact, which can lead to connection problems with the data source. Those conflicts easily lead to serious problems, such as connection timeouts, access denied problems, and network-related issues if the application or container cannot connect to the data source. This kind of problem were discussed in numerous SO posts that we reviewed and from which we extracted this cross-technology dependency, indicating the need to change these options consistently in all corresponding artifacts. In general, username and password options are always critical, since they also may expose security threats.

TABLE 8: Classification of dependency conflicts based on their ratings and the involved configuration types.

Configuration Type	Rating	
	yes	no
Version Number	670	14
Unknown	51	37
Command	26	2
Path	14	22
Name	6	23
License	5	0
URL	4	0
User Name	3	0
Password	3	0
Type	2	0
Language	1	0
Total	785	89

Other less reoccurring dependency conflicts were caused by URLs, file paths, packaging types, service/module names, or licenses. These dependency conflicts also vary in their severity, ranging from crashing to non-crashing errors. For example, service/modules names have to be changed consistently across all configuration artifacts in multi-module Maven projects, otherwise the build of the projects will fail. Moreover, service/module names or packaging types are often part of composed options, and changing them in an inconsistent way can cause serious problems, as shown in our introductory example.

Most conflicts that are labeled with *no* belong to the *Unknown*, *Name*, and *Path* categories. During the review of these conflicts, we noted two issues. First, as already mentioned above, a few plugins lack domain knowledge, so that the linker does not always take meaningful type information into account when creating links. Second, not all options that are specified in configuration files are necessary to track, because we found many conflicts caused by changes to the name or description of configuration files, to relative file paths, or to placeholder values for version numbers.

These results strongly support our framework-based approach to track configuration dependencies with technology plugins by incorporating domain knowledge. These are the key factor for the success of CFGNET. Specifically, the implementation of technology plugins has proven to be important for detecting dependency conflicts, since they decide which options are added to the network and which are not. Moreover, technology plugins encode domain knowledge, such as type information for configuration options, which is used by the equality-based linker. That is, plugins not only influence how much information a configuration network contains, but also improve the linking process. Hence, plugins should be carefully developed and always incorporate domain knowledge if possible. So, only 10 % of all reported conflicts are false positives even when using the most simple linker heuristic.

7.2.2 Number of Detected Dependency Conflicts

Interestingly, the number of detected dependency conflicts, varies widely from system to system, thereby ranging from 0 to almost 10k detected dependency conflicts. We identified two reasons why we encountered so many dependency conflicts in certain subject systems. First, these systems are primarily implemented in Java, JavaScript, and TypeScript and incorporate technologies, such as Maven, Node.js, and TSconfig. Our current implementation already covers these language-specific technologies, enabling us to easily detect dependencies between them and other technologies used in a software system. Conversely, we found significantly fewer or no dependency conflicts in systems built using programming languages that lack language-specific technology plugins, such as Rust, Go, Clojure, or C++. The other reason for the high number of detected dependency conflicts in certain systems is in the nature of the detected dependency conflicts. The majority of detected dependency conflicts arose from version inconsistencies between libraries and configuration files. These dependency conflicts primarily occurred in large and complex subject systems, consisting of multiple submodules, each with its own configuration artifacts. Changing the version of a library in one module can lead to conflicts with all other modules that specified this library. Given that we analyzed large and complex systems, which often comprised thousands of commits, and that a single change can cause several dependency conflicts, explains the high number of detected dependency conflicts in some systems.

7.2.3 Performance

For each subject system, we also measured CFGNET's performance in constructing a network and in detecting conflicts for each commit that we analyzed during the commit history analysis. We report the average time needed to construct the network and to detect conflicts in Table 7. On average, our results show that CFGNET's network construction usually took less than a second and the conflict detection was always below half a second. However, we also noted 10 outliers for which the network construction took more than a second. We found that these subject systems contained dozens and even hundreds of configuration files per commits, which were parsed by our plugins. Naturally, parsing such large amount of configuration files per commit affects CFGNET's performance in constructing the corresponding network. Nevertheless, our results show that CFGNET scales across software systems that differ in size, programming language, and domain.

7.2.4 Summary

We found that our current implementation, which includes nine technology plugins and an equality-based linker, can find hundreds of potential configuration conflicts in real-world software projects. The diversity of detected cross-technology dependency conflicts emphasizes their significance and the need to reliably track configuration dependencies across the used technology stack to prevent their violations. Our manually analysis with about 89 % true positives of varying severity indicates not only the relevance of our approach for real-world projects, but also its applicability already in its current state. That is, although CFGNET is designed for extensibility and incremental accuracy improvement, it has already shown promising results. Moreover, we found that the false positives can be mainly traced back to the implementation of plugins due to a lack

of domain knowledge and tracking of unnecessary options. This confirms that plugins are the key success factor for our approach, as they not only determine how much information a network contains, but also support the linking process by providing type information for configuration options.

8 Discussion

What is CFGNet's application scenario? We envision that CFGNet is primarily used within a Git hook that targets commits to prevent dependency conflicts during the development and maintenance of software systems. That is, whenever changes are made, CFGNet checks the changes before the actual commit gets pushed to the repository and reports an error if it has detected possible dependency conflicts. So, developers would get immediate response and can check the changes again together with their domain knowledge to fix the dependency conflicts. This way, developers can proactively prevent possible dependency conflicts and save time in resolving complex dependency conflicts when using our framework. Our results show that CFGNet efficiently detects dependency violations and reports useful information to fix them, demonstrating its applicability already in its current state.

Why are technology plugins the key success of our framework? Our results strongly support our framework-based approach to track configuration dependencies with technology plugins by incorporating domain knowledge. Plugins specify not only how much information a configuration network contains, but also improve the accuracy of the linking process by making links type-aware. This way, we can go from a pure value-equality to value-type-equality. This is only possible if according information is available. Moreover, our plugin-based approach allows developers or tool vendors to easily develop and integrate their own technology plugins into CFGNet. We believe that the ability of CFGNet to incorporate such diverse solutions as technology plugins and to perform analyses on them, is a novel strength of our approach.

Also note that technology plugins affect the accuracy of our framework, and thus need to be carefully developed. Specifically, the accuracy of technology plugins partially depends on their implementation by the corresponding file parser and the domain knowledge added by the developer. For this reason, we aimed to select official file parsers or those that are appreciated by the GitHub community, indicating that the file parsers have proven to be reliable. We also tested each implemented technology plugin to ensure their accuracy in extracting configuration options and their values and translating them into the network structure.

Naturally, the more plugins there are, the more dependency conflicts can be potentially detected, as the network includes more nodes that might be related. However, the number of false positives are not directly linked to the number of plugins. First, it depends on how many technologies there are in a project. Second, it depends on whether proper plugins are applied, containing type information on options, or only general purpose plugins. The later ones are likely to cause more false positives as they have only the value of an option available. So, we would encourage using only technology-specific plugins. Third, using the equality-based linker without type information and without a white- or black-list can substantially increase false positives. This is why we proposed means to reduce this effect. Basically, forbidding the most common values (e.g., true, false, 1, 0) had a tremendous positive effect on the reduction of false positives, and we expect this to play out especially when more plugins are added. Naturally, as we design the framework to be extensible, we see further gains in lowering false positives with heuristic-based linkers. However, —and this cannot be stressed enough— research on such linkers requires a data set to begin with. This is what we can provide for the first time. So, we enable such research with our configuration network.

Increasing the number of plugins may increase the construction time of the network. However, the parsing process can easily be parallelized, effectively decoupling the number of plugins with parsing time. Only the linking process needs to be serialized, which is the fastest part of the network analysis.

How can other linker heuristics be derived and integrated as plugins into our framework? Integrating a linker is typically a straightforward task, since our linker interface clearly defines the requirements (i.e., methods) of the linker. That is, a linker plugin must first implement the linker heuristic, then traverse the value nodes, and create links when the linker criteria is met. The challenging part is, of course, when to create a link. This is a research on its own as this requires vast amount of data that is not available in a suitable form. Linker heuristics can be obtained through repository mining, static code analysis, or machine learning. For example, Santolucito et al. [15] applied a more expressive form of association rule learning (ARL) to mine various rules within a configuration file. We believe that ARL can also be employed to mine rules across the used technology stack of a software project, which can be leveraged by a linker as criteria to establish links between configuration options.

How do we envision the further development of our framework? Our vision for the further development of CFGNet involves a collaborative and transparent effort by a community of contributors, including developers, tool vendors, and researchers. We aim to create a framework similar to popular open-source projects, where contributions can be made by various actors and shared upon. This is much like GitHub actions where the community contributes their actions to support some technologies and use cases. So, our framework is very similar in this regard. For example, a product teams could develop plugins to cover technologies used in their software development process and apply CFGNet to tackle configuration dependencies introduced by these technologies. Since the member of product teams, such as developers and DevOps engineers, regularly work with various technologies, they already have the necessary expertise to develop new and maintain existing technology plugins. So, instead of a single-purpose research tool that has no maintenance as soon as the research is done, we publicize the framework and invite the community to build upon it. At the very least, CFGNet will allow for further research on this topic by providing a platform to collect and structure the configuration landscape of modern software projects.

What are the challenges to determine the severity of dependency conflicts? Determining the severity of the detected configuration dependency conflicts is challenging due to their complexity in terms of how and where they manifest. Configuration typically happens at different stages, such as development, testing, deployment, or even in production [1]. Dependency conflicts can thus occur at different stages of the development and deployment process, which is one reason why configuration errors are often undetected until a system goes into production. Moreover, dependency conflicts can lead to a wide range of issues, including system failures, performance degradation, inconsistent behavior, or difficulty in configuring software systems properly. Version inconsistencies, for instance, can cause compatibility issues, security threats, broken builds, or difficulty in managing and updating dependencies [35], [36], [37]. Additionally, the linearization of the commit histories limits the visibility of the effects of detected dependency conflicts, as checking the run of a CI server, if available, is not possible. However, as illustrated in our introductory example, dependency conflicts can go undetected by a CI server, and thus passing CI pipelines may give a false impression. Given these challenges, we relied on domain knowledge and related work to determine the severity of the detected configuration conflicts. Our results indicate that we found relevant configuration dependency conflicts of varying severity, highlighting their significance and the need to reliably track configuration dependencies across the used technology stack to avoid their violations.

How can detected dependency conflicts be prioritized for developers? Using CFGNET to analyze the entire commit history of software projects, as shown in Section 7, may result in hundreds or even thousands of dependency conflicts. While the large number of detected dependency conflicts can be primarily attributed to the size and number of involved technologies of the analyzed software projects and the nature of the detected dependency conflicts (cf. Subsection 7.2.2), it is unlikely that so many dependency conflicts will be detected if CFGNET is used to proactively prevent dependency conflicts during software development and maintenance. In cases when numerous dependency conflicts are detected, we can resort to prioritizing the conflicts. The idea involves a ranking of all conflicts based on specific heuristics. One heuristic can be derived from the frequency of how often the affected configuration artifacts have been changed in the past, since frequently changed configuration artifacts are more likely to lead to dependency problems. Another heuristic may be based on the effort required to resolve a dependency conflict in terms of how many options are involved and have to be changed to fix the conflict. For example, resolving version inconsistencies in multi-module software projects often requires changing several options encoded in configuration artifacts of different modules, which can be tedious and time-consuming. Also, this points to dependency conflicts that may be less severe as backward compatibility often ensures that the system is still working.

In addition, grouping the detected dependency conflicts into certain categories, can help developers in evaluating the importance and severity of the detected dependency conflicts. This grouping can be realized by the type of the dependency conflicts, similar to what we already did in the commit history analysis, or by the heuristics used for ranking the dependency conflicts.

What are the limitations of CFGNET? A first limitation of our framework comes from linking configuration options based the equality of their values, currently restricting CFGNET to a specific form of configuration dependencies. Moreover, linking configuration options based on value equality bears the risk of creating false positives, since not all configuration options with the same value have to depend on each other. To mitigate this risk, we added two mechanism to support the equality-based linker to create meaningful links. First, we extended the linker criterion by incorporating domain knowledge if available to prevent linking options whose types are different. Second, we added global and local blacklists, which allow developers to adapt the linking algorithm to limit links of configuration values that bear the risk of being involved in too many links. Both mechanisms already reduced the number of false positives to a rate of practicability.

Another limitation concerns unique option names, which are required by CFGNET to find the correct nodes in the network via tracing. This works well for regular configuration files that rely on key-value pairs, but not always for configuration files that behave like scripts, such as Dockerfiles. For example, Dockerfiles often specify multiple *COPY* commands that are run one after another when the container is deployed. Those commands are different configurations and do not even have to be in the same order, which makes it challenging to track them without additional information.

Moreover, our approach currently cannot reliably handle all cases of repetitive configuration options and options with multiple values, as multiple definitions of the same configuration option in a single file make it challenging to create unique option names. However, our framework can already model nested structures, such that nested regions are allowed to define equally named options. This way, we already represent specializations of configuration options. Another way to deal with repetitive options would be to consider the absolute positioning of options in a configuration file by considering their line number in the file. Then, the linker would have to be configured to, for example, match the first occurrence of the option, or to match all occurrences. Hence, our framework can already handle inner dependencies and can be extended to model more complex structures.

9 THREATS TO VALIDITY

There are primarily two external threats that arise from the selection of SO dependencies and subjects systems for both evaluations. We aim to increase external validity first by extracting real-world configuration dependencies reported by developers of SO. By selecting only those configuration dependencies that occurred in highly scored posts and were brought up in several posts, we evaluate our framework in a real setting. Second, rather than building toy projects to simulate dependency conflicts of the extracted dependencies, we selected five real-world projects from GitHub for the

dependency violation that involve the technologies of the extracted configuration dependencies. Finally, we selected 50 software repositories for the commit history evaluation, which vary in size, language, and domain, to ensure that our results can be generalized to other software projects.

A threat to internal validity depends on how the technology plugins parse configuration artifacts and construct the configuration network. To mitigate this threat, we studied the technologies, created unit tests for the corresponding technology plugins as well as relied on third-party tools to parse configuration files if possible. It is important to note that we first developed all technology plugins and only afterwards extracted dependency conflicts from SO posts. This way, we avoid bias in the implementation and mitigate the threat of construct validity.

The commit history evaluation contained the manual review of conflicts, which may be subjective and introduce researcher bias. Here, one author of the paper went through all commits involving a conflict, used his experience and domain knowledge to reason about the severeness of the conflicts, and finally marked the conflicts either clearly yes or no, and borderline. We mitigated this threat by having two authors reviewing unclear decisions. Moreover, both authors have experience with configuration artifacts and related technologies. On top of that, we provide all material online.

10 Conclusion

Modern software projects use various technologies and tools that have to be configured to work together. This often means that tools and frameworks have dependent configurations, possibly unaware to an individual developer. We propose CFGNET to model dependent configurations, enabling the tracing and detection of dependency violations. This allows us to inform developers when they change a configuration without changing dependent configurations and even suggest possible fixes. We designed CFGNET as a framework to enable tailor-made solutions for different technologies and dependency detection techniques in a common data structure by leveraging the plugin-based architecture. In a literature study, we found that the most common form of dependencies is the equality of values of different options. To this end, we developed an equality-based linker to determine dependent options across different artifacts. We also implemented nine technology plugins to demonstrate CFGNET's extensibility

To evaluate CFGNET, we collected five real-world configuration dependencies from Stack Overflow and violated these dependencies in five software projects. Our framework detected not only the injected dependency conflicts but also found four additional ones, demonstrating its practicality. Moreover, we analyzed the commit history of 50 repositories and found conflicts in about two thirds of these repositories, demonstrating the need to reliably track cross-technology configuration dependencies and prevent their misconfiguration. Finally, we manually reviewed 883 conflicts and showed the relevance and applicability of our approach for real-word software projects.

Acknowledgement

The work of the authors has been supported by the Federal Ministry of Education and Research of Germany and by the Sächsische Staatsministerium für Wissenschaft Kultur und Tourismus in the program Center of Excellence for AI-research "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig", project identification number: ScaDS.AI, and by the BMBF project Agile-AI (01IS19059B). Siegmund's work has been funded by the German Research Foundation (SI 2171/2-2).

References

[1] N. Siegmund, N. Ruckel, and J. Siegmund, "Dimensions of software configuration: on the configuration context in modern software development," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 338–349. [Online]. Available: https://doi.org/10.1145/3368089.3409675

[2] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172.

[3] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 1–41, Jul. 2015.

[4] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 362–374. [Online]. Available: https://doi.org/10.1145/3368089.3409727

[5] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 255–265. [Online]. Available: https://doi.org/10.1109/ICSE.2017.31

[6] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: Recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 796–801. [Online]. Available: https://doi.org/10.1145/3238147.3240470

[7] S. Simon, N. Ruckel, and N. Sigemund, "Cfgnet: A framework for tracking equality-based configuration dependencies across a software project," https://github.com/AI-4-SE/CfgNet, 2023.

[8] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 244–259. [Online]. Available: https://doi.org/10.1145/2517349.2522727

[9] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 619–634.

[10] S. Simon, N. Ruckel, and N. Siegmund, "Cfgnet: A framework for tracking configuration dependencies across a software project (supplementary website)," https://github.com/AI-4-SE/CfgNet-A-Framework-for-Tracking-Equality-Based-Configuration-Dependencies-Across-a-Software-Project, 2023.

[11] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/osdi10/automating-configuration-troubleshooting-dynamic-information-flow-analysis

[12] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, ser. ASE '11, IEEE. USA: IEEE Computer Society, 2011, pp. 193–202. [Online]. Available: https://doi.org/10.1109/ASE.2011.6100053

[13] S. Zhang, "Confdiagnoser: An automated configuration error diagnosis tool for java software," in *2013 35th International Conference on Software Engineering (ICSE)*, ser. ICSE '13, IEEE. San Francisco, CA, USA: IEEE Press, 2013, pp. 1438–1440.

[14] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Proceedings of the 28th International Conference on Computer Aided Verification, Part II*. Toronto, ON, Canada: Springer, 2016, pp. 80–87.

[15] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[16] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 687–700.

[17] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 295–306.

[18] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–25.

[19] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th international conference on Autonomic computing*, ser. ICAC '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 169–178. [Online]. Available: https://doi.org/10.1145/1555228.1555269

[20] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643001

[21] C. A. Metcalf, F. Fowze, T. Yavuz, and J. Fortes, "Extracting configuration parameter interactions using static analysis," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2016, pp. 1–4.

[22] X. Liao, S. Zhou, S. Li, Z. Jia, X. Liu, and H. He, "Do you really know how to configure your software? configuration constraints in source code may help," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 832–846, 2018.

[23] T. Mahmud, D. Zhang, O. R. Gatla, and M. Zheng, "Understanding configuration dependencies of file systems," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 1–8.

[24] B. Kitchenham and S. M. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[25] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of systems and software*, vol. 80, no. 4, pp. 571–583, 2007.

[26] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[27] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[28] R. Bhagwan, S. Mehta, A. Radhakrishna, and S. Garg, "Learning patterns in configuration," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 817–828.

[29] W. Chen, H. Wu, J. Wei, H. Zhong, and T. Huang, "Determine configuration entry correlations for web application systems," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 42–52.

[30] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 88–97.

[31] O. Tuncer, A. Byrne, N. Bila, S. Duri, C. Isci, and A. K. Coskun, "Confex: a framework for automating text-based software configuration analysis in the cloud," *arXiv preprint arXiv:2008.08656*, 2020.

[32] I. Stack Exchange, "Stack Exchange Data Dump," Jun. 2021. [Online]. Available: https://archive.org/details/stackexchange

[33] M. Bagherzadeh and R. Khatchadourian, "Going big: a large-scale study on what big data developers ask," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 432–442.

[34] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 750–762.

[35] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, 2021.

[36] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 349–359.

[37] M. Santolucito, J. Zhang, E. Zhai, and R. Piskac, "Statically verifying continuous integration configurations," *arXiv preprint arXiv:1805.04473*, 2018.

[38] X. Chen, R. Abdalkareem, S. Mujahid, E. Shihab, and X. Xia, "Helping or not helping? why and how trivial packages impact the npm ecosystem," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–24, 2021.

Sebastian Simon is a Ph.D. student and a research assistant at the Chair of Software Systems, Leipzig University, Leipzig, Germany. His research interests include configurable software systems, configuration dependencies, and the configuration of machine learning projects and experiments. Simon received his B.Sc. in 2018 and M.Sc. in 2020 from the Otto-von-Guericke University of Magdeburg. Contact him at sebastian.simon@informatik.uni-leipzig.de.

Nicolai Ruckel is a software developer at Secunet. He received his B.Sc. in 2017 and his M.Sc. in 2018 from Bauhaus-Universität Weimar. Contact him at nicolai.ruckel@posteo.de.

Prof. Dr. Norbert Siegmund holds the Chair of Software Systems at Leipzig University, Germany. Prof. Siegmund received his PhD with distinction in 2012 from the University of Magdeburg. His research aims at the automation of software engineering by combining methods from software analysis and machine learning. His special interests include configurable software systems, performance, energy optimization, and SE4AI. Contact him at norbert.siegmund@uni-leipzig.de.