Automated Mitigation of Confidentiality Violations in Software Architectures using Discrete Optimization

Master's Thesis of

Benjamin Arp

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf H. Reussner
Second examiner:	Prof. Dr. Robert Heinrich
First advisor:	M.Sc. Nils Niehues
Second advisor:	M.Sc. Nicolas Boltz

18. September 2025 – 18. March 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Identifying confidentiality violations in software architectures at design time is a well-studied problem. Fixing them automatically is not. Frameworks such as STRIDE and LINDDUN use Data Flow Diagrams (DFDs) to expose threats at an early stage in the development lifecycle. But once a violation is detected, software architects must still manually determine and apply the appropriate countermeasures, which is both a time-consuming and error-prone process.

This thesis addresses the repair side of the problem. The central research question is which discrete optimisation method is best suited to automate this task. To answer it without presupposing an outcome, we design a comparative survey of three candidate methods: Branch and Bound, Integer Linear Programming (ILP), and Evolutionary Algorithms. These are evaluated against four criteria: optimality, runtime performance, extensibility, and reproducibility. The survey establishes ILP as the only method satisfying all four, primarily because its declarative problem formulation separates constraint specification from solving. This separation proves essential when mitigation strategies must be extended or customised.

Building on this result, we implement an ILP-based automated mitigation approach integrated into the ARCoViA framework. The approach operates on DFDs that are annotated with label-based confidentiality constraints. It then enumerates candidate mitigation strategies across the full space of label additions and deletions, node insertions and removals, as well as flow deletions. These strategies, along with their mutual dependencies and contradictions, are encoded into a Boolean ILP problem. The solver yields a minimal-cost repair, which is then applied to produce a repaired DFD automatically. The main engineering challenge is generating a complete and correct set of candidate mitigations and encoding their dependencies and contradictions without omissions.

The prior SAT-based approach, which this work extends, is limited to purely additive label changes and struggles to express richer constraint structures in CNF form. The present approach removes both restrictions. We evaluate it against four goals: effectiveness, extensibility, cost, and scalability, using DFD models from the MicroSecEnD benchmark. The approach eliminates all detected violations, produces repairs that are approximately 73% less invasive than a human baseline, and scales acceptably across all studied dimensions of model complexity.

Zusammenfassung

Die Identifizierung von Vertraulichkeitsverletzungen in Softwarearchitekturen zur Entwurfszeit ist ein gut erforschtes Problem. Ihre automatische Behebung hingegen nicht. Methoden wie STRIDE und LINDDUN verwenden Datenflussdiagramme (DFDs), um Bedrohungen in einer frühen Phase des Entwicklungszyklus sichtbar zu machen. Sobald jedoch eine Verletzung erkannt wird, müssen Softwarearchitekten nach wie vor manuell geeignete Gegenmaßnahmen bestimmen und anwenden: ein Prozess, der sowohl zeitaufwändig als auch fehleranfällig ist.

Diese Arbeit adressiert die Reparaturseite dieses Problems. Die zentrale Forschungsfrage lautet, welche diskrete Optimierungsmethode am besten geeignet ist, diese Aufgabe zu automatisieren. Um sie ohne Vorwegnahme eines Ergebnisses zu beantworten, entwerfen wir eine neutrale vergleichende Untersuchung dreier Kandidaten: Branch and Bound, Integer Linear Programming (ILP) und evolutionäre Algorithmen. Diese werden anhand von vier Kriterien bewertet: Optimalität, Laufzeitleistung, Erweiterbarkeit und Reproduzierbarkeit. Die Untersuchung zeigt, dass ILP die einzige Methode ist, die alle vier Kriterien erfüllt, in erster Linie, weil die deklarative Problemformulierung die Regelspezifikation von der Lösung trennt, eine Trennung, die sich als wesentlich erweist, wenn Mitigationsstrategien erweitert oder angepasst werden müssen.

Aufbauend auf diesem Ergebnis implementieren wir einen ILP-basierten automatisierten Mitigationsansatz, der in das ARCoViA-System integriert ist. Der Ansatz operiert auf DFDs, die mit labelbasierten Vertraulichkeitseinschränkungen annotiert sind. Er enumeriert Kandidaten-Mitigationsstrategien über den gesamten Raum von Label-Additionen und -Löschungen, Knoten-Einfügungen und -Entfernungen sowie Fluss-Löschungen. Diese Strategien werden zusammen mit ihren gegenseitigen Abhängigkeiten und Widersprüchen als boolesches ILP-Problem kodiert. Der Solver liefert eine kostenminimale Reparatur, die anschließend automatisch auf das DFD angewendet wird. Der technisch anspruchsvollste Aspekt dieses Entwurfs ist die Problemraumexploration, die die korrekte Enumeration aller anwendbaren Mitigationen sowie die präzise Repräsentation ihrer Beziehungen in der ILP-Kodierung umfasst.

Der vorherige SAT-basierte Ansatz, den diese Arbeit erweitert, ist auf rein additive Label-Änderungen beschränkt und kann komplexe Regelstrukturen in CNF-Form nur unzureichend ausdrücken. Der vorliegende Ansatz hebt beide Einschränkungen auf. Wir evaluieren ihn anhand von vier Zielen: Effektivität, Erweiterbarkeit, Kosten und Skalierbarkeit, unter Verwendung von DFD-Modellen aus dem MicroSecEnD-Datensatz. Der Ansatz eliminiert alle erkannten Verletzungen, erzeugt Reparaturen, die etwa 73% weniger invasiv sind als

eine menschliche Baseline, und skaliert akzeptabel über alle untersuchten Dimensionen der Modellkomplexität hinweg.

Contents

List of Figures

List of Tables

Acronyms

APR Automated Program Repair

B&B Branch and Bound

CNF Conjunctive Normal Form

DFA Data Flow Analysis

DFD Data Flow Diagram

DSL Domain Specific Language

ILP Integer Linear Programming

IP Integer Programming

ML Machine Learning

TFG Transposed Flow Graph

SAT Boolean Satisfiability Solving

SPL Software Product Line

GDPR General Data Protection Regulation

GQM Goal Question Metric

1 Introduction

Software architects bear the responsibility of ensuring that sensitive data remains protected throughout a system's design, long before any code is written. Frameworks such as STRIDE and LINDDUN use Data Flow Diagrams (DFDs) as a core modelling technique, exposing how data traverses a system and where it may be disclosed to unauthorised parties [42]. These frameworks offer support for the systematic identification of potential threats; however, they provide only very limited support for their mitigation, if any. Once a threat or violation has been identified, architects must manually select and apply appropriate countermeasures such as encryption, access control, or restructuring of data flows. A process that is time-consuming and prone to errors.

In the context of threat modelling, confidentiality is particularly critical. It is an essential security property, as stipulated by regulations such as the GDPR [51] and ensures that data is accessible only to authorised parties [49]. In contrast to continuous quality attributes such as performance, confidentiality is discrete and rule-based. A system either satisfies a given confidentiality requirement, or it does not [3, 28]. This binary nature makes confidentiality well-suited to formal constraint-based reasoning, but it also means that the space of possible repairs consists of discrete architectural decisions. Since each potential mitigation is an independent choice and multiple mitigations may interact, the number of valid repair combinations grows exponentially with system size [25, 24], making a manual search for an optimal repair infeasible in practice.

1.1 Problem Statement

Existing approaches to architecture-based security analysis have made detection reliable, but the step from detecting a violation to repairing it automatically remains largely unsolved. The central question this thesis examines is which discrete optimisation method is most suitable for automating this repair step.

Recent work has begun to address this challenge, yet existing approaches still fall short of an ideal solution. Niehues et al. [33] employ machine learning to rank uncertainty sources in software architectures based on their estimated impact on confidentiality. While effective at reducing the search space, this approach requires labelled training data that must be produced through manual system configuration or expert-driven simulation, making it semi-automated at best. Furthermore, machine learning cannot provide guarantees of minimal or optimal mitigation, and the remaining search space may still grow exponentially

for complex systems with many interacting uncertainties, making the approach impractical for larger systems.

To address these limitations, Niehues et al. [32] reframe the problem as an instance of Boolean Satisfiability, translating architectural confidentiality constraints into Conjunctive Normal Form (CNF) and solving them using a Boolean Satisfiability Solving (SAT) solver. This approach is notable for its completeness and efficiency. Nevertheless, the SAT-based approach presents drawbacks that limit its general applicability. One key issue is structural inflexibility: it only allows additive changes, such as adding encryption or logging annotations, and does not support the removal of nodes, labels, or flows. This restriction precludes simpler or lower-cost repairs, particularly when a component is unnecessary, redundant, or structurally misplaced. Additionally, expressing richer constraint interactions in CNF form limits the extensibility of the approach to more complex confidentiality requirements.

These findings reveal a concrete gap in the current state of the art. No existing approach supports fully automated, cost-minimal repair of confidentiality violations that encompasses both additive and structural modifications. It also remains an open problem to identify and validate the best-suited discrete optimisation strategy for this task.

1.2 Contributions

The thesis makes two main contributions to the field of architecture-based security analysis.

Survey and Evaluation of Discrete Optimisation Methods. This thesis provides a systematic survey and comparative evaluation of discrete optimisation methods for the automated repair of confidentiality violations in DFDs. Three candidate methods, Integer Linear Programming (ILP), Branch and Bound (B&B), and Evolutionary Algorithms, are evaluated against four criteria: optimality, runtime performance, extensibility, and reproducibility. A formal optimisation objective is defined that balances violation coverage, minimality of architectural changes, and implementation cost. The survey establishes ILP as the uniquely suitable method and provides a grounded, criterion-based justification for this selection.

Automated Mitigation Approach. This thesis develops and implements a concrete ILP-based mitigation approach as part of the ARCoViA framework, the same toolchain in which the prior Machine Learning (ML)-based and SAT-based approaches are realised [33, 32]. In contrast to prior work restricted to additive label changes, the approach supports the full range of structural modifications: label additions and deletions, node insertions and removals, as well as flow deletions. User-defined strategies and domain-specific cost models are supported without modifying the solver. The approach is validated on realistic DFD models drawn from the MicroSecEnD benchmark, demonstrating reliable violation elimination, a 73 % reduction in repair invasiveness over a human baseline, and acceptable scalability across all studied dimensions of model complexity.

1.3 Structure of Thesis

The remainder of the thesis is organized as follows. Chapter 2 introduces the foundational concepts, covering confidentiality, threat modelling frameworks (STRIDE and LINDDUN), Data Flow Diagrams and Transposed Flow Graphs, the Data Flow Analysis framework xDECAF and its constraint DSL, and the discrete optimisation foundations (ILP, SAT solving, and the Set Cover problem). Chapter 3 surveys related work across six areas: architecture-based confidentiality analysis, DFD modelling, automated program repair, software product lines, self-adaptive systems, and search-based optimisation. Chapter 4 presents the running example used throughout the thesis, defining the DFD of a simple online service platform along with its constraints, violations, and repaired model. Chapter 5 reports a comparative survey of discrete optimisation methods (Branch and Bound, ILP, and Evolutionary Algorithms), evaluated against four selection criteria, and motivates the choice of ILP. Chapter 6 details the automated mitigation approach, describing the full mitigation workflow, the supported strategy families, and the system architecture. Chapter 7 evaluates the approach with respect to effectiveness, extensibility, cost, and scalability, and discusses threats to validity and limitations. Chapter 8 concludes the thesis and provides an outlook on future work.

2 Foundations

This chapter introduces the background required to understand the approach and evaluation presented in later chapters. Readers already familiar with DFD-based confidentiality analysis may proceed directly to Section 2.4; those familiar with discrete optimisation foundations may skip to Chapter 3.

We begin with the security property that motivates the entire work: confidentiality. We then describe DFDs and their derived Transposed Flow Graphs (TFGs), followed by the Data Flow Analysis (DFA) framework xDECAF and its constraint Domain Specific Language (DSL). Those provide the analytical results on which the mitigation approach is built. Finally, we present the optimisation and complexity-theoretic foundations, including discrete optimisation, ILP, SAT solving, and the *set cover problem*, which underpin the mitigation approach developed in Chapter 6.

2.1 Confidentiality

Confidentiality is one of the three main properties of information security, alongside integrity and availability, commonly referred to as the *CIA triad* [40]. It denotes the property that information is accessible only to those authorised to access it [49]. In the context of software systems, confidentiality requires that sensitive data, such as personal records, authentication credentials, or business-critical information, is neither disclosed to unauthorised parties nor transmitted through channels that lack appropriate protection.

Unlike continuous quality attributes such as performance or latency, confidentiality is inherently discrete [46]. A system either satisfies a given confidentiality requirement or it does not; there is no notion of "partial" confidentiality. This binary nature means that confidentiality is easily subject to constraint satisfaction: a requirement is either encoded as a constraint that holds, or violated. In this thesis, confidentiality requirements are expressed as constraints over architectural elements and data flows, as formalised in later chapters.

2.2 Data Flow Diagrams

DFDs are a lightweight yet expressive modelling formalism to describe how data flows through a software system. Originally introduced by DeMarco, they represent the system as a network of nodes, external entities, processes, and data stores, connected by directed flows

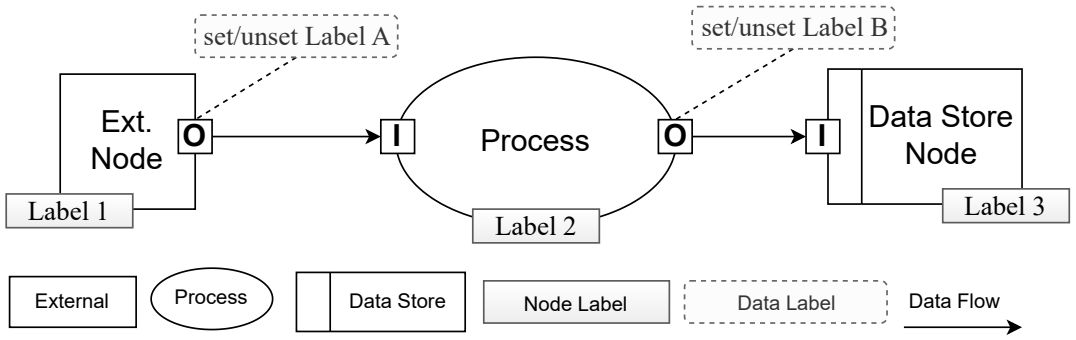

Figure 2.1: DFD syntax used in this work

of data [12, 7] (see Figure 2.1). This abstraction allows reasoning about system behaviour early in design, without requiring full implementation details.

For confidentiality analysis, DFDs are extended with *labels* that capture security-relevant properties of data (e.g., `personal`, `encrypted`) and components (e.g., `internal`, `non-EU`). During automated analysis, these labels propagate along flows, enabling the detection of confidentiality violations such as unencrypted personal data leaving trusted boundaries [7, 46].

The goal of using DFDs in this context is therefore twofold: first, to provide a clear, design-level representation of software architectures that enables early identification of confidentiality risks; and second, to serve as a formal basis for automated reasoning, where confidentiality requirements can be expressed as constraints over nodes, flows, and labels. In this way, DFDs facilitate architectural analysis and provide the natural foundation for automated mitigation strategies that adjust architectural options to restore confidentiality [7].

To support scalable analysis, recent work introduced TFGs, which represents the individual end-to-end data paths within a DFD [7]. A TFG is constructed by tracing, for each piece of data, the complete sequence of nodes and flows from its source to its final sink. Each such path constitutes an independent TFG that preserves the label semantics of the original DFD while isolating a single data trajectory.

The decomposition into TFGs offers two principal advantages. First, it enables *local reasoning*: a confidentiality violation can be localised to a specific TFG and thereby to a specific data path, rather than requiring a global analysis of the entire diagram. Second, it improves *scalability*: because constraints can be evaluated independently for each TFG, the analysis effort grows linearly with the number of data paths rather than combinatorially with the number of nodes and flows.

Throughout this thesis, violations are reported per TFG, mitigation strategies are applied along TFG paths, and scalability is evaluated by varying the length and quantity of TFGs.

2.3 Data Flow Analysis Framework and Constraint DSL

The automated confidentiality analysis that underpins this thesis builds upon the DFA framework proposed by Boltz et al. [7]. This framework provides two core capabilities: (i) a label-propagation engine that computes, for every node and flow in the DFD, which data labels and node characteristics are present, and (ii) a DSL for specifying confidentiality constraints over these labels.

2.3.1 Label Propagation

The analysis engine processes each TFG independently. Starting from the source node, it propagates the labels attached to data elements along the directed flows. At each intermediate node, the engine applies the node's behaviour specification, which may add, remove, or forward labels, and continues propagation until the sink node is reached. The result is a complete label assignment for every element along the path, which serves as the input for subsequent constraint evaluation [7, 44].

2.3.2 Constraint Specification

Confidentiality constraints are expressed in a declarative DSL that follows a *neverFlows* pattern. A constraint specifies conditions under which data with certain labels must never reach nodes with certain characteristics. For instance, the expression

```
data().neverFlows().toVertex()
    .withCharacteristic("Stereotype", "internal")
    .withoutCharacteristic("Stereotype", "local_logging")
```

states that data may not flow to an `internal` vertex that lacks the `local_logging` stereotype. Any TFG whose label assignment violates such a constraint is reported as a violation and associated with the offending vertex [7].

The DSL exposes four filtering predicates, split between the two selector types it provides:

- `data().withLabel(`*`type`*`, `*`value`*`)`: restricts the data selection to flows that carry the label *value* of label type *type*. Labels are assigned to data and are propagated through the TFG during analysis. They represent data classifications such as sensitivity or encryption state [7].
- `data().withoutLabel(`*`type`*`, `*`value`*`)`: selects only data flows that do *not* carry the specified label, for example, to target unencrypted or unanonymised data [7].

- `toVertex().withCharacteristic(type, value)`: restricts the vertex selection to nodes whose defined node characteristics include *value* for characteristic type *type*. Node characteristics are properties annotated directly to nodes in the architectural model, independently of the data flowing through them, such as a deployment location or a processing stereotype [7].
- `toVertex().withoutCharacteristic(type, value)`: selects only vertices that do *not* hold the specified node characteristic, such as nodes lacking a `local_logging` stereotype as illustrated above [7].

Multiple predicates of the same kind can be chained to form conjunctive selections, requiring a datum or vertex to satisfy all stated conditions simultaneously. A constraint can thereby express, for example, that data labelled `Personal` and lacking the label `Encrypted` must never flow to a vertex carrying the characteristic `offPremise` [7].

For constraints that exceed the expressiveness of the label-based DSL, for example, cross-TFG reachability conditions, the xDECAF framework allows custom *evaluation functions* that implement arbitrary analysis logic.

2.4 Discrete Optimisation

Discrete optimisation is a branch of mathematical optimisation where the decision variables are restricted to a discrete set of values (often integers), in contrast to continuous optimisation, which allows real-valued variables [23, 35]. This means the solution space of a discrete problem is typically finite but combinatorially large. Many classical discrete optimisation problems, such as the Travelling Salesman or Knapsack, are NP-hard, lacking efficient algorithms for their optimal solution [59]. The general goal in discrete optimisation is to find an optimal configuration of discrete decision variables that maximises or minimises a given objective function while satisfying all problem-specific constraints. Discrete optimisation aims to balance solution optimality with computational feasibility, which has led to both exact algorithms and approximation or heuristic methods [59, 35].

2.4.1 Exact Methods

Exact algorithms systematically explore the solution space to find a truly optimal solution. To avoid exhaustive enumeration, clever search strategies and mathematical formulations are used to prune the search tree [26, 23]. However, this may require exponential time in the worst case due to the nature of NP-hard problems.

One exact method is B&B, a tree-based search technique that branches the solution space into subproblems and uses bounds on the objective value to prune away suboptimal regions. This can reduce the search space that has to be explicitly examined [26]. Another exact method is to formulate the problem as an ILP and solve it using Integer Programming (IP) techniques, as described in detail in Section 2.5.

2.4.2 Heuristic Methods

When exact solvers are infeasible due to their time consumption, heuristic algorithms aim for good solutions with improved scalability. For example, local search iteratively improves an existing solution by exploring its neighbourhood; however, this can lead to convergence to local optima only. To address this, metaheuristics like Simulated Annealing [22] and Tabu Search [16] introduce randomness, memory, or population-based strategies to explore the search space more effectively. These methods sacrifice optimality guarantees for speed and scalability, making them indispensable for large-scale or real-time problems.

2.4.3 Evolutionary and Population-Based Algorithms

Another family of optimisation algorithms are evolutionary and population-based algorithms, which work with a population of candidate solutions rather than a single current solution. They maintain a population of candidate solutions and iteratively apply variation operators (crossover, mutation) and selection mechanisms to evolve increasingly better solutions. Well-known representatives include Genetic Algorithms, Genetic Programming, and Evolution Strategies [11]. Evolutionary approaches are particularly effective for large, complex search spaces where traditional heuristics struggle to maintain diversity [11, 4].

2.5 Integer Linear Programming

Integer Linear Programming (ILP) is a mathematical optimisation paradigm in which an objective function and all constraints are linear, but some or all decision variables are restricted to integer values [59, 30]. A general ILP can be stated as:

$$\begin{aligned} \text{minimise} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \in \mathbb{Z}^n, \end{aligned} \tag{2.1}$$

where $\mathbf{c} \in \mathbb{R}^n$ is the cost vector, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, $\mathbf{b} \in \mathbb{R}^m$ is the right-hand-side vector, and $\mathbf{x}$ is the vector of integer decision variables. When all variables are further restricted to $\{0, 1\}$, the problem is referred to as a *Binary Integer Programme* or *0-1 ILP* [59].

Modern ILP solvers combine several techniques to find provably optimal solutions efficiently [59, 30]:

- **LP Relaxation.** The integrality constraints are relaxed, allowing real-valued solutions. The resulting linear programme can be solved in polynomial time and provides a lower bound on the optimal integer objective value. If the relaxed solution happens to be integral, it is optimal for the original ILP.

- **Cutting Planes.** Additional linear inequalities (cuts) are iteratively added to tighten the LP relaxation without eliminating any feasible integer solution. This reduces the gap between the relaxed and integer optima, enabling faster convergence [30].
- **Branch-and-Cut.** This method combines the B&B tree search with cutting-plane generation at each node. It is the predominant strategy in state-of-the-art solvers and achieves strong practical performance on a wide variety of problem structures [59].

The declarative nature of ILP provides a clear separation between problem specification and solution algorithm: new constraints or variables can be added to the model without modifying the solver itself. This property makes ILP particularly suitable for problems where requirements evolve over time, as is the case for confidentiality mitigation.

2.6 Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) asks whether there exists an assignment of truth values to a set of Boolean variables such that a given propositional formula evaluates to *true* [6]. SAT was the first problem proven to be NP-complete, as established by Cook [10], forming the foundation of computational complexity theory.

2.6.1 Conjunctive Normal Form

Most modern SAT solvers require the input formula to be in Conjunctive Normal Form (CNF). A formula in CNF is a conjunction (logical *AND*) of *clauses*, where each clause is a disjunction (logical *OR*) of *literals* (variables or their negations) **[10]**. Formally, a CNF formula over Boolean variables $x_1, \ldots, x_n$ has the form:

$$\varphi = \bigwedge_{i=1}^{m} \left(\ell_{i,1} \vee \ell_{i,2} \vee \cdots \vee \ell_{i,k_i} \right), \tag{2.2}$$

where each literal $\ell_{i,j}$ is either a variable x_p or its negation $\neg x_p$. Any propositional formula can be converted into an equisatisfiable CNF formula using the Tseitin transformation [6].

2.6.2 Solving Approaches

State-of-the-art SAT solvers are based on the *Conflict-Driven Clause Learning* (CDCL) paradigm, which extends the classical Davis–Putnam–Logemann–Loveland (DPLL) algorithm with intelligent backtracking and clause learning [6]. When the solver encounters a conflict during search, it analyses the conflict to derive a new clause that prevents the same conflict from recurring, thereby pruning the search space.

Despite the NP-completeness of SAT in the worst case, modern CDCL solvers can handle instances with millions of variables in practice, due to sophisticated preprocessing, efficient data structures (e.g., watched literals), and restart strategies [14]. In the context of this thesis, SAT serves as the baseline approach: the prior SAT-based work encodes the architectural confidentiality repair problem in CNF and solves it with a SAT solver. The evaluation chapter (Chapter 7) compares the ILP-based approach developed here against this SAT-based baseline across multiple scalability dimensions.

2.7 Set Cover Problem

The *Set Cover Problem* is a classical combinatorial optimisation problem that arises naturally in the context of this thesis. Given a universe $U = \{e_1, e_2, \ldots, e_n\}$ of elements and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U such that $\bigcup_{j=1}^{m} S_j = U$, the Set Cover Problem asks for the smallest sub-collection $\mathcal{S}^* \subseteq \mathcal{S}$ whose union still equals U [21, 52].

In the *Weighted* variant, each set S_j has an associated cost $c_j > 0$, and the objective is to minimise the total cost $\sum_{S_j \in \mathcal{S}^*} c_j$ while covering all elements [9]. The decision version of Set Cover is NP-complete [21], and the optimisation version cannot be approximated to within a factor of $(1 - \varepsilon) \ln n$ for any $\varepsilon > 0$ unless P = NP [52].

The connection to the mitigation problem studied in this thesis is direct. The universe U corresponds to the set of confidentiality violations, the collection $\mathcal{S}$ corresponds to the set of available mitigation strategies (each of which resolves a subset of violations), and the cost c_j reflects the invasiveness of each strategy. The ILP formulation presented in Chapter 6 is therefore a *Weighted Set Cover* instance enriched with additional constraints (contradictions, dependencies) that reflect the architectural and logical relationships between mitigations. This connection allows the approach to benefit from the extensive body of research on Set Cover, including efficient solver techniques developed for this well-studied problem class [59, 52].

3 Related Work

Enforcing confidentiality in software architecture is related to multiple research areas. The following sections introduce the most relevant of these areas and discuss how each relates to the automated mitigation approach developed in this thesis.

3.1 Architecture-Based Confidentiality Analysis

A well-established line of research focuses on detecting confidentiality violations at the architectural design level. Seifermann et al. [46, 45] propose data flow-based analyses for software architectures to identify violating information flows and data leaks early in the design process. These approaches model how data propagates through components and deploy contexts, flagging any path that violates confidentiality requirements. A related line of research augments architecture models with access control constraints: Walter et al. [54] defines architecture-driven role-based access control checks to detect design-time privilege violations, later extending this with uncertainty modelling to assess confidentiality under incomplete knowledge of deployments or implementations [55]. More broadly, architecture-level security modelling includes approaches such as UMLsec [19] and SecDFD [50, 36]. While these methods effectively detect potential confidentiality flaws before implementation, they provide little support for automatically repairing them. When a violation is found, architects must still manually adjust the model. This thesis addresses precisely this gap by automating the repair step that architecture-based analysis currently leaves to human judgment.

3.2 Data Flow Diagrams

DFDs are a common foundation for threat modelling methods like STRIDE and LINDDUN, as they capture how data moves through a system [47, 53]. However, the standard DFDs notation is informal and cannot express security properties directly, making systematic security analysis difficult without expert insight. Recent research has therefore extended DFDs-based modelling with formal security semantics to support automated threat detection. One approach adds confidentiality and integrity labels to DFDs models and applies formal information-flow analysis to detect design-level security flaws early [50]. Another work proposes a notation enriched with security information to ensure that implementations comply with the secure data-flow constraints of the architecture [7]. Such methods introduce

rigour into DFDs-centric threat modelling and help bridge the gap between high-level architectural analysis and code-level enforcement of security requirements.

Recent advances enable automatic extraction of security-enriched DFDs from source code. Schneider et al. [41] present a tool-supported technique to automatically extract DFDs from microservice implementations, achieving 93% precision and 85% recall on Java applications. Their approach parses source code and configuration files using keyword-based evidence extraction, producing fully-fledged DFDs with security annotations and full traceability to code. Such extraction techniques provide realistic, large-scale DFD datasets, but currently stop after generating models. Our optimisation-based mitigation approach complements these extraction tools by automatically proposing repair strategies on the extracted DFDs, transforming purely diagnostic workflows into end-to-end architectural support.

3.3 Automated Program Repair

Automated Program Repair (APR) techniques aim to generate fixes for software bugs automatically, and researchers have begun applying them to security vulnerabilities as well. APR methods range from heuristic search-based approaches, such as evolutionary patch generation guided by tests, to semantic program analysis and learning-based techniques [56, 27]. Modern APR tools have succeeded in fixing many general bugs, though they often struggle with complex multi-hunk issues.

Security vulnerabilities, by contrast, frequently require only localised code changes, making them a promising target for automation. An empirical study on real-world Java vulnerabilities found that state-of-the-art repair tools could produce a patch for roughly 20 % of the vulnerabilities studied, though less than half of those candidate patches were actually correct and preserved functionality [8]. The central insight: that fixing errors can be at least partially automated, transfers directly to the architectural level. Confidentiality violations in an architecture model can be understood as design-level bugs requiring repair. This thesis realises an architectural analogue of APR: instead of source code edits, it applies model-level changes such as re-routing data flows, inserting encryption components, or adjusting deployment assignments to eliminate security weaknesses.

3.4 Software Product Lines and Variability

The need to consider many possible architecture variants for confidentiality is closely related to Software Product Line (SPL) engineering. SPL provides methods for managing variability by treating system variants as combinations of features. A foundational approach by Kang et al. [20] introduced feature models to represent these product families. Modern SPL tools like FeatureIDE encode feature selections and constraints, for example requires or excludes relations, as propositional logic and often use SAT solvers to verify consistency or derive products. Recent advances even integrate SMT solvers to handle numerical attributes in

feature models, extending their analytical power [48]. These logic-based techniques can efficiently enumerate or analyse thousands of possible configurations. However, mapping high-level variability models into pure feature models can introduce information loss or oversimplify constraints [13]. In the context of confidentiality, this limitation is significant: confidentiality constraints consist of predicates that go beyond simple feature toggles and cannot be faithfully represented in standard SPL formalisms.

3.5 Self-Adaptive Systems

Self-adaptive software systems modify their behaviour or structure at runtime in response to changes in their environment or internal state. Research in this field has produced frameworks that monitor a running system and perform adaptations through a feedback loop. The Rainbow framework, for example, uses architectural models at runtime to decide and enact adaptations when Quality of Service conditions are violated [15]. The planning component of self-adaptive systems, which selects reconfigurations in response to policy violations, is conceptually related to the automated repair approach developed in this thesis. However, existing self-adaptive research has focused predominantly on properties such as performance efficiency or fault tolerance [57, 58]; security and confidentiality have received comparatively little attention in the adaptation community. This thesis operates at design time rather than runtime, but the underlying challenge of selecting a correct and cost-minimal reconfiguration in response to a policy violation is shared with the self-adaptive systems literature.

3.6 Search-based and Evolutionary Optimisation

Across the areas above, a unifying technical thread is the use of search and optimisation to navigate large design spaces. Search-Based Software Engineering is an established paradigm that applies metaheuristic search to software design and analysis problems [17, 18]. In the context of architecture and security, evolutionary techniques have been applied to design space exploration, for instance, to generate architecture variants that satisfy multiple quality objectives [55]. This demonstrates that evolutionary approaches can effectively search vast and complex solution spaces. However, pure evolutionary methods are poorly suited to the strict correctness requirements of security repair: a random mutation might resolve one violation while inadvertently introducing another.

4 Running Example

In this chapter, we introduce the running example referenced throughout this thesis. The example is modelled as a DFD (see Figure 4.1). It represents a simplified online service platform that provides access to internal services for regular users as well as administrative functionality for elevated users.

The system environment implements two primary actors: users and administrators. Users interact with the platform via a single entrypoint, whereas administrators possess elevated privileges and may configure services or issue commands through dedicated interfaces. The model is intentionally kept compact, with the intention of clearly illustrating data flows and security-relevant properties.

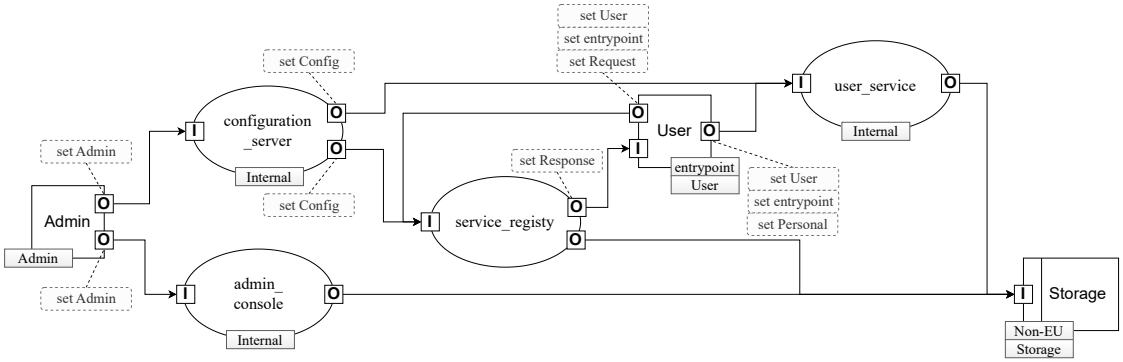

Figure 4.1: DFD of Example Model

The system shown in Figure 4.1 consists of the following main elements:

- **External entities:** User and Admin
- **Internal processing nodes:** a configuration server, a service registry, a user service, and an admin console.
- **Storage components:** a persistent data storage.

Overall, the model contains 5 TFGs, each representing an independent data flow of the system. The first flow represents the primary user interaction and starts at the User entrypoint, which serves as the sole interface for regular users. User requests are forwarded to the service registry, which acts as a directory for available services. Based on the request, the registry resolves the referred internal user service and responds to the user.

In the second TFG, the user sends personal data to the internal user service, where data is processed. The resulting output and operational data is then stored in the storage component.

In a separate set of flows, administrators interact with the system in two distinct ways. First, configuration data is sent to the *configuration_server*, which is responsible for updating *user_service* configurations and *Service_registry* entries. These flows give rise to two independent TFGs, each of which terminates at the storage component.

Lastly, administrators can issue commands directly through the *admin_console*, which are stored in the storage component. This interface enables privileged operations such as maintenance tasks or direct access to internal services.

4.1 Constraints

To evaluate the conformity of the described online platform, we introduce a set of constraints that formalise the relevant security, privacy, and compliance requirements. Each constraint specifies a condition that must hold for all valid executions of the modelled system.

The constraints defined for this example encompass operational security, access control, logging behaviour, and legal compliance, which reflect common security and privacy requirements for systems subject to the General Data Protection Regulation (GDPR) [51]. In total, nine constraints are considered:

C1. Internal processing nodes must perform logging to enable the identification of attackers and system errors.

C2. Logged data must be sanitised to ensure conformity with the GDPR.

C3. Locally collected logs must be transmitted to a central logging server to enable centralised analysis.

C4. Data sent to any internal resource from an entry point must be authorised.

C5. Internal data exchanges must take place over authenticated channels.

C6. Personal data must not be transmitted to logging servers.

C7. Personal data must not be stored on servers located outside the EU.

C8. Administrative information must not be stored on any persistent storage server.

C9. Internal administrative data must be securely encrypted before transmission.

To enable automated conformity checking, each constraint is subsequently encoded using the DSL described in Subsection 2.3.2. The formal representations below correspond directly to the natural-language constraints listed above. In two cases, the current expressiveness of the DSL does not suffice; these limitations are explicitly indicated, and the intended encoding is provided for reference:

C1. *data neverFlows vertex Stereotype.Internal, !Stereotype.local_logging*

C2. *data neverFlows vertex Stereotype.local_logging, !Stereotype.log_sanitisation*

C3. *data Stereotype.local_logging* `hasToFlowTo` *vertex Logging.Server*[1]

C4. *data Stereotype.entrypoint neverFlows vertex Stereotype.Internal* `withoutFlowingTo` *Stereotype.auth_server* `before`[1]

C5. *data Stereotype.authenticated_request neverFlows vertex Stereotype.Internal*

C6. *data Data.personal neverFlows vertex Logging.Server*

C7. *data Data.personal neverFlows vertex Location.nonEU*

C8. *data User.admin neverFlows vertex Stereotype.Storage*

C9. *data User.admin, Encryption.encrypted neverFlows vertex Stereotype.Storage*

4.2 Violations

Applying the above constraints to the running example yields the following violations.

- **C1:** The first constraint is violated by three internal nodes that lack local logging: V1.1 *configuration_server*, V1.2 *admin_console*, and V1.3 *user_service*.
- **C2:** The second constraint is not directly violated in the original model, since no node is yet configured for local logging. However, C2 requires that any node implementing logging also perform log sanitisation. Consequently, once logging is introduced to mitigate C1, additional violations occur at V2.1 *configuration_server*, V2.2 *admin_console*, and V2.3 *user_service*.
- **C3:** Constraint C3 also depends on C1, as only nodes that perform logging can forward logs to a central logging server. Therefore, the TFGs associated with the nodes affected by C1 additionally violate C3, resulting in violations V3.1 *configuration_server*, V3.2 *admin_console*, and V3.3 *user_service*.
- **C4:** Since the *User* entity is the only entry point of the system, all data entering the system flows towards the *user_service*. As this data is not authorised before reaching the internal service, there is a single violation: V4.1 *user_service*.
- **C5:** Three internal nodes receive unauthenticated internal data: V5.1 *configuration_server*, V5.2 *admin_console*, and V5.3 *user_service*.
- **C6:** In the original DFD, no logging server is present; thus, C6 is initially not violated. However, when a logging server is introduced to mitigate C3, personal data is forwarded from the *user_service* to the logging server, yielding violation V6.1.

[1]These constraints cannot be expressed in the current DSL. The formulas above represent the intended semantics.

- **C7:** In two TFGs, personal data is transmitted to the storage component, which is located outside the EU. This leads to violation V7.1 at *Storage*.
- **C8:** The *admin_console* currently forwards administrative data to the storage component, thereby violating C8 (V8.1).
- **C9:** The admin sends unencrypted data to both the *configuration_server* (V9.1) and the *admin_console* (V9.2), violating C9.

4.3 Repaired Model

Figure 4.2 shows the repaired example model, in which all identified violations have been mitigated. To achieve full compliance with the constraints, each of the violations discussed above must be addressed.

To mitigate V1.1–V1.3 and the resulting violations V2.1–V2.3, all affected internal nodes (*configuration_server*, *admin_console*, and *user_service*) are extended with local logging and log sanitisation capabilities. A dedicated *logging_server* is introduced as a central sink, and log flows from the *user_service*, *admin_console*, and *configuration_server* are added to mitigate V3.1–V3.3. Constraint C4 is addressed by introducing an *auth_server* that authorises all user data before it is forwarded to the internal *user_service*, thereby resolving V4.1.

Data sent from the *User* to the *auth_server*, as well as data sent from the *Admin* to the *configuration_server* and *admin_console*, is transmitted over authenticated channels to mitigate V5.1–V5.3. When adding the flows from the *user_service* to the *logging_server*, all personal data is removed or anonymised to ensure compliance with Constraint C6. To resolve V7.1, the storage component is relocated from a non-EU to an EU data centre location.

In the repaired model, the flow from the *admin_console* to the storage component is removed, as it violated Constraint C8 and was not required for the intended functionality. Finally, all outgoing flows originating from the *Admin* entity are encrypted, thereby mitigating violations V9.1 and V9.2 with respect to Constraint C9.

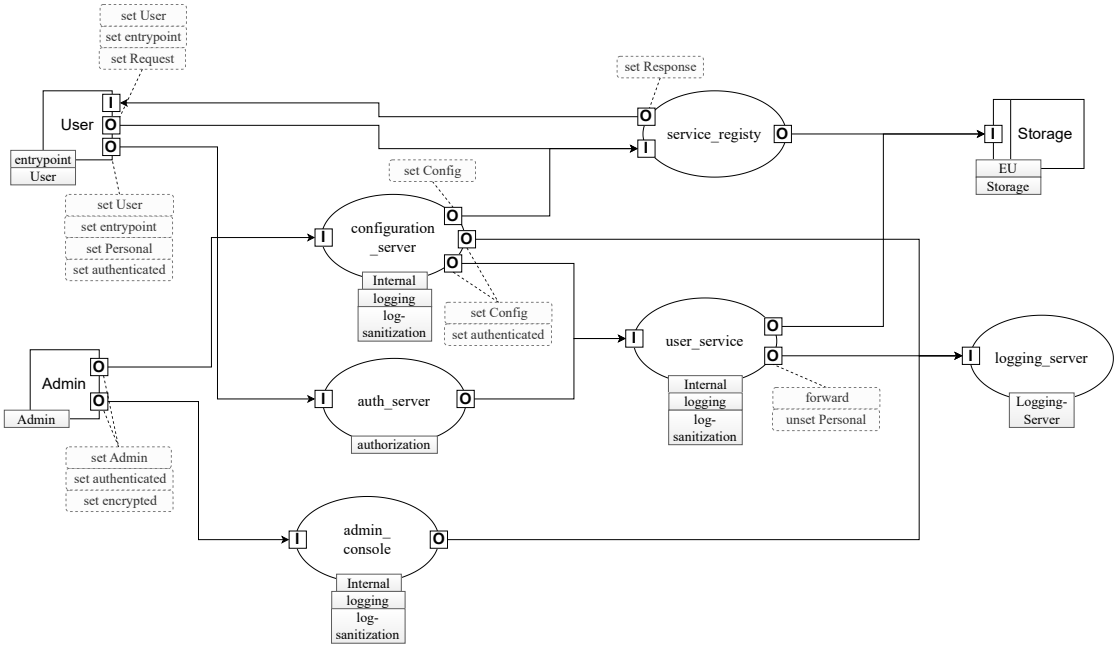

Figure 4.2: Repaired Example Model

5 Survey

The aim of this survey is to determine which discrete optimisation method is most suitable for automatic confidentiality mitigation at design time. This study employs the DFD of a simple online shop, defined in Chapter 4, to assess the capabilities of different discrete optimisation methods. Based on this example model, we demonstrate how the different methods approach mitigation through their distinct problem formulations. Specifically, we compare B&B, ILP, and Evolutionary algorithms, evaluating each on their optimality, performance, extensibility, and reproducibility. These methods were selected as representative approaches for discrete optimisation. We exclude any suboptimal methods, like heuristic approaches, since any suboptimal solution could lead to security risks or unnecessary costs. Constraint Programming has also been dismissed, despite its apparent suitability for handling (confidentiality) constraints in a specific solution space [38], namely software architecture. But upon closer examination, it became evident that in our case, the constraints do not restrict the solution space but rather define its structure. Consequently, Constraint Programming is less appropriate for our task, as it excels in problems with tightly constrained search spaces, whereas our problem exhibits few, if any, hard restrictions. Further exact methods were not considered, as they are highly specialised for problem classes unrelated to our specific domain.

5.1 Selection Criteria

To objectively compare the discrete optimisation methods, four selection criteria are defined: Optimality, Performance, Extensibility, and Reproducibility. Each criterion captures a different dimension of the method's effectiveness, efficiency, or practicality. These criteria were chosen to reflect both theoretical properties and practical implementation considerations relevant to design-time confidentiality mitigation.

5.1.1 Optimality

Optimality is defined as the property of an optimisation method to guarantee the best possible solution within a defined search space [37]. In the context of this thesis, this translates to finding the minimal change set that mitigates all confidentiality violations without introducing superfluous changes.

In discrete optimisation, we distinguish between methods that provide provable optimality guarantees and those that seek high-quality solutions without such guarantees [26, 59].

Exact methods systematically explore the solution space and prove that no better solution exists within the feasible region, while heuristic and metaheuristic approaches may converge to near-optimal solutions without providing such proofs [11].

For confidentiality mitigation, optimality ensures that the implemented solution introduces no unnecessary architectural complexity, implementation cost, or performance overhead, each of which could negatively impact system quality or maintainability.

5.1.2 Performance

Performance evaluates the computational efficiency and scalability of an optimisation approach. Specifically, it describes how rapidly a method identifies solutions and how its computational requirements grow with increasing model complexity, number of violations, and size of the mitigation space [29].

The performance of discrete optimisation methods varies widely depending on problem structure and algorithmic sophistication. Some methods exhibit exponential worst-case complexity but may perform well on structured instances with effective pruning strategies [26]. Others employ advanced preprocessing, relaxation techniques, or stochastic search to achieve practical efficiency [59, 11].

For design-time mitigation, performance determines whether the approach can provide timely feedback to architects and support interactive design exploration, particularly for large-scale systems with complex confidentiality requirements.

5.1.3 Extensibility

Extensibility describes the capacity of an optimisation approach to accommodate novel or evolving requirements with minimal modification to its core algorithmic structure. In the context of confidentiality mitigation, this includes the ability to integrate new types of confidentiality constraints, additional mitigation strategies, or alternative cost models as the security landscape evolves.

The extensibility of a method fundamentally depends on its level of abstraction and the separation between problem specification and solution algorithm. Methods that operate through declarative problem formulations tend to exhibit higher extensibility, as new requirements can be expressed through additional constraints or variables without modifying the solver [59]. In contrast, methods that encode problem-specific logic directly into the search algorithm may require more substantial modifications [26].

For tool implementation, extensibility determines the approach's maintainability and the effort required to adapt the system to emerging security requirements.

5.1.4 Reproducibility

Reproducibility refers to the consistency of results produced by an optimisation approach when executed multiple times under identical conditions. A method exhibits perfect reproducibility if it produces identical solutions across all runs; partial reproducibility if solutions differ but maintain equivalent quality; and poor reproducibility if substantial variation occurs.

Deterministic methods inherently provide perfect reproducibility, as their execution paths are fully determined by the input [26, 59]. Stochastic methods introduce randomness through initialisation, operator selection, or search guidance, which can lead to variation across runs [11].

For security-critical applications, reproducibility is essential for verification, debugging, regulatory auditing, and establishing trust in automated design decisions. Non-reproducible behaviour complicates validation efforts and may reduce confidence in the tool's reliability.

5.2 Methods

This section describes how each optimisation method approaches the automatic mitigation of confidentiality violations in the online shop DFD example from Chapter 4. The foundational concepts of B&B, ILP, and Evolutionary algorithms are covered in Chapter 2. Here, we focus exclusively on their application to the mitigation problem, demonstrating how each method would process the constraint violations identified in the running example.

5.2.1 Branch and Bound

B&B approaches the mitigation problem by systematically exploring the space of possible DFD modifications [26]. The algorithm constructs a search tree in which each node represents a partial state of the DFD, with some violations mitigated and others remaining. The edges of the search tree represent the application of specific mitigation actions.

Application to the Online Shop Example: Consider the initial state of the online shop DFD with all constraint violations unresolved. The B&B algorithm begins at the root node representing this initial state and must decide which mitigation to apply first.

For the first violation ("Internal nodes require logging"), several mitigation options may exist: add a logging component to the configuration server, add logging to the service registry, add logging to the user service, add logging to the admin console, or add a centralised logging infrastructure that all nodes connect to. Each option represents a distinct branch from the root node.

Suppose the algorithm explores the branch "add centralised logging server." This single action may simultaneously address multiple violations: it provides logging capability and establishes infrastructure for central log transmission. The new node in the search tree represents the DFD state with a logging server added, and two violations are marked as resolved. Seven violations remain.

From this node, the algorithm branches again. To address a second violation ("Logging requires log sanitisation), it might explore adding a sanitisation component between each internal node and the logging server.

At each node, B&B computes a *lower bound* on the total number of mitigations required from that partial state to full compliance [26]. A simple lower bound would be: current mitigations applied + number of unresolved violations (assuming optimistically that each remaining violation requires exactly one mitigation). If the algorithm has already found a complete solution with 6 mitigations, and the current node has applied 4 mitigations with 5 violations remaining, the lower bound is $4 + 5 = 9$. Since $9 > 6$, this branch is *pruned*; no further exploration occurs, as this path cannot improve upon the incumbent solution.

The search continues, systematically exploring promising branches and pruning unpromising ones, until all branches are either explored or pruned. The algorithm then reports the minimal mitigation set that resolves all violations.

Observations: The efficiency of B&B critically depends on the quality of the bounding function and the branching order [26]. Poor bounds lead to excessive exploration of suboptimal branches. Additionally, the algorithm operates directly on DFD structures: each branching decision modifies the architectural model by adding or altering components. This tight coupling between the optimisation logic and the domain model means that introducing new types of mitigations may require extending the branching rules to understand how to generate and evaluate these modifications.

5.2.2 Integer Linear Programming

ILP formulates the mitigation problem as a mathematical optimisation model, abstracting away from the procedural details of search [59, 31]. The approach is declarative: the problem is specified through variables, an objective function, and constraints; a general-purpose solver then finds the optimal solution.

Application to the Online Shop Example: We begin by enumerating all possible mitigation actions applicable to the online shop DFD. For illustration, consider a subset of potential mitigations:

- m_1: Add centralized logging server
- m_2: Add log sanitization component
- m_3: Add authorization at User entrypoint

- m_4: Add authorization at service registry
- m_5: Add authentication for internal data flows
- m_6: Encrypt admin data flows
- m_8: Deploy storage on EU-compliant server
- ... (additional mitigations as needed)

Each mitigation m_i is represented by a binary decision variable $x_i \in \{0, 1\}$, where $x_i = 1$ means the mitigation is applied. The violations from the running example are enumerated as $V = \{v_1, v_2, \ldots, v_9 9\}$.

The objective function minimises the total number of applied mitigations (or their cost):

$$\text{minimize} \quad \sum_{i=1}^{n} c_i x_i$$

where c_i represents the cost of mitigation m_i. For simplicity, we set $c_i = 1$ to minimise cardinality.

For each violation v_j, we specify which mitigations can resolve it. For example:

- Violation v_1 (internal nodes require logging): resolved by m_1 (centralized logging)
- Violation v_2 (log sanitization required): resolved by m_2 (add sanitization)
- Violation v_4 (entrypoint authorization): resolved by m_3 OR m_4
- Violation v_9 (admin data encryption): resolved by m_6

These relationships are encoded as constraints:

$$x_1 \geq 1 \quad (\text{for } v_1)$$

$$x_2 \geq 1 \quad (\text{for } v_2)$$

$$x_3 + x_4 \geq 1 \quad (\text{for } v_4)$$

$$x_6 \geq 1 \quad (\text{for } v_9)$$

Additional constraints may model dependencies (e.g., sanitization m_2 requires logging m_1 to be present: $x_2 \leq x_1$) or mutual exclusions.

Once the model is formulated, it is solved using a state-of-the-art ILP solver. The solver employs sophisticated techniques like: preprocessing to simplify the model, cutting planes to tighten the relaxation, and branch-and-cut to explore the solution space efficiently [59]. The output is a vector of binary values $(x_1, x_2, \ldots, x_n)$ indicating precisely which mitigations to apply to achieve compliance with minimal cost.

Observations: The ILP formulation operates on the *solution space* of mitigations rather than the *problem space* of DFD structures. The model does not procedurally construct or modify the DFD; instead, it selects an optimal subset of mitigations from a pre-enumerated set. This abstraction provides significant advantages: adding a new mitigation type (e.g., homomorphic encryption for computation on encrypted data) requires only defining a new variable x_{new} and specifying which violations it resolves; no changes to the solver or optimisation logic are necessary. Furthermore, the mathematical formulation is conceptually equivalent to a *minimal set covering problem* [59], a well-studied problem class for which highly optimised solvers exist.

5.2.3 Evolutionary Algorithms

Evolutionary algorithms employ a population-based stochastic search inspired by natural selection [11]. The approach iteratively evolves a population of candidate solutions through selection, recombination, and mutation, gradually improving solution quality over generations.

Application to the Online Shop Example: Each individual in the population represents a candidate mitigation plan for the online shop DFD. This is encoded as a binary chromosome of length n (the number of possible mitigations), where gene i has value 1 if mitigation m_i is included and 0 otherwise.

For instance, an individual chromosome might be:

$$[1, 1, 0, 1, 1, 0, 1, 1, 0, \ldots]$$

indicating that mitigations $m_1, m_2, m_4, m_5, m_7, m_8$ are applied, while m_3, m_6, m_9 are not.

The algorithm begins by initialising a population of random individuals. Some individuals may be *infeasible*; they fail to resolve all nine violations. For example, an individual that omits mitigation m_1 (centralised logging) would leave violation v_1 unresolved. Other individuals may be *feasible but suboptimal*, applying more mitigations than necessary.

Each individual is evaluated using a *fitness function* that balances two objectives: maximising the number of violations resolved and minimising the total cost of mitigations [11]. A formulation could be:

$$\text{fitness} = \alpha \cdot (\text{violations resolved}) - \beta \cdot (\text{total mitigation cost})$$

where α and β are weighting parameters. Individuals that resolve all violations and use fewer mitigations receive higher fitness scores.

The population evolves through genetic operators:

- **Selection**: Individuals with higher fitness are more likely to be chosen as parents for the next generation.

- **Crossover**: Two parent individuals exchange segments of their chromosomes, producing offspring that inherit traits from both. For example, parents $[1, 1, 0, 1, 0]$ and $[0, 1, 1, 0, 1]$ might produce offspring $[1, 1, 1, 0, 1]$.
- **Mutation**: Random genes are flipped (0 to 1 or 1 to 0) to introduce diversity and prevent premature convergence.

A critical challenge is *constraint handling*: ensuring that individuals satisfy feasibility, in our case, resolving all violations [11]. Two common approaches are:

1. **Penalty functions**: Infeasible individuals receive a fitness penalty proportional to the number of unresolved violations, guiding the population toward feasible regions.
2. **Repair operators**: After crossover or mutation, if an offspring is infeasible, a repair heuristic adds missing mitigations to satisfy all constraints. For example, if violation v_1 is unresolved, the repair operator sets $x_1 = 1$ to include mitigation m_1.

After a fixed number of generations (e.g., 100-500) or upon convergence (no improvement in best fitness for several generations), the algorithm terminates. The best individual in the final population represents the recommended mitigation plan.

Observations: Evolutionary algorithms can rapidly explore large solution spaces and often converge quickly to high-quality solutions [11]. However, they do not provide optimality guarantees: the best solution found may not be minimal. For the online shop example, one run might identify a 6-mitigation solution, while another run with different random initialisation might find a 5-mitigation solution or might converge to a different 6-mitigation solution. This stochastic variability means that results are not perfectly reproducible across runs, though multiple independent runs can be performed and the best result selected. Like ILP, evolutionary algorithms operate on the solution space of mitigations: the chromosome encodes which mitigations are active, not the procedural steps to construct the DFD. Extending the approach to new mitigation types requires only extending the chromosome length and updating the fitness evaluation logic with no changes to the genetic operators themselves.

5.3 Results

Table 5.1 presents an evaluation of the three discrete optimisation methods across the four selection criteria. The following subsections interpret these results in light of the approaches described in Section 5.2 and the theoretical properties discussed in Section 5.1.

Criteria\Method	Branch and Bound	Integer Linear Programming	Evolutionary
Optimality	+	+	?/−
Performance	○	+	++
Extensibility	+	++	++
Deterministic	+	+	−

Table 5.1: Evaluation of discrete optimisation methods for confidentiality mitigation.

5.3.1 Optimality

B&B and ILP both provide provable optimality guarantees for the mitigation problem. As demonstrated in their application to the online shop example, both methods systematically explore the solution space and prove that no better solution exists. B&B through explicit tree search with pruning, ILP through mathematical programming with branch-and-cut. For the online shop scenario, both methods guarantee identification of the minimal mitigation set.

Evolutionary algorithms, by contrast, do not provide such guarantees. As illustrated in the Evolutionary method description, the stochastic nature of population initialisation, crossover, and mutation means that convergence to the global optimum is not assured. However, evolutionary methods can ensure *feasibility* through constraint handling mechanisms such as penalty functions or repair operators. The rating of "?/–" reflects this distinction: while feasibility can be reliably achieved, minimality cannot be proven.

5.3.2 Performance

The performance characteristics of the three methods vary, as reflected in their problem formulations and search strategies.

Evolutionary algorithms receive the highest rating (++) due to their rapid convergence behaviour. As described in their application, evolutionary methods can quickly identify high-quality feasible solutions through parallel exploration of the population, often converging within hundreds of generations. For large-scale DFD models with many violations and mitigations, this speed advantage can become particularly significant.

ILP receives a strong rating (+), indicating good performance for practical instances. Modern ILP solvers incorporate decades of algorithmic refinements such as preprocessing, cutting

planes, and primal-dual methods, that enable efficient solution of many real-world problems [59].

B&B receives a neutral rating (○) because its performance is highly sensitive to problem structure and the quality of the bounding function. As noted in the method description, weak lower bounds lead to excessive exploration of suboptimal branches, potentially resulting in exponential search. While B&B can perform well with carefully designed bounds, it generally lacks the optimisations built into modern ILP solvers.

5.3.3 Extensibility

ILP and Evolutionary algorithms both receive the highest rating (++) for extensibility, stemming from their operation on the solution space of mitigations.

As illustrated in the ILP method description, adding a new mitigation type (e.g., differential privacy for data anonymisation) requires only introducing a new binary variable and specifying which violations it resolves through additional constraints. The solver's optimisation logic remains entirely unchanged and continues to minimise cost subject to coverage constraints, regardless of mitigation semantics. This declarative separation between problem specification and solution algorithm is a fundamental architectural advantage.

Evolutionary algorithms exhibit similar extensibility. As described in their application, new mitigation types simply extend the chromosome representation (add a gene) and may require updates to the fitness evaluation function. Crucially, the genetic operators operate at an abstract level and require no modification.

B&B receives a good but lower rating (+) due to its operation on the problem space. As noted in the method description, B&B explores DFD modifications directly: each branch represents adding or altering specific architectural components. Introducing a fundamentally new type of mitigation (e.g., adding secure hardware enclaves as nodes) may require extending the branching logic to understand how to generate, evaluate, and bound these new structural changes. While the general tree-search framework is flexible, these domain-specific modifications introduce maintenance complexity.

5.3.4 Reproducibility

ILP and B&B are both deterministic methods that produce identical solutions across repeated executions with the same input and solver configuration [26, 59]. For the online shop example, running B&B or ILP ten times will yield the same minimal mitigation set in all ten runs. This determinism is essential for verification, debugging, and establishing trust in the automated design process.

Evolutionary algorithms, by contrast, exhibit stochastic behaviour due to random initialisation, crossover, and mutation. For the online shop example, ten independent runs might produce five different mitigation plans, all feasible but with varying costs or compositions.

This non-determinism complicates verification efforts: reproducing a specific result requires storing and restoring the exact random seed, and demonstrating correctness becomes more challenging when behaviour varies across executions.

5.3.5 Overall Assessment

The overall analysis reveals distinct trade-offs among the three methods. B&B provides optimality and reproducibility but exhibits moderate performance and limited extensibility due to its problem-space formulation. Evolutionary algorithms excel in computational speed and extensibility but sacrifice optimality guarantees and reproducibility. ILP uniquely combines four desirable properties: guaranteed optimality, strong performance through modern solver technology, maximal extensibility via declarative problem specification, and complete reproducibility through deterministic execution.

For the automatic confidentiality mitigation problem, where provable correctness is essential for security, extensibility is necessary for long-term tool evolution, and reproducibility is required for verification, ILP emerges as the most suitable approach.

6 Automated Mitigation Approach

This chapter presents the proposed approach and its rationale for implementation. It shows how confidentiality violations in arbitrary DFDs and their associated constraints are analysed and systematically mitigated by combining the DFA framework xDECAF of Boltz et al. [7] with its DSL for constraint specification. The analysis and all subsequent mitigation steps operate on this DSL-based representation of constraints and the corresponding DFD models.

The chapter first introduces the mitigation workflow, which starts from an initial violating DFD, derives constraint-specific mitigation strategies, and incrementally transforms the model into a repaired variant via a discrete optimisation formulation. It then details the supported strategy families, discusses extensibility for constraints that exceed the expressiveness of the DSL, and explains which mitigation steps are automated and which require manual specification. Finally, it outlines how the overall system integrates these components and how users interact with the tool to obtain and inspect repaired models.

6.1 Mitigation Workflow

This section details the sequential mitigation steps that form the core workflow of the proposed approach, as illustrated in Figure 6.1. It provides a systematic account of how a violating DFD and the set of constraints are transformed into a repaired model through iterative analysis and optimisation.

Each phase defines clear input and output artefacts and describes how its results are consumed by subsequent phases, making the internal structure of the system and key algorithmic choices explicit.

The mitigation process begins with constraint and strategy analysis, followed by DFD evaluation through DFA integration. Based on these analyses, the approach explores the corresponding problem space and formulates a discrete optimisation problem that captures all feasible mitigation options and their associated costs. Solving this problem with an ILP solver yields a cost-minimal mitigation strategy, whose actions are then applied to the original model to produce a repaired system representation.

Figure 6.1: Depiction of the Mitigation Workflow

6.1.1 Constraint Analysis

The Constraint Analysis phase derives the set of mitigation strategies that can be applied for a given constraint set and identifies dependencies between these constraints. Its input is the set of confidentiality constraints expressed in the DSL, and its output is a set of mitigation strategies with associated costs, together with relations that capture how constraints and strategies influence one another. This effectively defines the problem space, as it establishes the structured "language" in which the space of possible mitigation options is later explored and constructed. As the DSL already captures the majority of confidentiality constraints used in existing DFA-based analyses and is designed to be extensible, this work focuses automation efforts on DSL-expressible constraints. For constraints that fundamentally exceed the DSL's label-based expressiveness, extending full automation would require substantial additional modelling and implementation effort, which is not justified within the scope of this thesis. This phase is crucial because all subsequent steps in the mitigation workflow operate exclusively on the strategies and relations computed here or provided by the user.

A first task of this phase is the automatic derivation of label-based mitigation strategies from the constraints specified in the DSL. For constraints that prohibit/enforce certain combinations of labels (for example, that data labelled *internal* and not labelled *localLogging* may not be processed by a particular component), the analysis systematically generates label-addition strategies from negative literals. Concretely, for each negative literal in such a constraint, the phase introduces a mitigation strategy that adds the corresponding label, typically with a low cost, because adding missing labels is considered a minimal, non-disruptive change.

Handling label-deletion strategies requires an additional decomposition of constraints. Many confidentiality constraints in the DSL follow a pattern where one part identifies data and another part identifies the nodes or locations to which this data may or may not flow (for example, "personalData neverFlows nonEU"). In this case, the data part (here: *personalData*) is interpreted as describing core information that should be preserved, whereas the node part (here: *nonEU*) characterises where this data may appear. Consequently, the analysis phase only derives deletion strategies for the node-related part of such constraints: each positive literal in the node part gives rise to a strategy that removes the corresponding node label or property. These strategies are assigned a comparatively high cost to express the preference for preserving system functionality and structure, and to make label addition the default choice whenever possible.

Beyond deriving individual strategies, the Constraint Analysis phase also captures interactions between constraints. In some situations, satisfying one constraint implicitly requires satisfying another. For instance, a constraint enforcing that *internal* data must be subject to *localLogging*, and a second constraint requiring that all *localLogging* activity must be accompanied by *logSanitisation*, yields a dependency: any strategy that enforces the first constraint must also ensure that the second constraint is not violated. To represent such relationships, the phase records "requires" relations between strategies that must be jointly selected, as well as "conflicts" where applying a strategy that satisfies one constraint would violate another. These relations are later translated into logical clauses in the optimisation problem to prevent incompatible combinations and to enforce necessary co-occurrences of strategies.

The DSL currently supports constraints that can be expressed in terms of labels on data and architectural elements, which enables the automated derivation of label-addition and label-deletion strategies. In contrast, strategies that modify the structure of the DFD itself, such as adding or removing nodes and flows, cannot be expressed adequately within the DSL. As a result, the Constraint Analysis phase treats such structural strategies as external inputs: they must be specified manually by users or tool developers and are incorporated into the set of available strategies without automatic generation. This separation ensures that the automated analysis remains generic and scalable for all constraints expressible in the DSL, while still allowing project- or domain-specific structural strategies to be integrated on demand. Together, the automatically derived label-based strategies, the manually specified structural strategies, and the recorded constraint interactions form the basis for the subsequent exploration of the mitigation problem space and its encoding as a discrete optimisation problem.

Algorithm 1 consolidates these steps: lines 4–9 derive label-addition strategies from negative literals, lines 10–15 derive label-deletion strategies from the node part of each constraint, lines 17–24 record requires and conflict relations.

6.1.2 xDECAF

This phase identifies all concrete locations in the DFD where mitigations are required, using the xDECAF DFA-framework. For this purpose, it processes the results of the confidentiality analysis and collects every reported violation, each of which is associated with a specific node in the DFD. For every such violation, the phase creates a dedicated node object that bundles the affected node, the violated constraint, and the corresponding TFG.

By focusing only on the violating nodes and their violated constraint, the subsequent exploration operates on a reduced, violation-centred subset of the model instead of the full DFD. This reduction is crucial for scalability in realistic systems, because it prevents the combinatorial growth that would result from considering all strategies for every node, regardless of whether it participates in a violation. Representing violations in this way enables later phases to reason locally about mitigation options without repeatedly querying the global model.

Algorithm 1 Constraint Analysis

Require: Set of constraints C, set of user-defined strategies $\mathcal{S}_{\text{custom}}$
Ensure: Set of strategies $\mathcal{S}$, required relations $\mathcal{R}$, contradiction relations $\mathcal{K}$

$\mathcal{S} \leftarrow \mathcal{S}_{\text{custom}}$
$\mathcal{R} \leftarrow \emptyset$
$\mathcal{K} \leftarrow \emptyset$
for each constraint $c \in C$ **do**
    if c is expressible in the DSL **then**
        for each negative literal l^- in c **do**
            $s \leftarrow$ CreateLabelAdditionStrategy(l^-, cost = 1)
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
        end for
        $(d_{\text{data}}, d_{\text{node}}) \leftarrow$ DecomposeConstraint(c)
        for each positive literal l^+ in d_{node} **do**
            $s \leftarrow$ CreateLabelDeletionStrategy(l^+, cost = high)
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
        end for
    end if
end for
for each pair $(c_i, c_j) \in C \times C,\ c_i \neq c_j$ **do**
    if SatisfyingOneRequiresOther(c_i, c_j) **then**
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{(c_i \rightarrow c_j)\}$
    end if
    if StrategiesContradict(c_i, c_j) **then**
        $\mathcal{K} \leftarrow \mathcal{K} \cup \{(s_i, s_j)\}$
    end if
end for
return $\mathcal{S}, \mathcal{R}, \mathcal{K}$

6.1.3 Problem Space Exploration

The Problem Space Exploration phase systematically enumerates all mitigation options that can resolve the detected violations in the DFD. Its inputs are the violating node objects produced in the DFD Analysis phase and the set of mitigation strategies derived during Constraint Analysis. For each violating node-constraint pair, this phase determines all concrete ways in which the available strategies can be applied, thereby constructing the space of all candidate mitigations that can be derived from the available strategies for the observed violations. In the following, a *mitigation strategy* denotes an abstract description of a change type, whereas a *mitigation* denotes a concrete instantiation of such a strategy for a specific violation context.

For every violating node, the phase first considers all strategies that are applicable to the corresponding violated constraint. Each such strategy yields at least one possible mitigation, but some strategies can be instantiated in multiple ways along the TFG of the node. For

example, a label-addition strategy may be applied at different nodes or edges on the path that contributes to a violation, resulting in several distinct mitigation options that all satisfy the same constraint. Similarly, constraints that admit multiple strategy types, such as adding a repairing label or removing a violating label, give rise to alternative mitigations, each with its own cost and structural impact on the model.

In addition, this phase resolves all required mitigation relations identified during Constraint Analysis. When a strategy is marked as requiring other strategies, the exploration phase systematically expands the corresponding mitigation option to include all required strategies. If required, strategies themselves can be applied in multiple ways; the phase accounts for all valid combinations, ensuring that the resulting composite mitigations are closed under these dependencies. The outcome is a set of fully specified mitigation candidates that respect both local applicability at violating nodes and the global dependency relations between strategies, providing the foundation for the subsequent formulation of the discrete optimisation problem.

Besides dependencies, the Problem Space Exploration phase also detects and records contradictions between mitigation options. Contradictions arise in two principal forms. The first type is introduced by constraints that explicitly forbid certain label combinations on a flow or node. If two strategies together would add labels that a constraint declares incompatible, their joint application is marked as contradictory and thus excluded from the joint application of valid mitigation candidates. The second type of contradiction stems from the structure of the DFD itself: for example, one strategy may delete a node or flow while another strategy intends to add or remove labels on the same element. In such cases, applying both strategies would either operate on elements that no longer exist or leave violations unaddressed at removed locations; therefore, the corresponding combinations are identified as invalid. By explicitly encoding these contradictions, the phase ensures that subsequent optimisation does not select strategy combinations that are known to be incompatible with respect to the given constraints and modelled structure.

Algorithm 2 formalises the procedure described above: lines 3–13 iterate over all violating node objects, instantiate applicable strategies at each position along the TFG, and expand each mitigation to include all required co-mitigations derived from the constraint relations; lines 14–18 then perform a pairwise scan over all candidate mitigations to detect and record both constraint-based and structural contradictions.

6.1.4 ILP Problem Formulation and Solving

The ILP Problem Formulation and Solving phase translates the enumerated mitigation options and their logical relationships into a discrete optimisation problem that can be solved by an ILP solver. The inputs to this phase are three lists: the set of all unique mitigations generated during Problem Space Exploration, the set of all violations with their corresponding candidate mitigations, and the set of contradictions between mitigations. The outputs are a cost-minimal selection of mitigations that collectively resolve all violations while respecting all dependency and contradiction relations.

Algorithm 2 Problem Space Exploration

Require: Violating node objects $\mathcal{N}_v$, strategies $\mathcal{S}$, required relations $\mathcal{R}$, contradiction relations $\mathcal{K}$
Ensure: Candidate mitigations $\mathcal{M}$, active contradictions $\mathcal{K}'$

$\mathcal{M} \leftarrow \emptyset$
$\mathcal{K}' \leftarrow \emptyset$
for each violating node object $(n, c, tfg) \in \mathcal{N}_v$ **do**
  $\mathcal{S}_c \leftarrow \{s \in \mathcal{S} \mid s \text{ is applicable to } c\}$
  for each strategy $s \in \mathcal{S}_c$ **do**
    for each applicable position p in tfg **do**
      $m \leftarrow$ InstantiateMitigation(s, n, p)
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
      if $(c \rightarrow c') \in \mathcal{R}$ for some c' **then**
        $\mathcal{S}_{c'} \leftarrow \{s' \in \mathcal{S} \mid s' \text{ is applicable to } c'\}$
        for each $s' \in \mathcal{S}_{c'}$, each position p' in tfg **do**
          $m' \leftarrow$ InstantiateMitigation(s', n, p')
          AddRequiredMitigation(m, m')
          $\mathcal{M} \leftarrow \mathcal{M} \cup \{m'\}$
        end for
      end if
    end for
  end for
end for
for each pair $(m_a, m_b) \in \mathcal{M} \times \mathcal{M},\ m_a \neq m_b$ **do**
  if IsConstraintContradiction$(m_a, m_b, \mathcal{K})$ **or** IsStructuralContradiction(m_a, m_b) **then**
    $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{(m_a, m_b)\}$
  end if
end for
return $\mathcal{M}, \mathcal{K}'$

Problem Encoding The encoding process begins by assigning a unique Boolean ILP variable to each unique mitigation in the problem space. If two violations can be resolved by the same mitigation, both violations reference a single shared variable, ensuring that selecting one mitigation for multiple purposes incurs its cost exactly once. Formally, let $M = \{m_1, m_2, \ldots, m_n\}$ be the set of all unique mitigations and let $x_i \in \{0, 1\}$ be the Boolean decision variable associated with mitigation m_i. The value $x_i = 1$ indicates that mitigation m_i is selected as part of the final repair strategy, whereas $x_i = 0$ indicates that it is not applied.

With these variables established, the problem formulation must encode four classes of constraints that ensure the correctness and completeness of the mitigation strategy: coverage constraints, contradiction constraints, required-mitigation constraints, and the minimisation objective.

Coverage Constraints Coverage constraints guarantee that every violation in the DFD is resolved by at least one selected mitigation. Let $V = \{v_1, v_2, \ldots, v_k\}$ denote the set of violations, and for each violation v_j, let $M_j \subseteq M$ be the subset of mitigations that can resolve it. In first-order logic, the requirement that violation v_j is covered can be expressed as a disjunction over the corresponding mitigation variables:

$$x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_p}$$

where $\{m_{i_1}, m_{i_2}, \ldots, m_{i_p}\} = M_j$. In the ILP formulation, this logical disjunction is encoded as the linear inequality:

$$x_{i_1} + x_{i_2} + \cdots + x_{i_p} \geq 1$$

This constraint ensures that at least one mitigation variable in the set M_j must be set to 1. One such constraint is generated for each violation $v_j \in V$.

Contradiction Constraints Contradiction constraints prevent the solver from selecting incompatible combinations of mitigations. As discussed in the Problem Space Exploration phase, contradictions arise either from explicit constraint incompatibilities (e.g., two strategies that add labels forbidden to co-occur by a constraint) or from structural conflicts (e.g., one strategy deletes a node while another attempts to modify a label on that node). Let (m_a, m_b) denote a pair of contradictory mitigations. In first-order logic, the requirement that m_a and m_b cannot both be selected is expressed as the negation of their conjunction:

$$\neg(x_a \wedge x_b)$$

which is logically equivalent to:

$$\neg x_a \vee \neg x_b$$

In the ILP encoding, this is represented by the linear inequality:

$$x_a + x_b \leq 1$$

This constraint allows at most one of x_a or x_b to be set to 1, thereby preventing the joint selection of contradictory mitigations. One such constraint is generated for every contradictory pair identified during the exploration phase.

Required-Mitigation Constraints Required-mitigation constraints model situations where selecting a particular mitigation necessitates the selection of one or more additional mitigations. These dependencies arise from constraint interactions identified during Constraint Analysis. For instance, if mitigation m_A enforces a constraint that implicitly requires another constraint to be satisfied, then selecting m_A must force the selection of at least one mitigation that satisfies the required constraint.

The logical structure of such a dependency can be expressed as an implication:

$$x_A \rightarrow (x_b \wedge x_c) \vee (x_d \wedge x_e) \vee (x_f \wedge x_g)$$

This reads as: "x_A is selected only if at least one of the conjunctions $(x_b \wedge x_c)$, $(x_d \wedge x_e)$, or $(x_f \wedge x_g)$ holds." Each conjunction represents a set of mitigations that together satisfy the required constraint, and the disjunction expresses the fact that any one of these sets suffices.

To encode this in ILP, auxiliary Boolean variables are introduced for each conjunction. Let Y_1, Y_2, Y_3 be auxiliary variables corresponding to the three conjunctions. The bi-implication is then decomposed into a set of linear inequalities:

1. **Linking x_A to the disjunction of auxiliary variables:**

   $$-x_A + Y_1 + Y_2 + Y_3 \geq 0$$

   This ensures that if $x_A = 1$, then at least one of Y_1, Y_2, Y_3 must be 1.

2. **Enforcing each auxiliary variable to imply its corresponding conjunction:**

   For each auxiliary variable Y_i and its associated child variables (e.g., Y_1 corresponds to $x_b \wedge x_c$), two types of constraints are added:

   a) Each child must be selected if the auxiliary is selected:

   $$-Y_1 + x_b \geq 0$$
   $$-Y_1 + x_c \geq 0$$

   b) The auxiliary can only be selected if all its children are selected:

   $$-Y_1 + x_b + x_c - 2 < 0$$

The general form for an auxiliary variable Y_i with k child variables $x_{c_1}, \ldots, x_{c_k}$ is:

$$-Y_i + \sum_{j=1}^{k} x_{c_j} - k \leq 0$$

Together, these constraints ensure that $Y_i = 1$ if and only if all its children are selected, and that $x_A = 1$ implies at least one such conjunction is active. This encoding allows the ILP solver to choose the least costly set of required mitigations that satisfy the dependency.

Minimization Objective The objective function of the ILP minimises the total cost of the selected mitigations. Each mitigation m_i is assigned a cost c_i during the Problem Space Exploration phase, either automatically (based on the type of strategy) or manually (for user-defined strategies). Costs are chosen to reflect the preference for minimal, non-disruptive changes: label additions typically receive low costs, label deletions higher costs, and structural modifications the highest costs. Concretely, label-addition strategies are automatically assigned a cost of 1, reflecting that adding a missing label is considered the least disruptive change to the system model. Label-deletion strategies are automatically assigned a cost of Integer.MAX_VALUE, making them a near-last resort in the optimisation that is only selected when no label-addition resolves the violation. Structural strategies, such as adding or removing nodes and flows, cannot be derived automatically and must therefore be supplied manually by the user, together with an explicit cost that calibrates their relative intrusiveness within the specific deployment context. Under this scheme, a cost-minimal solution is the mitigation strategy that resolves all violations with the smallest possible aggregate cost. Importantly, the cost of a mitigation is independent of whether it appears as a primary or required mitigation; the objective accounts for all selected mitigations uniformly.

The objective function is defined as:

$$\text{minimize} \quad \sum_{i=1}^{n} c_i \cdot x_i$$

Solving and Extraction Once the ILP problem is fully encoded with Boolean variables, coverage, contradiction, and required-mitigation constraints, and the cost-minimisation objective, it is passed to an ILP solver. The solver explores the feasible region defined by the constraints and returns an optimal solution (if one exists) that assigns values to all decision variables x_i. The solution extraction step then retrieves all mitigations for which $x_i = 1$, producing the final set of actions to be applied to the DFD in the subsequent Applying Mitigations phase.

Concrete Example To illustrate the encoding, consider a simplified scenario derived from the running example (Chapter 4). Suppose two violations are detected:

- **Violation v_1:** Internal data at the `configuration_server` node lacks the `localLogging` label (Constraint C1).
- **Violation v_2:** Internal data at the `admin_console` node lacks authentication (violating Constraint C5).

Assume the Problem Space Exploration phase has identified the following mitigations:

- m_1: Add `localLogging` label to `configuration_server` (cost $c_1 = 1$).
- m_2: Add `localLogging` label to `admin_console` (cost $c_2 = 1$).
- m_3: Add `authenticated` label to `admin_console` (cost $c_3 = 1$).

In this scenario, Violation v_1 can be resolved either by logging the data at the `configuration_server` (mitigation m_1) or by logging it at the `admin_console` (mitigation m_2). Violation v_2 can be resolved by enforcing authentication at the `admin_console` (mitigation m_3).

Suppose further that Constraint Analysis determined that any node performing local logging must also satisfy a log sanitisation constraint (Constraint C2), which can be satisfied by:

- m_4: Add `logSanitization` label to `configuration_server` (cost $c_4 = 1$).
- m_5: Add `logSanitization` label to `admin_console` (cost $c_5 = 1$).

This yields two required-mitigation relations: selecting m_1 requires m_4, and selecting m_2 requires m_5.

Finally, assume that m_2 and m_3 are contradictory, for instance due to a hypothetical constraint forbidding a particular combination of logging and authentication behaviour at the `admin_console`. In other words, it is not permitted to both add `localLogging` and enforce `authenticated` on the same node in this configuration.

The ILP encoding proceeds as follows.

Variables:

$$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}$$

where $x_i = 1$ indicates that mitigation m_i is selected.

Coverage constraints:

For v_1 (resolved by m_1 or m_2):

$$x_1 + x_2 \geq 1$$

For v_2 (resolved by m_3):

$$x_3 \geq 1$$

Contradiction constraint:

For the contradictory pair (m_2, m_3):

$$x_2 + x_3 \leq 1$$

This constraint ensures that m_2 and m_3 cannot both be selected.

Required-mitigation constraints:

If m_1 is selected, then m_4 must be selected (single required mitigation):

$$-x_1 + x_4 \geq 0$$

If m_2 is selected, then m_5 must be selected:

$$-x_2 + x_5 \geq 0$$

Objective function:

$$\text{minimize} \quad x_1 + x_2 + x_3 + x_4 + x_5$$

Solving the example:

From the coverage constraint for v_2, any feasible solution must satisfy $x_3 = 1$. The contradiction constraint $x_2 + x_3 \leq 1$ then implies $x_2 = 0$. The coverage constraint for v_1 becomes

$$x_1 + x_2 \geq 1 \quad \Rightarrow \quad x_1 \geq 1,$$

so $x_1 = 1$. The required-mitigation constraint $-x_1 + x_4 \geq 0$ then forces $x_4 = 1$. Since $x_2 = 0$, there is no requirement to select m_5, and the cost-minimal choice is $x_5 = 0$.

Thus, the unique cost-minimal feasible solution is:

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 1, \quad x_4 = 1, \quad x_5 = 0,$$

with a total cost of 3. Intuitively, the solver selects mitigations m_1 and m_4 at the `configuration_server` to repair v_1 and satisfy the sanitisation requirement, and mitigation m_3 at the `admin_console` to repair v_2. The contradiction constraint prevents the solver from choosing the alternative combination that would log at the `admin_console` using m_2 together with m_3, even though m_2 would otherwise be a valid option to repair v_1.

6.1.5 Applying Mitigations

The Applying Mitigations phase takes the optimal mitigation strategy computed by the ILP solver and transforms it into concrete modifications of the DFD and its associated data dictionary. Its input is the set of selected mitigations returned by the solver, and its output is a repaired DFD model that conforms to all specified confidentiality constraints.

Deriving Actions from Mitigations Each selected mitigation encodes a high-level intent but requires translation into concrete, executable actions on the model. The phase begins by extracting the underlying mitigation strategy from each mitigation object. Each mitigation strategy specifies the operation type, the domain (the specific element in the DFD to which the action applies), and any associated labels. By collecting all mitigation strategies from the selected mitigations, the phase produces a flat list of atomic actions (instantiated strategy applications) that collectively constitute the complete repair strategy.

Ordered Application of Actions Not all actions can be applied in arbitrary order; structural dependencies between actions necessitate a specific execution sequence. The phase applies actions in the following order:

1. **Label addition and removal** on existing nodes and flows
2. **Node addition** (including creation of new processes)
3. **Sink addition** (creation of new storage or external entity nodes)
4. **Node removal** (deleting nodes and reconnecting their predecessors and successors)
5. **Flow removal** (deleting edges from the DFD)

This ordering reflects a general principle: *additive* modifications are applied before *subtractive* ones. This ensures that when a removal operation is executed, all dependencies have already been established, and the removal can be performed cleanly without leaving dangling references or breaking the overall connectivity of the model.

After all actions have been applied, the resulting DFD is guaranteed to satisfy the coverage and contradiction constraints enforced by the ILP formulation, because every violation has at least one resolving mitigation and no contradictory mitigation pair is selected by construction. The repaired model is then ready for re-analysis, and users can inspect the applied changes to understand how the system was modified to restore compliance with the confidentiality constraints.

6.2 Mitigation Strategy Specification

Mitigation strategies describe the concrete types of changes that the approach may apply to a DFD to address constraint violations. The following subsections introduce the four strategy families supported by the approach and explain their intended use, relative intrusiveness, and relation to the evaluation baselines, such as the SAT-based technique.

6.2.1 Label Addition

Label addition is the baseline strategy of the approach and represents the least invasive type of change to the system model. It modifies only the annotations of data or node elements, leaving the general structure of the DFD unchanged. This is the class of modifications that the SAT-based baseline can express, making label addition the common denominator for comparison between the approaches.

Conceptually, a label-addition strategy annotates one or more new labels to a node or to the outgoing data of a node. Because many confidentiality constraints are expressed over labels (for example, requiring that internal data be logged or authenticated), adding suitable labels can often repair violations without altering data flows or behaviour. The approach, therefore, prefers label addition wherever possible and assigns comparatively low costs to these strategies in the optimisation problem. This preference reflects the goal of minimising the effort required from users to adapt their systems, since purely annotational changes are typically easier to implement than structural refactorings.

A crucial aspect of label addition in this context is the role of forwarding. Many systems propagate data through multiple nodes without materially changing it or only adding new data. If a label can be added close to the source of a data flow, the label may be forwarded along the path and thereby repair multiple violations at once. From an optimisation perspective, such an early label addition may have a larger impact in terms of repaired violations and occurrences of the label while still resulting in a small change set. The strategy specification, therefore, allows the creation of label-addition strategies at different positions along the transpose flow graph, enabling the solver to trade off change vs. impact set when selecting minimal-cost repairs.

6.2.2 Label Deletion

Label-deletion strategies remove labels from nodes or data flows. Typical examples include constraints that prohibit personal data from being sent to a non-EU storage location, or that forbid admin information from being logged. In such cases, adding further labels cannot repair the violation; instead, the violating property must be removed, or the structure of the system must be changed.

The approach uses label deletion as a less invasive alternative to structural modifications such as node or flow removal. Deleting a label preserves the functional behaviour of the node or data flow while changing its security-relevant classification. However, label-deletion strategies are assigned higher costs than label addition to reflect their stronger impact on the interpretation of the system (for example, a node may lose its internal status). To avoid inadvertently deleting essential functionality, the strategy generation step distinguishes between labels that represent core data (such as *personalData*) and labels that represent locations or roles (such as *nonEU*). The former are treated as non-deletable, whereas the latter can be removed when necessary to satisfy constraints.

6.2.3 Node Insertion and Removal

Node-level strategies capture structural changes that add or remove processing or storage elements in the DFD. These strategies are more invasive than label changes but may be required when violations cannot be repaired by annotations alone. The approach distinguishes between adding nodes and deleting nodes.

When adding nodes, the strategy specification supports two roles. First, intermediate processing nodes can be inserted into existing flows to implement additional behaviour such as encryption or access control. Such a node typically has a behaviour that forwards input data and attaches or transforms labels (for example, adding *access control* label before data reaches an internal component). Second, nodes can be introduced as sinks on existing flows by splitting a flow and terminating one branch in a newly added node that collects data, e.g., for compliant logging or storage.

Node-deletion strategies remove nodes from the model. When a node is removed, the mitigation semantics rewire incoming and outgoing flows to bypass the node while preserving data forwarding as far as possible. Node-level strategies therefore provide powerful means to restore compliance when neither label additions nor deletions suffice.

6.2.4 Flow Removal

Flow-level strategies in the current approach are restricted to the *removal* of existing flows. The addition of new flows is intentionally not supported, because adding a flow cannot repair violations that occur on a parallel, independent TFG and would instead introduce new behaviour.

Flow-removal strategies eliminate paths that lead to constraint violations, such as flows from an internal admin console directly to a non-compliant storage component. Removing a flow prevents data from reaching a problematic destination while leaving the involved nodes themselves intact. Because cutting flows can change which components receive which data, flow-removal strategies are generally more intrusive than pure label modifications but can still be less intrusive than deleting entire nodes. In combination with node-level strategies, they provide the structural flexibility needed to repair violations that cannot be addressed with label operations alone.

6.3 System Architecture and Integration

This section describes the implementation architecture shown in Figure 6.2. The figure shows the core components of the mitigation workflow and its supporting classes.

The following subsections detail the responsibilities of each core component, describe how users interact with the system to provide models, specify constraints, and define mitigation strategies.

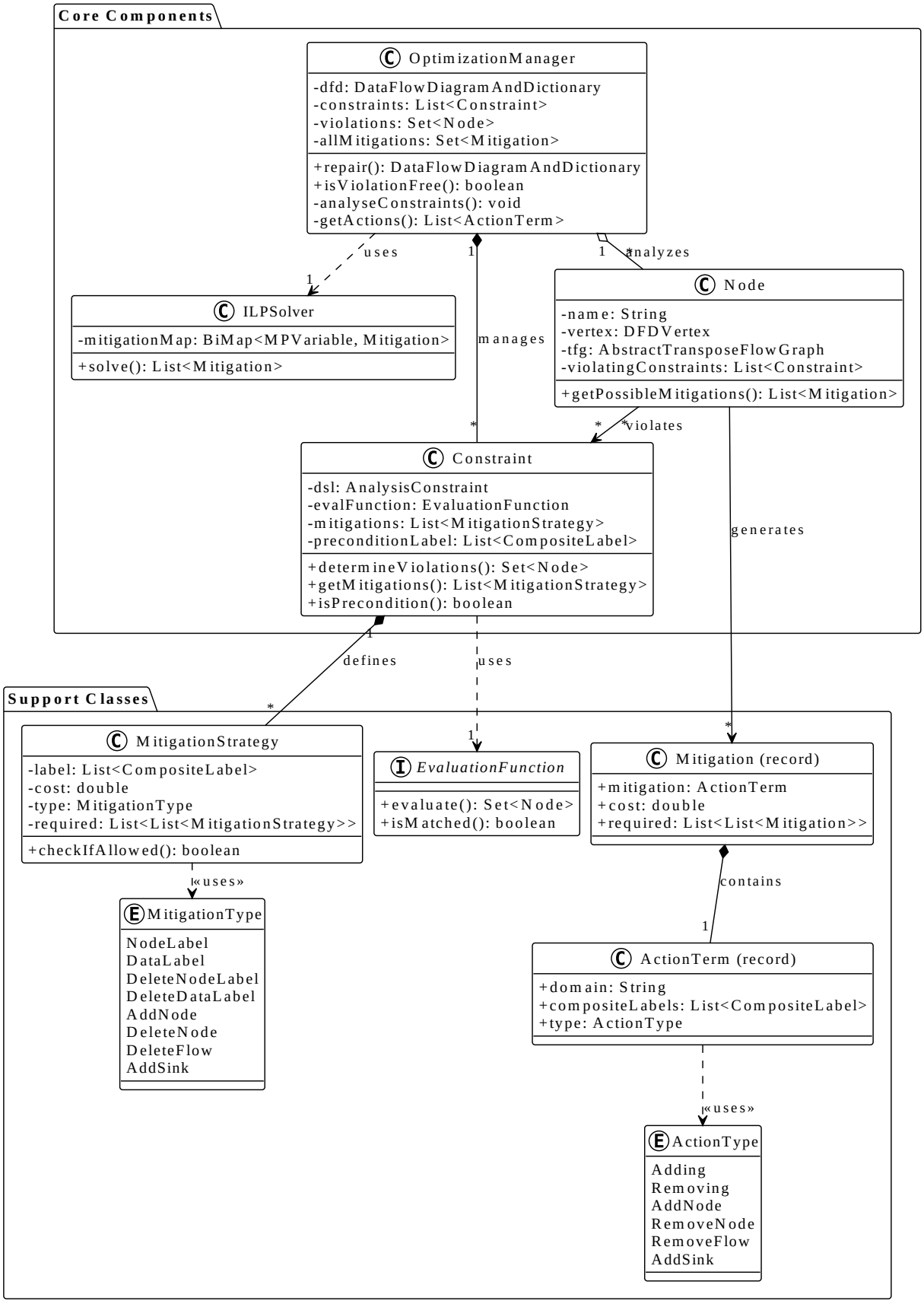

Figure 6.2: System Architecture

6.3.1 Components and Responsibilities

The implementation is structured around four main elements: the *Optimisation Manager*, the *Constraint* abstraction, the *Node* representation for violations, and the *ILP solver* interface. In contrast to the workflow description in Section 6.1, this section focuses on their concrete responsibilities and interactions in the code base.

Optimization Manager The Optimisation Manager coordinates the execution of all phases of the mitigation workflow. It keeps the global state of

- the loaded DFD model and associated data dictionary,
- the set of constraints and their violations,
- the pool of mitigation strategies and instantiated mitigations, and
- the final selection of mitigations and the repaired model.

It invokes the individual phases in the correct order (constraint analysis, DFD analysis, problem-space exploration, ILP formulation/solving, application of mitigations) and passes only the artefacts required for each step. This centralisation avoids duplicated computations and provides a single point where intermediate results can be inspected, logged, or exported.

Constraints The `Constraint` class uniformly represents both DSL-based and custom constraints. Each instance encapsulates

- its definition (either a DSL expression or an attached evaluation function),
- a reference to the mitigation strategies associated with this constraint (automatically derived for DSL constraints or manually attached for custom ones), and
- meta-information such as costs.

For constraints expressed in the DSL, the implementation can inspect the DSL term and derive label-based strategies and their costs automatically, as described in the Constraint Analysis phase. For constraints defined solely by an evaluation function, the `Constraint` object is used only to locate violations; all strategies for such constraints must be provided explicitly by the user or tool developer.

Node The `Node` class is a thin wrapper around a violating DFD vertex. It stores

- the underlying DFD vertex,
- the transpose flow graph (TFG) used in the confidentiality analysis, and
- the `Constraint` instance that is violated.

During problem-space exploration, each `Node` instance is responsible for exploring all mitigation strategies applicable to its constraint. The resulting mitigations carry concrete action terms (for example, “add label L on vertex v” or “remove flow f”).

ILP Solver The ILP solver component provides a narrow interface that hides the details of the underlying optimisation library. It accepts:

- the set of unique mitigations,
- the mapping from violations to their candidate mitigations, and
- the set of contradictions between mitigations,

and constructs the Boolean ILP instance accordingly. The solver component returns only the identifiers of the selected mitigations; it does not modify the model itself. This separation allows the optimisation backend to be replaced (e.g. by switching to another solver) without touching the rest of the architecture.

6.3.2 Component Integration

The data and control flow between components closely follows the conceptual phases, but with a clear separation between *analysis* and *execution* concerns.

- The Optimisation Manager first queries all `Constraint` instances to perform the initial confidentiality analysis using the underlying DFA framework. Each reported violation is wrapped into a `Node` object.
- Given the set of `Node` objects and the strategy pool associated with the constraints, the problem-space exploration code attached to the Optimisation Manager asks each `Node` to enumerate its mitigation candidates. During this step, dependencies and conflicts between mitigations are also recorded.
- The Optimisation Manager then passes the resulting sets (mitigations, coverage relations, contradictions, and required relations) to the ILP solver component, receives the subset of selected mitigations, and forwards them to the applying phase.
- Finally, the applying phase, again managed by the Optimisation Manager, executes the action terms attached to the selected mitigations on the in-memory DFD representation.

As a result, constraint evaluation and mitigation enumeration are fully decoupled from ILP encoding and solving. Only abstract identifiers and action terms flow across this boundary, which simplifies testing and incremental evolution.

6.3.3 User Interaction

From a user perspective, the system is configured by supplying three inputs: a DFD model, a set of constraints, and optional custom mitigation strategies.

DFD Model The model must be provided in one of the formats supported by the underlying DFA framework (either the web JSON format or the combination of data dictionary and data flow diagram files). The Optimisation Manager delegates parsing and semantic checks to the DFA framework and keeps only the resulting internal representation.

Constraints Constraints can be attached using the DSL or via evaluation functions.

For DSL-based constraints, the user writes definitions such as:

```
Constraint localLogging = new Constraint(new ConstraintDSL().ofData()
        .neverFlows()
        .toVertex()
        .withCharacteristic("Stereotype", "internal")
        .withoutCharacteristic("Stereotype", "local_logging")
        .create());
```

Listing 1: Constraint 1 – local logging expressed in the DSL

This example encodes that internal data may not reach vertices that lack the `local_logging` stereotype. Because the constraint follows a standard never-flows pattern, the implementation can automatically derive label-addition strategies (adding `local_logging`) and, if needed, label-deletion strategies for non-core labels on the destination.

For more complex conditions, users implement an `EvaluationFunction` that returns the set of violating nodes based on arbitrary analysis over the flow graphs. The example below checks that data logged at any node annotated with `local_logging` is eventually transmitted to a dedicated logging server:

```
EvaluationFunction evalLoggingServer = new EvaluationFunction() {
    @Override
    public Set<Node> evaluate(DFDFlowGraphCollection flowGraph) {
        Set<Node> violatingNodes = new HashSet<>();

        for (var transposeFlowGraph: flowGraph.getTransposeFlowGraphs()) {
            for (var node: transposeFlowGraph.getVertices()) {
                if (!hasNodeCharacteristic(node, "Stereotype", "local_logging")) {
                    continue;
                }

                if (transposeFlowGraph.stream()
                        .anyMatch(vertex -> hasNodeCharacteristic(
                                vertex, "Stereotype", "logging_server"))) {
                    continue;
                }

                if (!checkAcrossTFGs(flowGraph.getTransposeFlowGraphs(),
                        node, "Stereotype", "logging_server")) {
                    var vertex = (DFDVertex) node;
                    violatingNodes.add(new Node(vertex, transposeFlowGraph,
                    ↪  loggingServer));
                }
            }
        }
        return violatingNodes;
    }
};
```

Listing 2: Constraint – logged data must reach a logging server

Here, `checkAcrossTFGs` performs a cross-TFG reachability check to logging servers. Nodes that perform local logging but have no such path are reported as violations and wrapped as `Node` objects. Since this constraint goes beyond the expressiveness of the label-based DSL, the tool cannot derive strategies automatically; suitable mitigation strategies (e.g. adding a logging server or redirecting flows) must be supplied manually.

Mitigation Strategies Finally, users may provide additional mitigation strategies, especially for structural changes that cannot be inferred from DSL constraints. Strategies are defined by specifying a list of labels, a cost, and a strategy type. For example:

```
MitigationStrategy addAuthServer = new MitigationStrategy(
        List.of(new Label("Stereotype", "auth_server")),
        5,
        MitigationType.AddNode);
```

Listing 3: Custom mitigation strategy – adding an authentication server node

This strategy instructs the system to add a new node with stereotype `auth_server` when selected by the ILP solution. Additional labels in the list can describe further properties or outgoing data characteristics of the new node. Once registered, such strategies can be referenced by constraints and instantiated for concrete violations during problem-space exploration.

7 Evaluation

In this chapter, we present the evaluation of our approach. To establish an evaluation structure, we use the Goal Question Metric (GQM) approach as introduced by Basili et al. [5]. The GQM approach requires that each evaluation goal specify a *purpose*, the *issue* to be examined, the *object* under study, and the *viewpoint* from which the evaluation is conducted. We further structure this chapter as follows: Section 7.1 covers the evaluation design, including four evaluation goals and their associated questions and metrics. In Section 7.2, we give a detailed description of the concrete setups used to evaluate each question. In Section 7.3, we present and discuss the findings for each goal. To conclude this chapter, we discuss the assumptions and limitations of our contribution in Section 7.5, the threats to validity following the categorisation of Runeson et al. [39] in Section 7.4, and data availability in Section 7.6.

7.1 Evaluation Design

We define four evaluation goals:

G1 **Analyze** the mitigation results **for the purpose of** evaluating the effectiveness **with respect to** the elimination of confidentiality violations **from the viewpoint of** the software architect.

G2 **Analyze** the design of the mitigation framework **for the purpose of** characterizing its extensibility **with respect to** the support for diverse mitigation strategies **from the viewpoint of** the tool developer.

G3 **Analyze** the mitigation recommendations **for the purpose of** evaluating the cost **with respect to** the invasiveness of the changes applied to the architectural model **from the viewpoint of** the software architect.

G4 **Analyze** the optimization process **for the purpose of** evaluating the scalability **with respect to** the computational runtime under increasing model complexity **from the viewpoint of** the tool developer.

7.1.1 Studied Scenarios

The validation of the optimisation approach is carried out using the MicroSecEnD dataset provided by Schneider et al. [43]. `MicroSecEnD` contains 17, security-enriched DFD models of microservice applications corresponding to open-source implementations, such as the model of the *jferrater* application shown in Figure 7.1. The *jferrater* application is a microservice architecture project for restaurants and stores. In addition to the models, the dataset provides a set of confidentiality constraints derived from architectural security best practices, covering aspects such as authentication requirements, encryption of sensitive data flows, and the presence of logging infrastructure. Multiple model variants are included, in which specific constraints are either satisfied or deliberately violated, enabling systematic evaluation across a range of compliance scenarios. These features ensure that the evaluation is grounded in realistic and well-structured architectural scenarios that have already been applied in prior research on confidentiality analysis. By relying on `MicroSecEnD`, we achieve both external credibility and internal accuracy, providing a strong empirical foundation for assessing the effectiveness, extensibility, and cost of the chosen optimisation strategy.

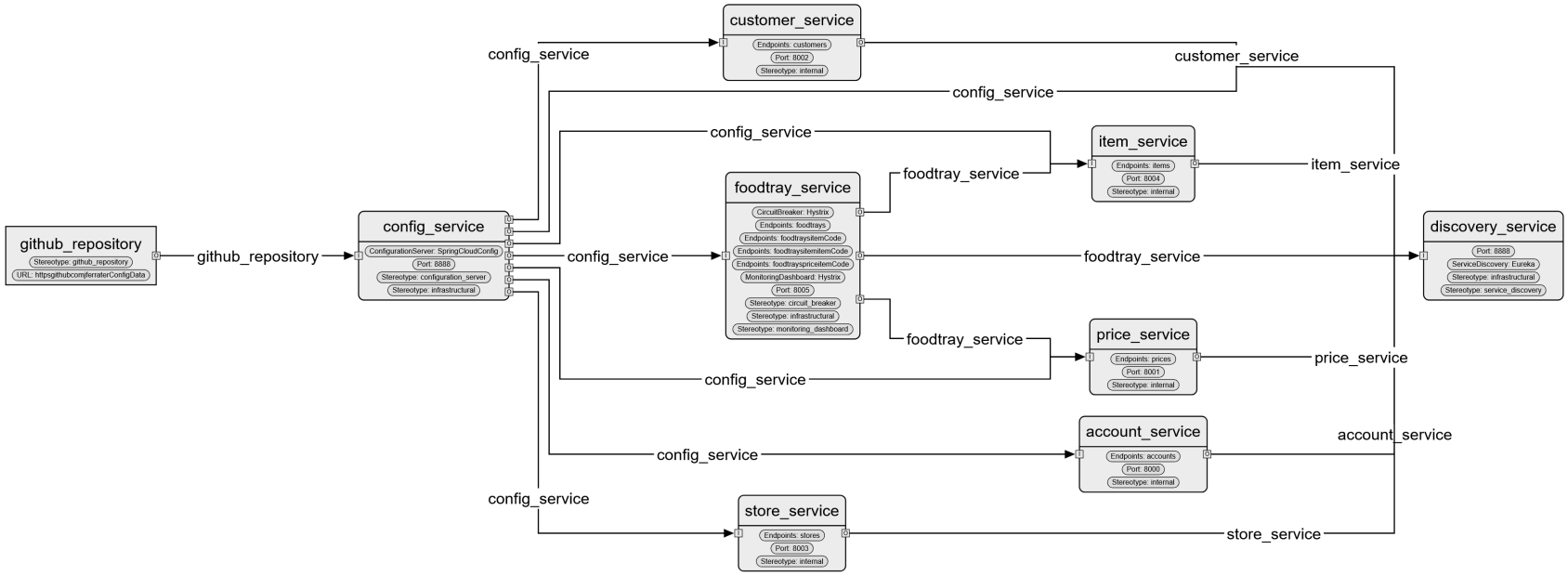

Figure 7.1: Example of MicroSecEnD DFD – jferrater

7.1.2 Evaluation Design for Effectiveness (G1)

To evaluate the effectiveness of the mitigation approach, we apply the optimisation to each studied DFD scenario. For each scenario, we record the set of violations identified by the confidentiality analysis before mitigation and the set of violations remaining after the optimised mitigations have been applied.

The primary evaluation criterion is that an effective approach should eliminate all identified violations. We additionally examine whether residual violations occur in specific scenarios and, if so, characterise their nature. This raises the following questions:

Q1.1 Does the optimised solution successfully reduce or eliminate all confidentiality violations identified in the DFD?

Q1.2 Are there scenarios in which the approach fails to resolve all violations, and if so, what characterises these cases?

To answer, we use the following metrics:

M1.1 Number of confidentiality violations before and after applying the recommended mitigations. An effective approach should reduce this number to zero for all studied scenarios.

M1.2 Classification and count of residual violations after mitigation, categorised by violation type and constraint characteristics.

7.1.3 Evaluation Design for Extensibility (G2)

To evaluate the extensibility of the approach, we need to argue why the additional mitigation strategies introduced in Section 6.2 are necessary in practice. The original SAT-based baseline can only add labels, whereas our ILP-based framework also supports deleting labels, adding nodes or sinks, and removing nodes or flows. These structural strategies are more invasive, but they are indispensable for resolving certain confidentiality violations. In particular, constraints that forbid specific labels (e.g., personal data may not be stored on a non-EU server) or require additional processing (e.g., authentication before data leaves the internal domain) cannot be satisfied by adding labels alone. Node- and flow-level strategies provide structural flexibility when neither label additions nor deletions are enough.

The necessity of these strategies depends purely on the specific constraints in a model. Therefore we stick to a purely argumentative evaluation with the following questions:

Q2.1 For which classes of confidentiality constraints is label addition sufficient to resolve a violation, and for which classes is it inherently insufficient?

Q2.2 When adding labels is insufficient, do the user-defined strategies: label deletion, node or sink insertion, and flow removal, repair the violation without compromising the intended behaviour of the DFD?

7.1.4 Evaluation Design for Cost (G3)

To evaluate the cost of the mitigation approach, we focus on the invasiveness of the modifications produced by the optimisation. A good mitigation strategy should strive to be as non-intrusive as possible, preserving most of the original design and functionality while resolving confidentiality issues. This raises the following questions:

Q3.1 How invasive are the changes that the approach recommends to the DFD?

Q3.2 How does the invasiveness of the computed solution compare to baseline approaches from prior work?

We use the following metrics to answer these questions:

M3.1 An invasiveness metric defined as a sum of model modifications, where different types of changes carry different costs. For instance, removing a component is considered more invasive than adding an encryption label to an existing data flow. The concrete weight assignments are documented in the evaluation setup.

M3.2 Compare the invasiveness between the solution produced by our approach and alternative mitigation strategies or manual resolutions documented in the literature.

7.1.5 Evaluation Design for Scalability (G4)

Scalability refers to the behaviour of the approach as the size and complexity of the DFD models grow. A scalable mitigation framework should maintain acceptable execution times even as the number of nodes, flows, constraints, and label combinations increases. This is important because real-world architectures can be substantially larger than those provided in `MicroSecEnD`; an approach that scales poorly would be impractical for industrial use. The execution time of the optimisation depends on the structure and size of the DFD, the number of violations, and the number of available mitigation strategies. We focus on the scalability of the model aspects that are most relevant to the optimisation performance. This leads to the following questions:

Q4.1 How does the execution time of the optimisation scale when increasing the length of the TFG?

Q4.2 How does the execution time scale when increasing the number of TFGs?

Q4.3 How does the execution time scale when increasing the total number of constraints?

Q4.4 How does the execution time scale when increasing the number and variety of labels in the constraints (both for data and nodes, and both as positive and negative conditions)?

Q4.5 How does the execution time scale when increasing all dimensions of constraints?

We use the execution time needed for each run of the optimisation as metric **M4** to evaluate the questions concerned with G4. The time needed to repair models for each scaling is measured while corresponding model aspects are incremented individually, and all other aspects are held constant. The execution times at each increment are plotted to visualise the general tendency of the runtime behaviour.

7.2 Evaluation Setup

This section defines the setups we use to answer the questions raised in the previous section. These setups describe the DFD instances and constraints used to evaluate the effectiveness, extensibility, cost, and scalability of our approach.

7.2.1 Setups for Evaluating Effectiveness (G1)

For G1, we use all MicroSecEnD variants for which confidentiality constraints are defined. Each variant is first analysed with the existing DFA-based analysis to obtain the baseline set of violations. We then apply the ILP-based repair restricted to label-based mitigations with the same costs to ensure comparability with the SAT-based approach. After applying the computed mitigations, we rerun the analysis to obtain the post-mitigation violation count.

The effectiveness results are aggregated in two views: Figure 7.3 shows, for each variant, the total number of violations before and after repair across all constraints; Figure 7.4 refines this perspective by plotting violations per constraint and per model. Together, these plots allow assessing whether all violations are removed and how the initial violation density differs between systems.

7.2.2 Setups for Evaluating Extensibility (G2)

To evaluate the extensibility of the mitigation approach, the focus lies on those confidentiality violations that cannot be resolved by the SAT-based baseline, which only supports adding labels to existing nodes and flows. The evaluation is therefore based on representative constraint scenarios and their qualitative analysis, rather than on quantitative metrics or exhaustive enumeration of all possible models. For each evaluation question, Q2.1 and Q2.2, suitable scenarios are constructed, and it is argued whether (i) label addition alone suffices and, if not, (ii) the additional user-defined strategies are able to repair the violation without compromising the intended behaviour of the DFD.

For Q2.1, we distinguish two classes of violations on the example model:

- **Label-completeness violations**, where a constraint is violated solely because a required label is missing on an otherwise suitable node or flow. On the example model, this applies to Constraint C1, C2, C5, and C9: missing logging annotations, missing authentication labels, and missing encryption labels can all be resolved by adding the respective labels at the correct pins or nodes without changing the DFD structure.
- **Structural and negative-label violations**, where constraints interact such that label addition alone cannot yield a repair. On the example model, this occurs with the logging and authorisation server (Constraint C3, C4). Additionally, where the privacy and residency constraints with the logging and storage (Constraint C6-8). Because the label-propagation semantics are monotonic, once a label such as `PersonalData` or `Admin` is attached to data, it propagates until explicitly removed. There is no way to satisfy both groups by adding labels only.

For Q2.2, these structural violations are mapped to the individual strategy types of the ILP-based framework. We argue, per strategy, that it resolves the targeted violation without compromising the intended behaviour of the DFD—defined as the conjunction of all nine constraints and all functionally required data flows (Chapter 4).

7.2.3 Setups for Evaluating Cost (G3)

For G3, we define an invasiveness metric that assigns a cost to each type of mitigation (action) supported by the framework. Conceptually, the metric distinguishes:

- label additions (e.g., adding `Encrypted` or `Sanitized` to a flow),
- label deletions (e.g., removing a location or stereotype),
- node additions (e.g., adding a logging or authorisation node),
- node deletions, and
- flow deletions.

Throughout the G3 evaluation, each label addition is assigned a unit cost of 1, and the remaining mitigation types are excluded. This restriction ensures comparability with the SAT-based baseline, which operates exclusively on label additions, so that any observed differences in the resulting mitigation sets can be attributed to the solver technology and search strategy rather than to the use of different action families.

7.2.4 Setups for Evaluating Scalability (G4)

This section describes the experimental setup for each scaling dimension. All scalability experiments are based on a minimal source-to-sink model(see Figure 7.2) consisting of two nodes connected by a single flow and containing exactly one violation. This base model is then incrementally extended along one dimension at a time, while all other model aspects

are held constant, ensuring that any observed change in runtime or solution quality is attributable solely to the varied dimension.

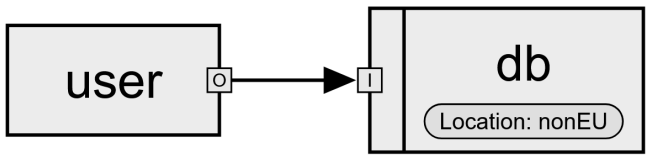

Figure 7.2: Source-Sink model for Scalability evaluation

Setup for Q4.1 – Length of TFGs The length of a TFG, defined as the number of nodes per path, directly influences the number of nodes analysed by the confidentiality analysis and our approach. Longer paths may expose more points at which confidentiality properties may be violated, requiring additional mitigations such as encryption, sanitisation, or adjusted storage locations. We scale this dimension by extending the paths in the DFD with additional nodes.

Setup for Q4.2 – Number of TFGs The number of TFGs is determined by the number of distinct source-sink pairs in the DFD and can be scaled by introducing additional sink nodes. A higher number of TFGs generally leads to a greater number of nodes and flows because every element is unique in its TFG, even if it occurs in multiple TFGs. This results in a higher amount of violations, nodes and flows. However, many of these violations may trace back to a common insecure element in the shared DFD, meaning that a single mitigation may resolve violations across multiple TFGs simultaneously.

Setup for Q4.3 – Number of Constraints The total number of constraints determines the density of the requirement network. Each additional constraint introduces at least one further violation. Moreover, a denser constraint network restricts the feasible region for valid mitigation combinations more tightly, which increases both the number of detected violations and the difficulty of finding a set of mitigations. As constraints accumulate, interactions between them may generate contradictions or require complex combinations of strategies, thereby expanding the search space for the ILP solver. By scaling the number of constraints, we evaluate how well the solver handles a richer set of requirements.

Setup for Q4.4 – Labels in Constraints We vary the number and variety of labels that appear in constraints in four sub-dimensions:

1. number of `withLabel` conditions on data (data with labels),

2. number of `withoutLabel` conditions on data (data without labels),
3. number of `withCharacteristic` conditions on nodes, and
4. number of `withoutCharacteristic` conditions on nodes.

These sub-dimensions correspond to the positive and negative label conditions on both data and nodes that define the semantics of the *neverFlows*-style constraints introduced earlier. An increasing number of `withLabel` conditions on data specifies a larger set of data that must be protected by the subsequent conditions; in the absence of other dimensions, such violations can only be repaired by removing the entire flow. By contrast, adding more `withoutLabel` introduces additional negative label requirements; this dimension enables additive data mitigations that can be addressed by adding authentication or encryption labels. `withCharacteristic` conditions on nodes prohibit receiving nodes from exhibiting certain characteristics (e.g., `internal` or `non-EU`); corresponding repairs consist of deleting the offending label. Finally, `withoutCharacteristic` conditions on nodes describe characteristics that can resolve a violation when they are added as node characteristics.

When multiple label conditions occur within a single constraint following a neverFlows clause, the constraint must be transformed into CNF to be representable using the ILP approach; however, each additional condition increases the number of clauses and variables, thereby raising the complexity of the optimisation problem. For each sub-dimension, we increase only the respective number of label conditions, keeping the others constant.

Setup for Q4.5 – All Constraint Dimensions at Once Finally, we scale all constraint-related dimensions together by proportionally increasing the number of constraints and the number of label and characteristic conditions in each constraint. This provides an upper-bound scenario that captures the combined effect of richer constraint sets on the solver.

All scalability experiments are conducted on the following configuration:

- **Processor:** Intel 12700KF
- **RAM:** 16 GB
- **Operating System:** Win 11.25H2
- **Solver Version:** OR-Tools v9.10

7.3 Results and Discussion

In this section, we present and discuss the findings of our evaluation. The results are organised according to the four evaluation goals defined in Section 7.1.

7.3.1 Effectiveness (G1)

Findings on Q1.1 Figure 7.3 shows, for each MicroSecEnD variant, the number of confidentiality violations before and after repair across all constraints. Across all variants, the values "after the repair" are zero, indicating that the ILP-based repair always produces a model without remaining violations. The number of initial violations differs substantially between applications, ranging from a few dozen to several thousand. This discrepancy can be attributed to variations in system size, topology and constraint density. However, the approach consistently identifies a set of mitigations that eliminates all of the violations.

Figure 7.4 refines this view by plotting violations per constraint and per model. Again, all "after repair" values are zero, indicating that, for each individual constraint, all previously detected violations are removed. This confirms that the optimisation formulation correctly captures the semantics of the confidentiality constraints and that the solution space explored by the ILP is sufficient to resolve all violations with additive repairs. Taken together, Figure 7.4 and Figure 7.3 demonstrate that, within the scope of the studied dataset, the optimisation achieves full effectiveness.

Findings on Q1.2. To verify that the repair process does not introduce new violations, we re-executed the DFA on every repaired model and compared the post-repair violation set against the pre-repair baseline. In all studied scenarios, the post-repair violation count is zero, and no violation present in the repaired model was absent from the original. This confirms that the ILP formulation, by construction of the contradiction constraints in Section 6.1, cannot select mutually exclusive mitigations that would produce a new structural inconsistency. Scenarios in which Q1.2 could produce non-trivial findings, for instance, models with stricter constraints or intentionally forbidden strategies, are discussed as a threat to validity in Section 7.4.

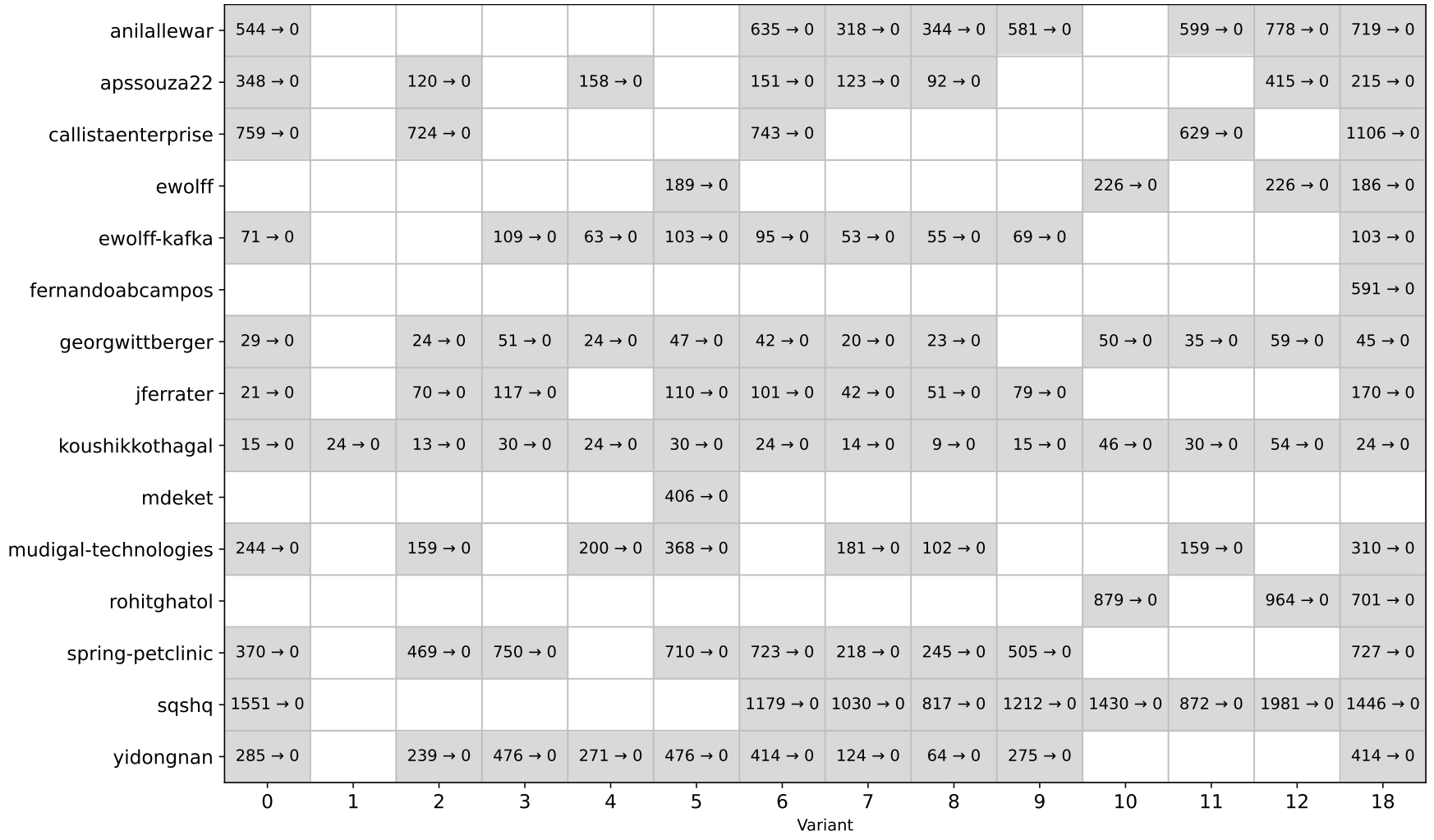

Figure 7.3: Violations before and after Repair for each variant and all constraints

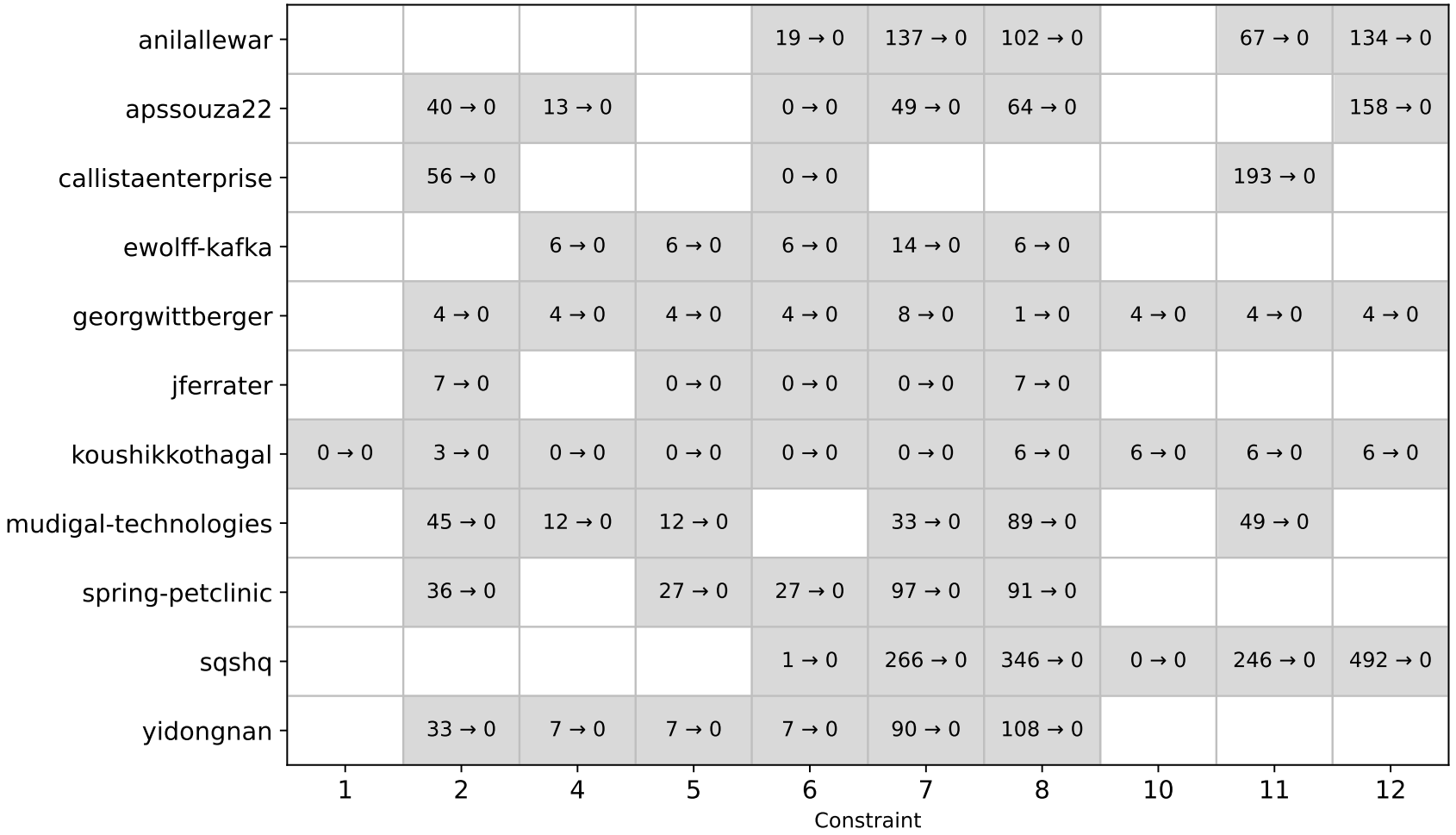

Figure 7.4: Violations before and after Repair for each model and constraint

7.3.2 Extensibility (G2)

Discussion on Q2.1. Q2.1 asks for which violations label addition alone suffices, and where it is inherently insufficient. On the example model, constraints 1, 2,5, and 9 can each be satisfied purely by enriching existing flows and nodes with appropriate labels. Internal services can be marked as logged (Constraint C1); logged data can be sanitised (Constraint C2); all internal communication can be marked as sent via an authenticated channel (Constraint C5); and admin flows can be encrypted (Constraint C9). In all cases, the DFD structure remains unchanged, confirming that the SAT-based baseline is adequate for this subset of constraints.

However, the Constraint C3 and Constraint C6-8 demonstrate clear limits. Logging requires that information be propagated to a logging server, which must be explicitly added to the model. Privacy and residency additonally constraints forbid personal or admin data from actually appearing on logging or non-EU storage nodes. Because label propagation is monotonic, adding more labels cannot remove the offending `PersonalData` or `Admin` labels from materialised log data. Furthermore, Constraint C7 forbids personal data on a non-EU server regardless of any additional annotations. Consequently, label addition is necessary but not sufficient to satisfy all constraints simultaneously.

Discussion on Q2.2. Q2.2 asks whether the user-defined structural strategies can repair the remaining violations without compromising the intended behaviour. In the example model, each strategy type becomes necessary at least once:

- **Label deletion.** When internal services log user requests, the resulting log also records personal data. To satisfy Constraint C6, this label must be removed from the data sent to the logging server. Deleting `PersonalData` on the outgoing log flow, while retaining diagnostic attributes, resolves the conflict without altering the functional data flows of the system.
- **Node insertion.** Constraint C4 requires the authorisation of data from an entrypoint using an authorisation server. Inserting an architect-approved authorisation process on the internal flows ensures compliance with constraint 4.
- **Sink insertion.** Constraint C3 requires all locally logged records to be forwarded to a central logging server. The new sink therefore does not alter the core request-response behaviour.
- **Flow or node removal.** If the *Storage* node is deployed as a non-EU server and currently receives flows carrying personal data, Constraint C6 is violated regardless of any label changes. Removing the offending flows from personal-data sources to *Storage*, or replacing the non-EU node with an EU-based alternative, eliminates the propagation path. To avoid loss of essential functionality, the architect restricts removal eligibility to flows and nodes that can be substituted.

In each case, the strategy usage is governed by user-defined constraints and costs in the ILP objective: destructive actions such as flow or node removal should receive higher costs than label manipulations or node insertions, so the solver prefers less invasive repairs whenever possible. Under these assumptions, all remaining violations on the example model can be repaired while preserving the functionally required data flows. The example thus provides a single, coherent argument that every strategy type offered by the ILP-based framework is not only available but in fact necessary to satisfy realistic combinations of operational access-control and legal constraints.

7.3.3 Cost (G3)

To ensure comparability across all three evaluated approaches, repair cost is measured uniformly as the number of additive architectural changes applied, with each individual addition assigned a unit cost of 1. This restricted metric ensures that no approach is penalised or advantaged by differing change types; all three baselines are compared on the same operational ground. Structural operations such as node or flow deletion are therefore excluded from the cost calculation for all approaches in this comparison and are thus disallowed for ILP in this experiment.

Findings on Q3.1. Figure 7.5 presents, for each model-constraint combination in the dataset, the cost difference between the repair recommended by the ILP-based approach and the corresponding human-created repair from the MicroSecEnD baseline, computed as:

$$\Delta_{\text{cost}} = \text{Cost}_{\text{ILP}} - \text{Cost}_{\text{Manual}}$$

Green cells indicate that the ILP approach is less invasive than the human baseline, grey cells indicate equal invasiveness, and red cells indicate that the ILP solution requires more modifications than the human baseline. Across all $n = 51$ model-constraint pairs, the ILP approach accumulated a total repair cost of 271, while the human baseline incurred a total cost of 1,005. This corresponds to an absolute cost reduction of 734 units, representing a relative decrease of approximately 73% in architectural invasiveness.

Visual inspection of Figure 7.5 confirms this general trend: the vast majority of cells are green, a small number of grey cells reflect scenarios in which both approaches arrive at equally invasive repairs, and only one red cell indicates cases where the ILP approach results in a higher cost than the human baseline. This exception arises from a constraint of the current evaluation design: the ILP approach is limited to label-addition mitigations and therefore does not allow the removal of nodes, flows, or labels. In this particular case, the human annotator resolved the violation by deleting an *essential* functional node from the DFD, which fundamentally changes the system's topology and behaviour. Because the ILP approach must restore compliance while preserving behaviour, it is forced to apply a larger number of individual modifications in such scenarios. These cases therefore do not represent a shortcoming of the ILP solver; rather, they are a direct consequence of the

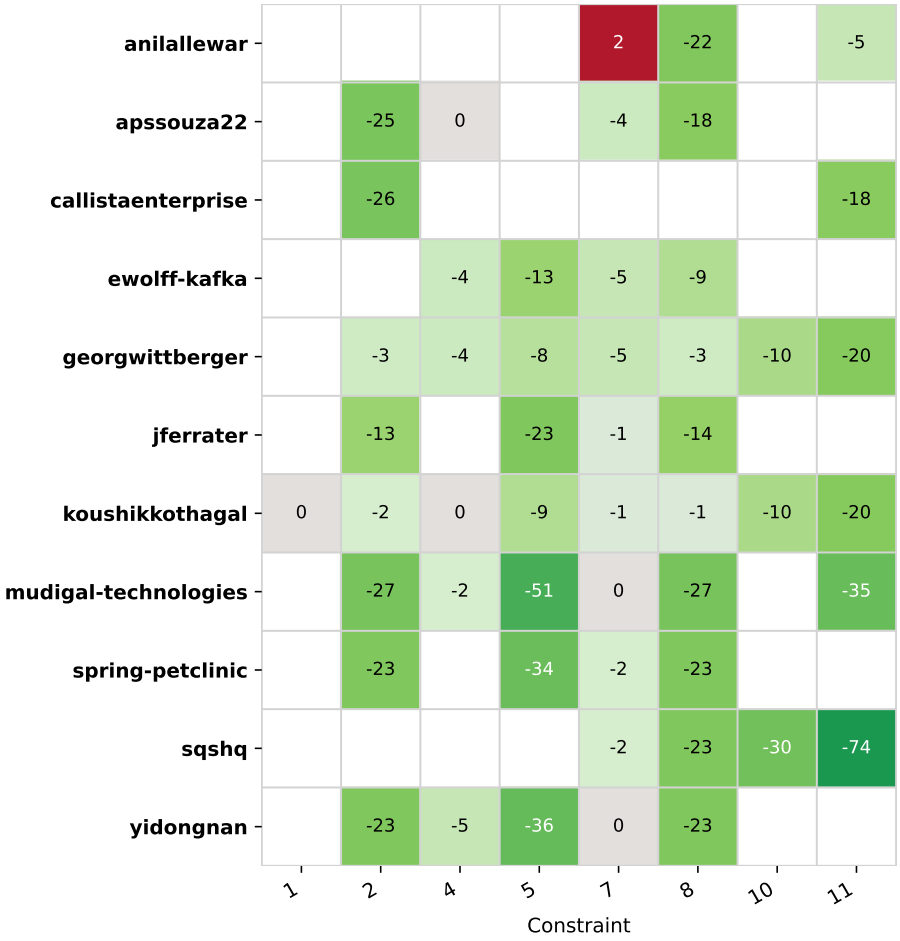

Figure 7.5: Comparison of ILP and Manual mitigation invasiveness

constraint that the repaired architecture must preserve the full functional intent of the original system.

To formally assess whether the ILP approach yields significantly less invasive repairs than the human-created baseline, we state the following hypothesis:

Hypothesis: *Repairs produced by the ILP approach are significantly less invasive than those of the human-created baseline*

We evaluate this hypothesis using the Wilcoxon signed-rank test applied to the $n = 51$ paired cost differences across all model-constraint combinations. Pairs with a cost difference of zero are excluded under the standard Wilcoxon procedure, and the test is conducted as one-sided, reflecting the directional hypothesis that the ILP approach is cheaper. The test reveals a statistically significant reduction in repair cost ($W = 1075.0$, one-sided $p = 2.56 \times 10^{-9}$). The effect size is large ($r = 0.860$), indicating that the cost advantage of the ILP approach is not only statistically robust but also practically substantial.

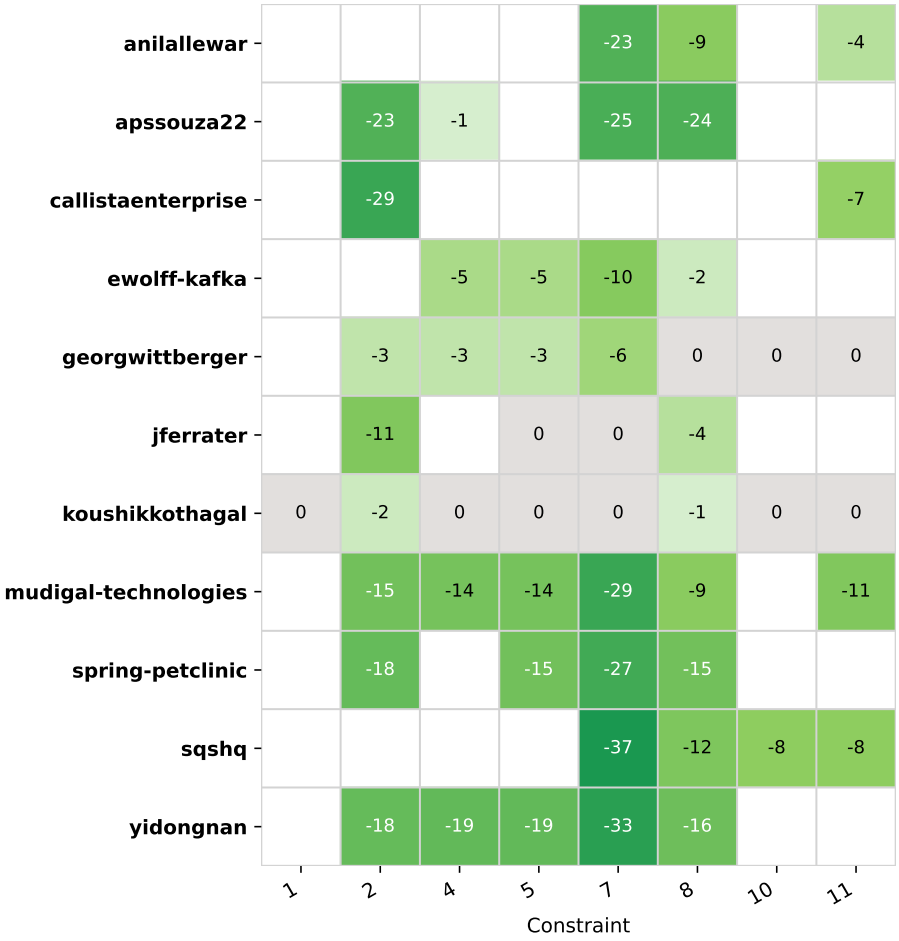

Figure 7.6: Comparison of ILP and SAT mitigation invasiveness

Findings on Q3.2. Figure 7.6 presents the cost difference between the ILP approach and the SAT-based baseline for each model-constraint combination, computed as:

$$\Delta_{\text{cost}} = \text{Cost}_{\text{ILP}} - \text{Cost}_{\text{SAT}}$$

Across all $n = 51$ model-constraint pairs, the SAT-based approach incurred a total repair cost of 808, compared to 271 for the ILP approach, corresponding to a reduction of 537 units, or approximately 66%. All cells in Figure 7.6 are either green or grey, indicating that the ILP approach never produces a more invasive repair than the SAT-based baseline for any individual model-constraint combination. This result is particularly meaningful because, in this comparison, both approaches operate under the same additive-only, unit-cost constraint space. Any observed cost difference is therefore attributable exclusively to the optimisation strategy employed, not to differences in the types of available mitigations.

The ILP approach consistently produces cheaper repairs because its formulation allows shared mitigations to cover multiple violations simultaneously, especially in cases where data is forwarded through the entire system. When a single label addition resolves several

TFG violations at once, the ILP solver exploits this by counting the mitigation's cost only once in the objective function. The SAT-based approach, by contrast, encodes each violation independently and may select redundant or overlapping repairs, missing opportunities for such cost reductions.

An analogous Wilcoxon signed-rank test confirms that this cost reduction is statistically significant ($W = 820.0$, one-sided $p = 1.77 \times 10^{-8}$), with a large effect size ($r = 0.870$).

Discussion. The combined evidence from Figures 7.5 and 7.6 and the statistical tests supports a clear conclusion: the ILP-based repair consistently produces less invasive architectural modifications than both the human-created baseline and the SAT-based automated approach. The 73% reduction in cost relative to the manual baseline demonstrates the practical value of optimisation-driven automated repair for real-world microservice architectures. The 66% reduction relative to the SAT baseline further shows that formalising the repair problem as an ILP, rather than a satisfiability instance, enables the solver to exploit the combinatorial structure of shared mitigations, yielding solutions that are not only correct but also structurally minimal. The few cases in which the human baseline outperforms the ILP approach highlight an inherent trade-off between behaviour preservation and minimality: permitting functional node removal would close this gap, but at the cost of potentially altering system behaviour.

7.3.4 Scalability (G4)

This subsection discusses, for each defined scaling dimension, how the execution time of the ILP-based repair behaves and how it compares to the SAT-based baseline. We explicitly distinguish between the three measured times: *solving only* (problem-space exploration plus ILP solving), *isolated* (mitigation workflow without DFA), and *total* (end-to-end including DFA).

Each scaling dimension is accompanied by six graphs, grouped into two figures. The first figure contains four plots that characterise the runtime behaviour of the ILP-based approach in isolation. The *top-left* plot shows all three time components (total, isolated, and solving-only) on a *linear scale*, making absolute differences and the overall growth shape directly readable. The *top-right* plot shows the same three components on a *logarithmic scale*, which reveals relative differences and low-magnitude behaviour that would be invisible on the linear plot. The *bottom-left* plot shows the *absolute external DFA analysis overhead* in milliseconds, computed as the difference between total and isolated time; this isolates the cost attributable to the upstream confidentiality analysis from the cost attributable to the mitigation workflow itself. The *bottom-right* plot shows the same overhead as a *percentage of total runtime*, making it straightforward to judge whether the DFA analysis or the ILP solving step dominates the end-to-end execution time at each dimension value. The second figure contains two plots that directly compare the ILP approach against the SAT-based baseline. The *top* plot shows the *total execution time* of both approaches on a logarithmic scale, allowing order-of-magnitude differences to be assessed across the full dimension range. The *bottom* plot shows the *isolated execution time* of both approaches on a logarithmic scale, which removes the shared DFA overhead and therefore isolates any difference that is attributable to the solver itself rather than to the common analysis infrastructure.

Findings on Q4.1 – Length of TFGs. Figure 7.7 presents the ILP runtime as the TFG path length is extended by 0 to 550 nodes per path. The linear-scale plot shows that total execution time grows from approximately 30 ms at the shortest paths to approximately 160 ms at length 550, following a visually near-linear trajectory throughout the entire range. The isolated time closely tracks the total, while the solving-only component accounts for the majority of that isolated time. The overhead plot shows that the absolute external DFA analysis cost grows from roughly 5 ms to 35 ms over the same range, and the overhead-share plot confirms that the DFA contribution stabilises in the 25%–45% band, with a mild downward trend at longer paths indicating that ILP solving becomes proportionally more dominant as path length increases.

Figure 7.8 compares both approaches on a logarithmic scale. Both the total and isolated curves of the SAT baseline occupy the upper portion of the 10^1–10^3 ms band, while the ILP curves remain in the 10^1–10^2 ms band. At the maximum tested length of 550 nodes, the SAT approach is therefore approximately one order of magnitude slower than the ILP approach on a per-run basis.

Discussion. The strictly linear growth in this dimension is directly based on the structure of the Problem Space Exploration phase. For a TFG of length L and a fixed set of k labels, each

additional node adds exactly k new label-addition candidates to the ILP: one for each label that the `withoutLabel` conditions require to be present. The ILP therefore gains exactly k new binary decision variables per additional node, while the number of unique coverage constraints increases by L, since path extensions do not introduce new types of constraint violations. The only structural complexity that grows faster than linearly is the pairwise contradiction set: two candidates that share a TFG but address different conditions may be contradictory, giving at most $O(L^2)$ contradiction constraints. In practice, however, the number of labels k is small and the contradiction density is sparse, so the observed behaviour is dominated by the linear term in L. The 25%–45% DFA overhead share reflects that the DFA analysis itself processes each TFG as a fixed structure and only needs to re-evaluate label propagation at the added nodes; this scales linearly in L as well, however, with a smaller constant factor than the ILP solving component, which explains the gradual shift of overhead share toward lower values for longer paths. The consistent one-order-of-magnitude advantage of the ILP over SAT arises because the SAT encoding must represent every possible label-assignment state of every node along the path as a Boolean variable, yielding a CNF formula that grows as $O(L \cdot 2^k)$, whereas the ILP formulation grows only as $O(L \cdot k)$.

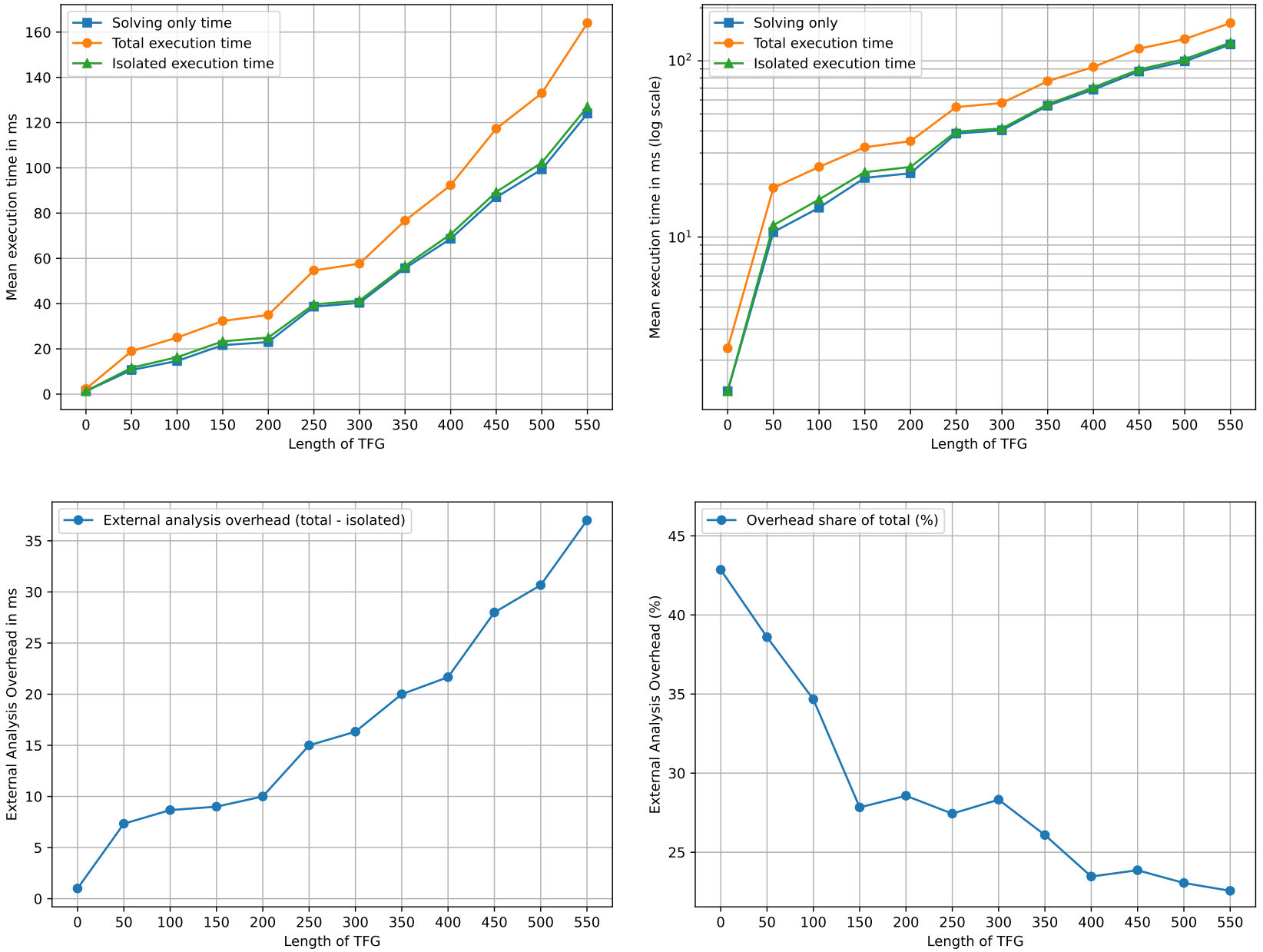

Figure 7.7: Evaluation result for Length of TFG

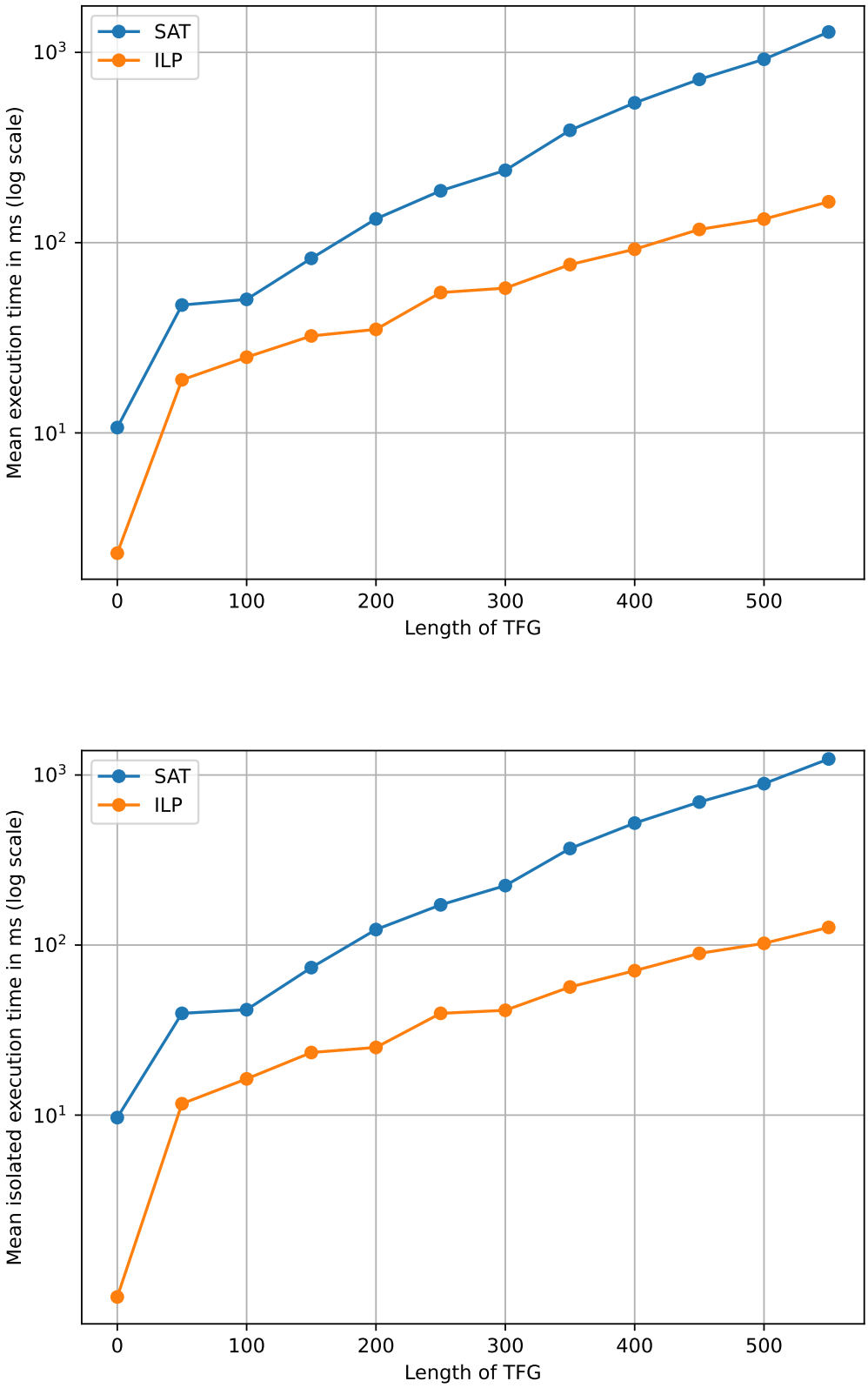

Figure 7.8: Comparison of SAT and ILP approaches for Length of TFG.

Findings on Q4.2 – Number of TFGs. Figure 7.9 shows the ILP runtime as the number of TFGs is scaled from 0 to 15,000 by introducing additional parallel sinks. The linear-scale plot reveals a polynomial growth line from near 0 ms to approximately 100,000 ms, which strongly polynomial growth in this range. At a closer look, the solving-only and isolated curves remain almost constant near the baseline throughout the entire range. The absolute overhead plot confirms this: the DFA analysis overhead also rises from 0 to approximately 100,000 ms, essentially shadowing the total curve. The overhead-share plot corroborates this observation: the DFA portion of the total runtime increases from roughly 40% to 100% as the number of TFGs grows. Notable in this plot is the abrupt spike to 100%, which indicates that the ILP solver terminated almost immediately and that the entire measured runtime was incurred by the external analysis alone.

Figure 7.10 presents the SAT-versus-ILP comparison on a logarithmic scale. At low TFG counts, both approaches are clustered around 10^1–10^2 ms. Beyond approximately 2,000 TFGs, both curves diverge into the 10^3–10^5 ms band, with neither approach maintaining a significant advantage. The isolated time, on the other hand show a clear and distinct difference between the two approaches. While ILP has a logarithmic growth below 10^2 ms , SAT has slower divergence and ends up beyond 10^5 ms. Reinforcing that the shared DFA analysis infrastructure drives the overall runtime of the ILP, but not the SAT-based approach.

Discussion. The TFG decomposition architecture of the DFA framework explains the polynomial growth in this dimension. On the ILP side, we have a linear growth, since each new TFG introduces a fixed number of new violations (proportional to the number of active constraints) and a corresponding fixed set of new mitigation candidates, yielding a linear increase in the number of binary decision variables and coverage constraints. A key structural advantage of the ILP formulation is that mitigations are defined globally over the DFD rather than per TFG: a single candidate variable can cover violations across multiple TFGs if they share the same insecure element. This shared-variable structure prevents the ILP problem size from growing at the same rate as the DFA, which explains why the solving-only component remains nearly constant while the total time grows with the DFA overhead. Since SAT does not operate on this, it grows equally to the DFA, resulting in a clear advantage for using ILP over SAT.

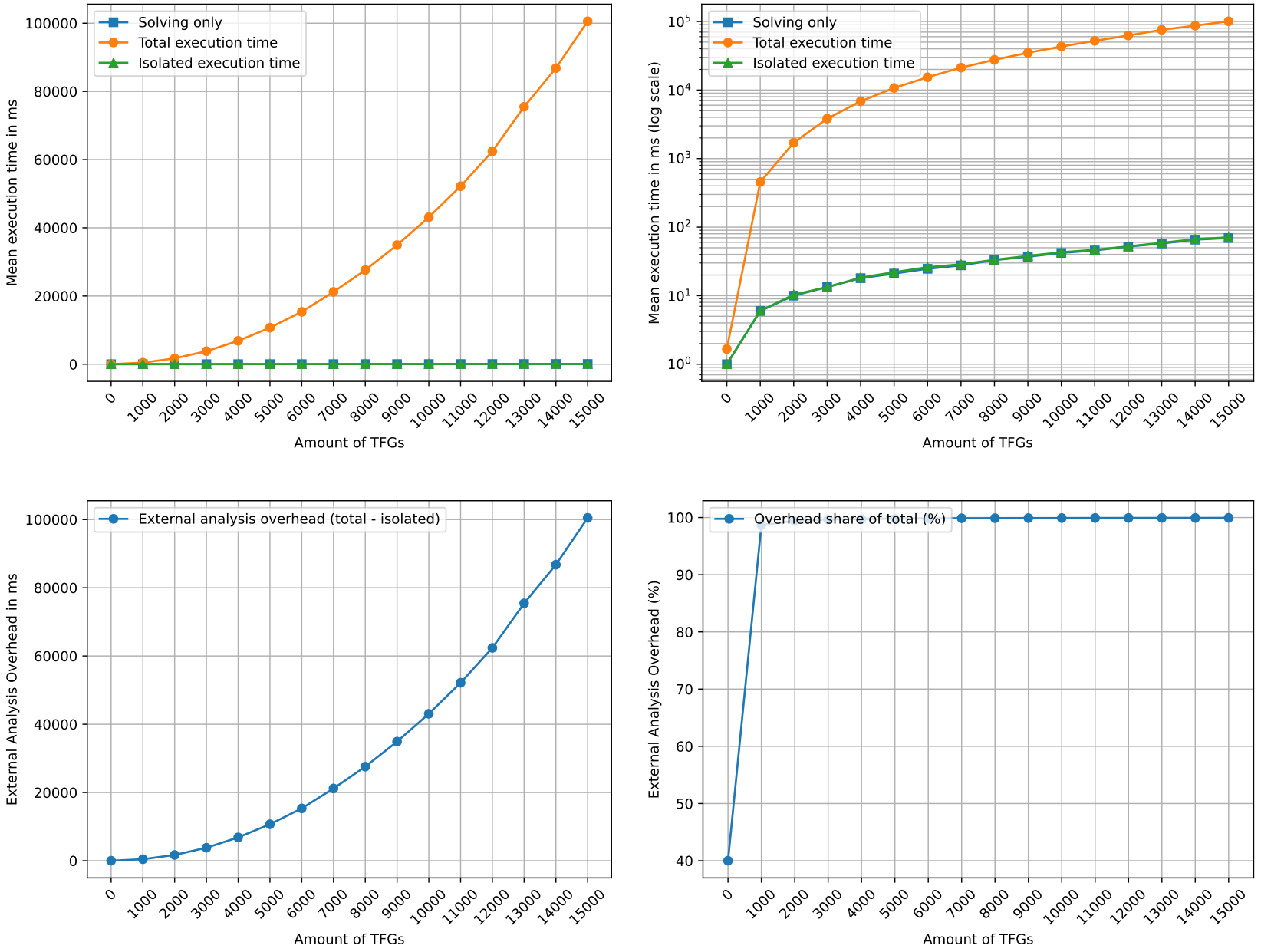

Figure 7.9: Evaluation result for Amount of TFGs

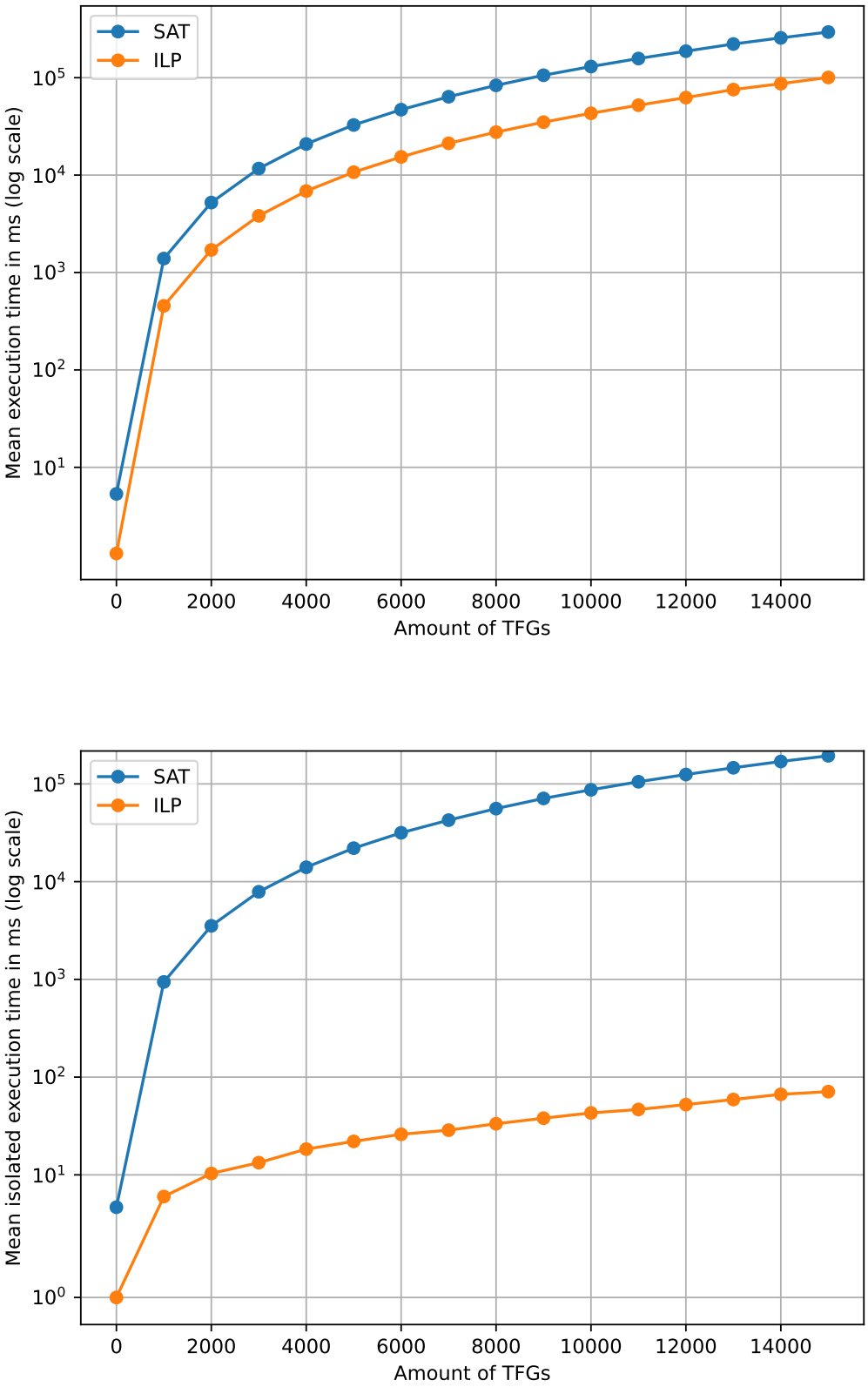

Figure 7.10: Comparison of SAT and ILP approaches for the amount of TFGs.

Findings on Q4.3 – Number of Constraints. Figure 7.11 shows ILP runtimes as the number of constraints is scaled from 1 to 140. The linear-scale plot shows total execution time rising gradually from near 0 ms to approximately 50 ms, exhibiting a linear concave growth curve. The solving-only component of the system stays constant throughout the entire measurement, indicating that scaling in this dimension does not influence the problem space exploration and solving. The isolated curve tracks the total curve, while the absolute overhead stays between 0 ms and approximately 12 ms. The overhead-share plot shows a declining trend from roughly 70% at sparse constraint sets to approximately 30% at 140 constraints, indicating that the ILP optimisation takes on a progressively larger share of total runtime as constraints accumulate.

Figure 7.12 compares SAT and ILP on a logarithmic scale for constraint counts up to 20. The ILP curve remains stable in the 10^0–10^1 ms range, whereas the SAT curve grows exponentially within the 10^4–10^5 ms range. Consequently, the SAT approach cannot be measured beyond a scaling factor of 17 and is thus considered infeasible. Across the comparison range, ILP stays constant and is between two and four orders of magnitude faster than the SAT baseline on the worst-case measurements. For greater scaling, the difference between the two approaches will grow significantly, since ILP stays nearly constant while SAT grows exponentially.

Discussion. The linear growth of total execution time with increasing constraint count is explained by the linear increase of violations $k * C$ and a linear increase of possible mitigations $3k * C$. Where k represents the number of Incoming Flows over all TFGs and C the number of constraints. The fact that the isolated execution grows faster than the solving indicates that the process of finding general mitigation strategies and desolving required and contradictory conditions takes a greater effort than solving for the concrete mitigations. One reason for this might be that the LP relaxation effectively prunes the problem space. The extreme SAT spike in Figure 7.12 reveals a fundamental structural weakness of the CNF encoding: interacting constraints may generate clause combinations that are locally satisfiable but globally contradictory, forcing the solver into deep backtracking cascades. The ILP formulation avoids this because contradiction relations between candidates are encoded as explicit linear inequalities with known structure, allowing the LP relaxation to detect and prune infeasible regions without enumeration.

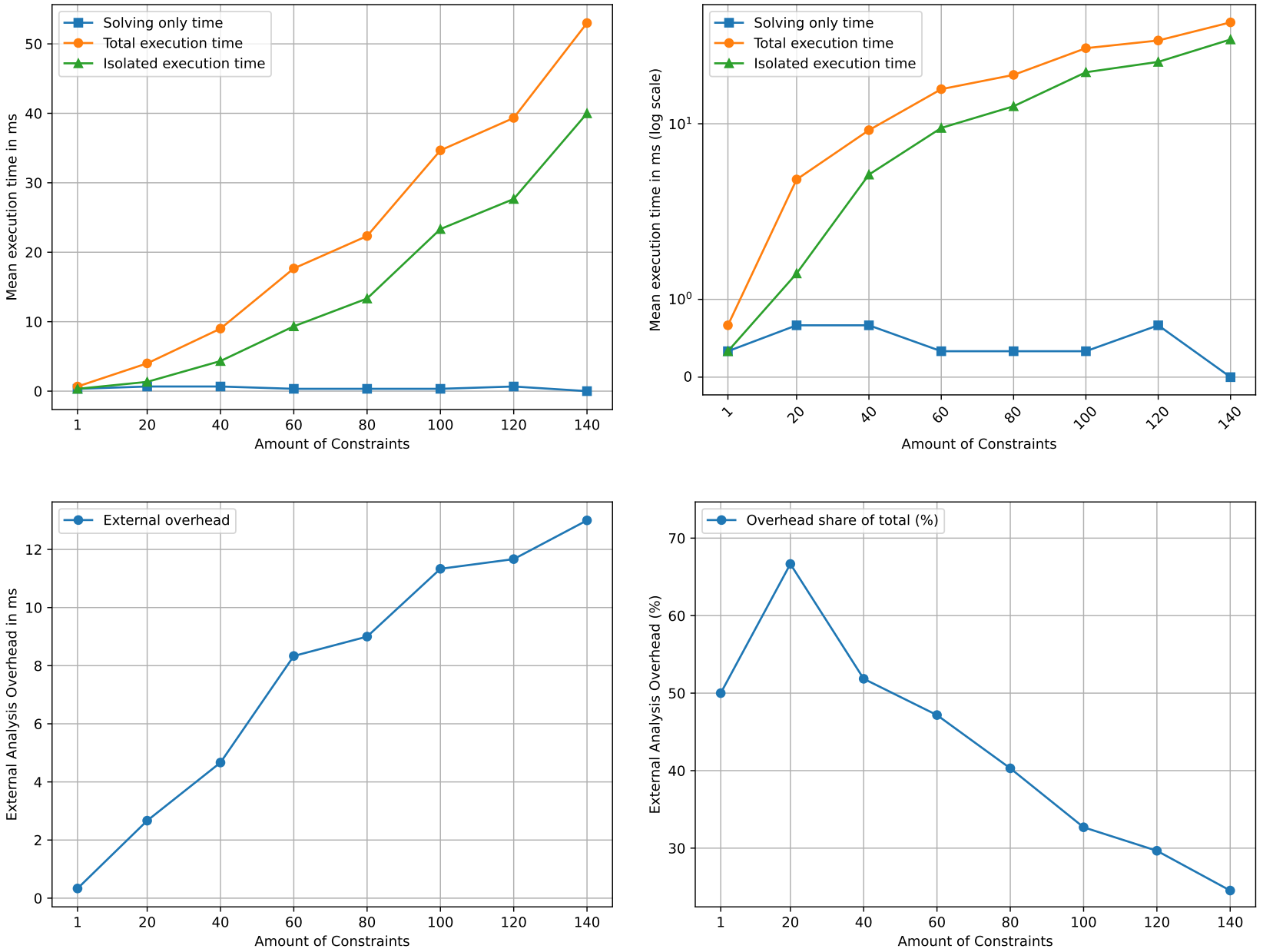

Figure 7.11: Evaluation result for Amount of Constraints

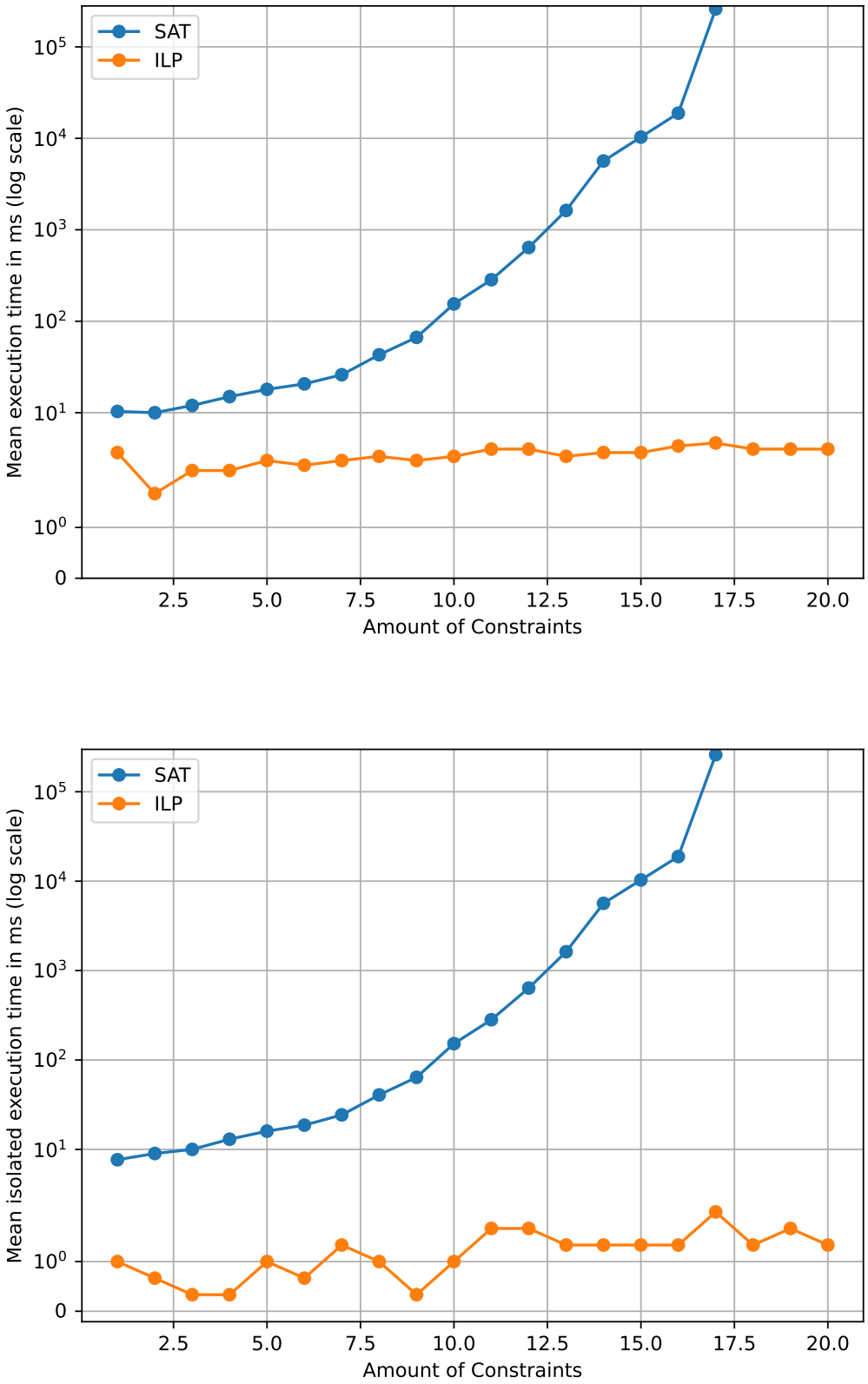

Figure 7.12: Comparison of SAT and ILP approaches for Amount of Constraints

7.3.4.1 Findings on Q4.4 – Labels in Constraints

Question Q4.4 independently varies four sub-dimensions of the constraint: `withLabel`, `withoutLabel`, `withCharacteristic`, and `withoutCharacteristic` conditions. The semantic distinction between these groups is critical for interpreting the results: *positive* conditions (`withLabel`, `withCharacteristic`) narrow the set of TFGs or nodes subject to a violation, but do not generate new additive mitigation candidates, whereas *negative* conditions (`withoutLabel`, `withoutCharacteristic`) define properties that can be *added* to resolve a violation and therefore directly expand the mitigation candidate space.

Data With Label Figure 7.13 shows that increasing the number of `withLabel` conditions from 1 to 140 yields a near-flat runtime. When plotted on the linear scale, the three curves (total, isolated and solving-only) are observed to cluster within the 0–5 ms band, exhibiting no clear upward trend. The absolute overhead plot confirms that the DFA analysis cost stays between 0.5 ms and 4 ms and is effectively constant, while the overhead-share plot is the most diagnostic feature of this dimension: the DFA contribution consistently equals or exceeds 80% of the total runtime, indicating that the ILP solving step finished in sub-millisecond time and the entire measured total is attributable to DFA analysis overhead.

Figure 7.14 reveals a striking contrast between approaches. The ILP curve on the logarithmic scale remains within the 10^0–10^1 ms band throughout the entire range, while the SAT baseline spans 10^0–10^2 ms with increasing spread at higher condition counts. The ILP approach is therefore up to two orders of magnitude faster than the SAT baseline at 140 `withLabel` conditions, especially when exploring the isolated execution times of the two approaches.

Discussion. The flat runtime profile of this sub-dimension follows directly from the semantics of `withLabel`: these conditions filter which data elements are subject to a `neverFlows` constraint, but they do not change which architectural modifications can resolve the resulting violations. Outside the context of our test environment, it is reasonable to expect a decrease in execution time for the solving-only process, given our approach of reducing the number of violations by introducing new required data without introducing new mitigation measures. This results in a reduction of the problem space, leading to a decrease in execution time. The SAT baseline, in contrast, must encode the full constraint satisfaction problem, including the filtering logic in CNF, and additional `withLabel` conditions introduce new Boolean variables that expand the clause set even if they do not correspond to actionable mitigations, which explains the growing SAT runtimes.

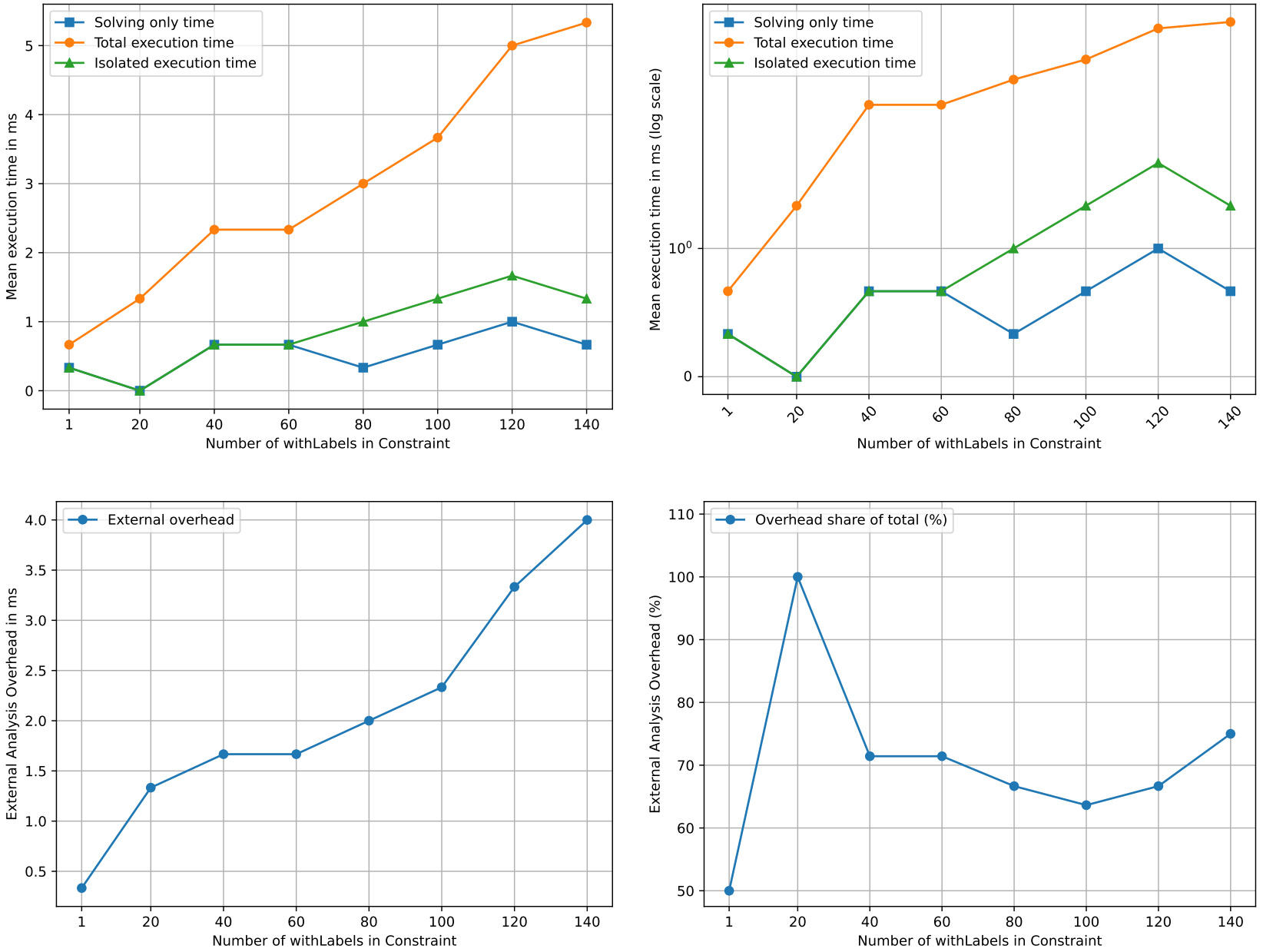

Figure 7.13: Evaluation result for Amount Data With Label

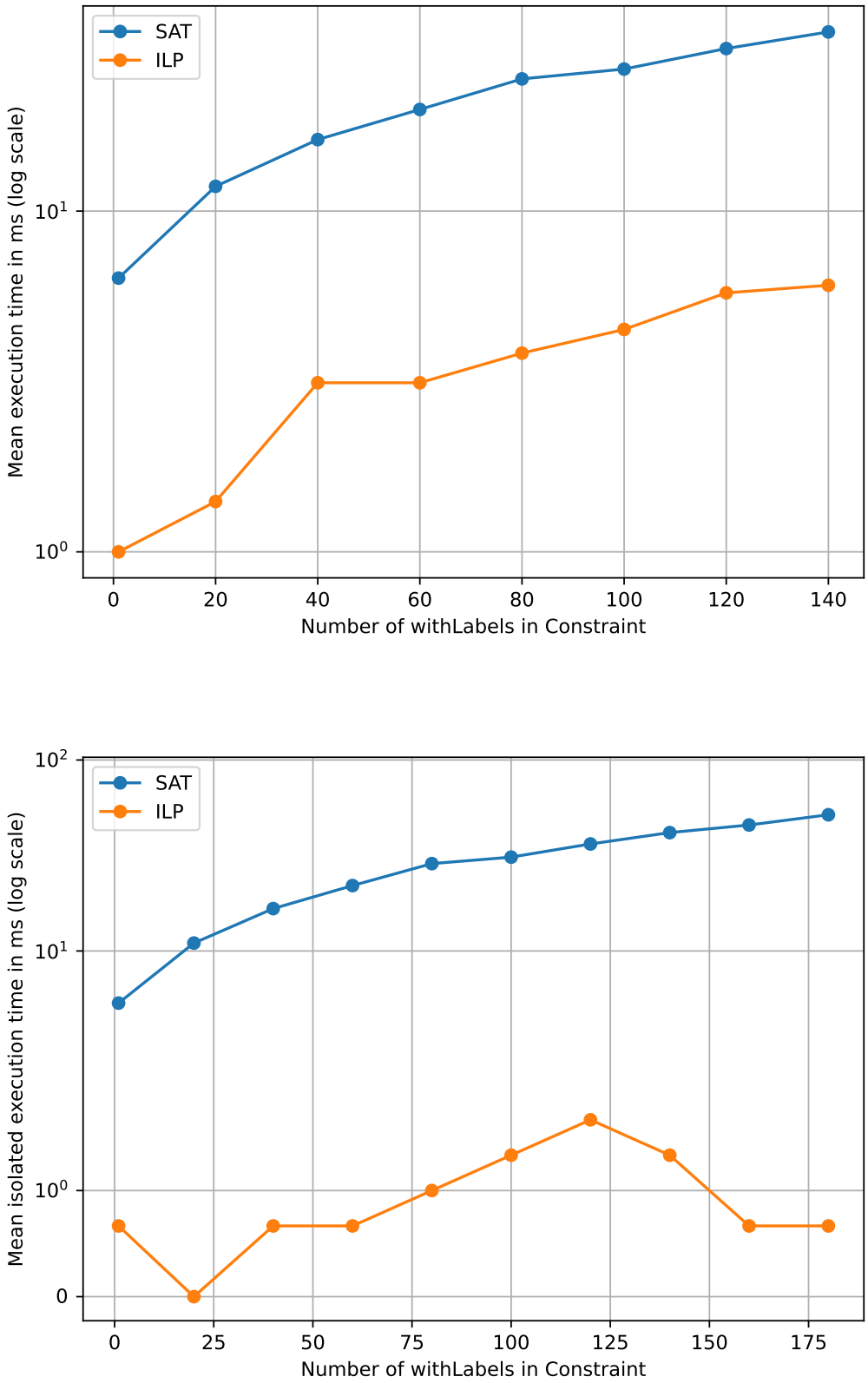

Figure 7.14: Comparison of SAT and ILP approaches for Number of data with Label in Constraints

Data Without Label Figure 7.15 presents a qualitatively different picture. The linear-scale plot shows a clearly visible upward trend from approximately 1 ms to 12 ms as `withoutLabel` conditions grow from 1 to 140. The growth appears approximately linear, with the total and isolated curves tracking each other closely and the solving-only component contributing constant values around <1ms. The absolute overhead stays between 0.5 ms and 3.5 ms, while the overhead-share plot shows a declining trend from roughly 80% at low condition counts to approximately 30% at 140 conditions, confirming that the problem space definition step becomes the dominant contributor as the condition count grows.

Figure 7.16 shows that the ILP approach remains in the 10^1 ms range, while the SAT baseline spans 10^0–10^2 ms, both with a rising trend. The ILP is therefore approximately one order of magnitude faster than the SAT baseline at high condition counts, but both seem to grow at the same speed.

Discussion. The linear growth in this sub-dimension is a direct consequence of the candidate generation logic. Each `withoutLabel` condition specifies a label ℓ that must be present on data to comply with the constraint. For every violation caused by the absence of ℓ, the Problem Space Exploration phase generates one new label-addition candidate per incoming data flow along the violating TFG, if ℓ can be added at any point on the path. If the path has F incoming Flows, each additional `withoutLabel` condition therefore contributes exactly F new binary decision variables to the ILP. With w `withoutLabel` conditions and a fixed number of incoming flows F, the number of variables grows as $O(w \cdot F)$, which is linear in w for fixed F. The declining DFA overhead share further confirms this: the DFA cost is fixed (the analysis does not depend on the number of conditions, only on the TFG structure), while ILP solving time grows proportionally to the candidate count. SAT seems to have the same complexity for a growing number of `withoutLabel` as ILP, but starts with a higher base execution time.

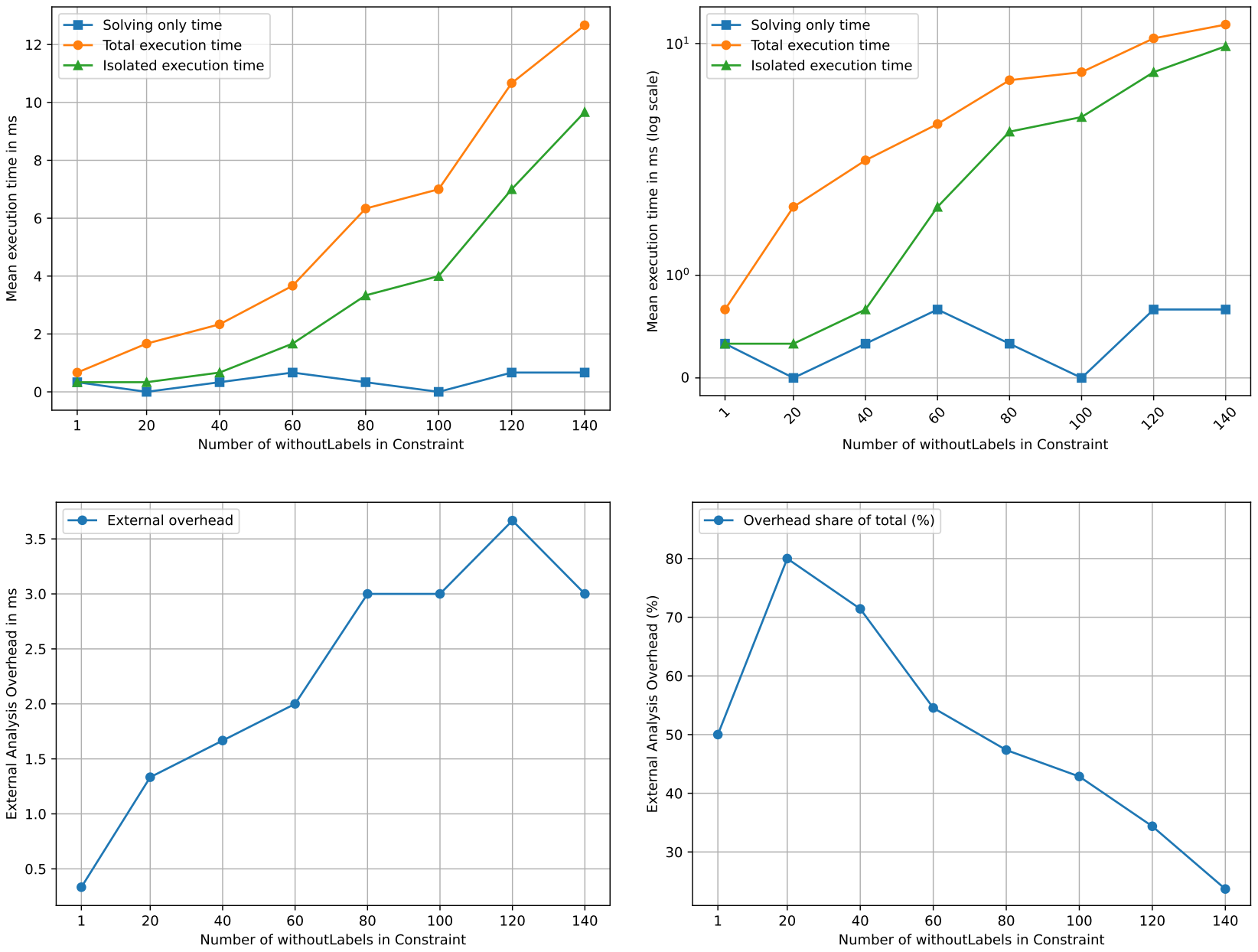

Figure 7.15: Evaluation result for Amount Data Without Label

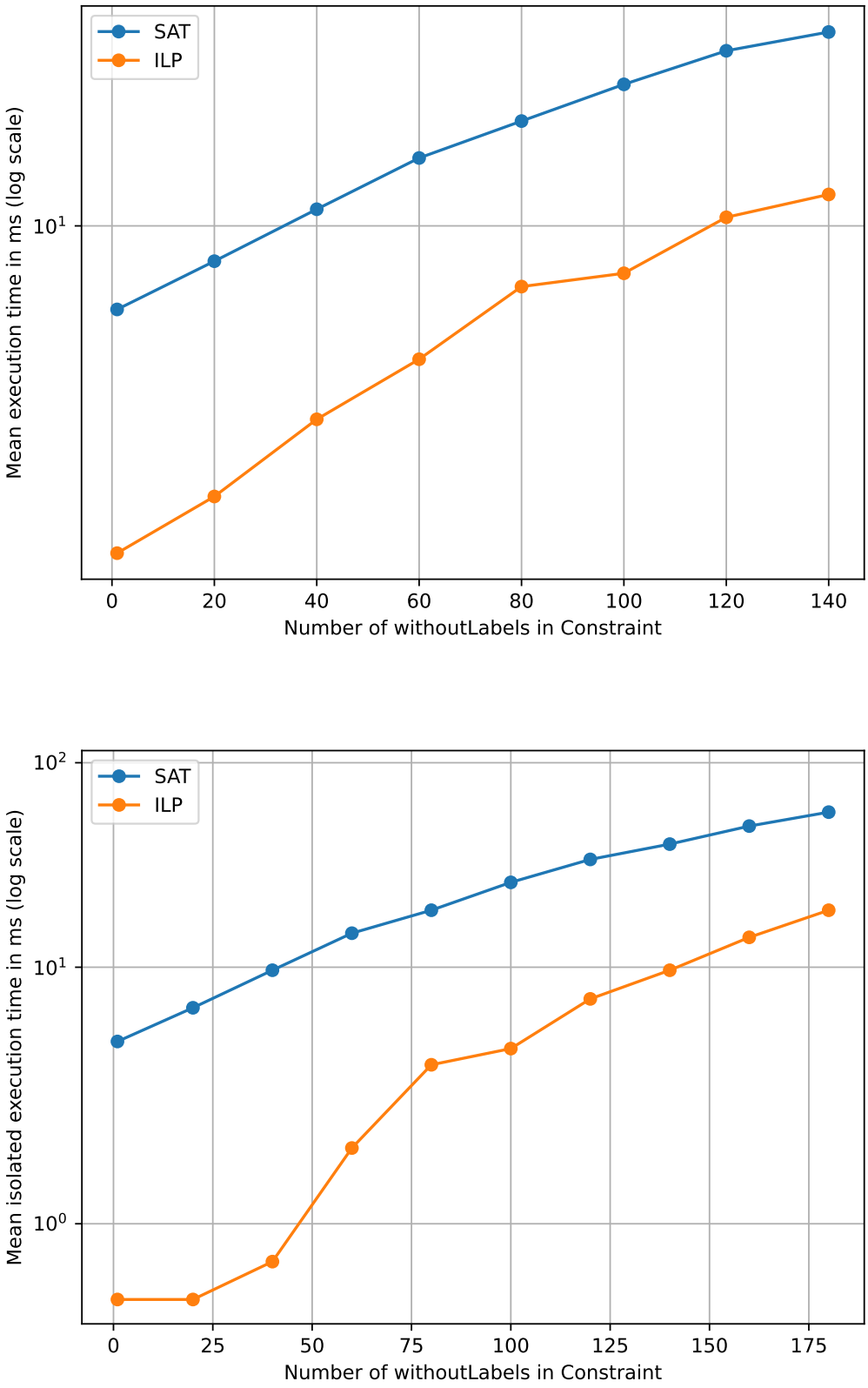

Figure 7.16: Comparison of SAT and ILP approaches for Number of data without Label in Constraints

Nodes With Characteristic Figure 7.17 mirrors the behaviour observed for `withLabel`. The linear-scale plot shows that total execution time fluctuates between 0.5 ms and 4.5 ms across the entire range of 1 to 140 conditions, with a linear upward trend. The absolute overhead is equally flat at 0.5 ms–4.0 ms, and the overhead-share plot shows values consistently between 50% and 90%, with the share rising toward 100% at the highest condition counts. Noteworthy is that the solving-only curve is practically indistinguishable from zero on both the linear and logarithmic plots, confirming that the ILP solver adds essentially zero overhead for this sub-dimension.

Figure 7.18 shows both approaches in the 10^0–10^2 ms band on the total-time comparison plot, with a constant runtime advantage for the ILP approach. The isolated-time comparison is more diagnostic: the SAT baseline rises to 10^1–10^2 ms for isolated runs, yielding an advantage of approximately one order of magnitude for the ILP approach while being constant around 1 ms.

Discussion. The flat runtime profile is structurally symmetric to the `withLabel` case. The `withCharacteristic` conditions identify nodes that are prohibited from receiving the labelled data; violating nodes are those that *do* possess the specified characteristic. The corresponding repair is to *remove* the offending characteristic from the node. Since only k new binary variables enter the ILP, the solver finishes in negligible time. The rise of the overhead share toward 100% at higher condition counts quantifies this precisely: with more `withCharacteristic` conditions active, the DFA analysis must evaluate more conditions per node, slightly increasing its absolute cost, but the ILP remains nearly constant, so the relative overhead share increases monotonically.

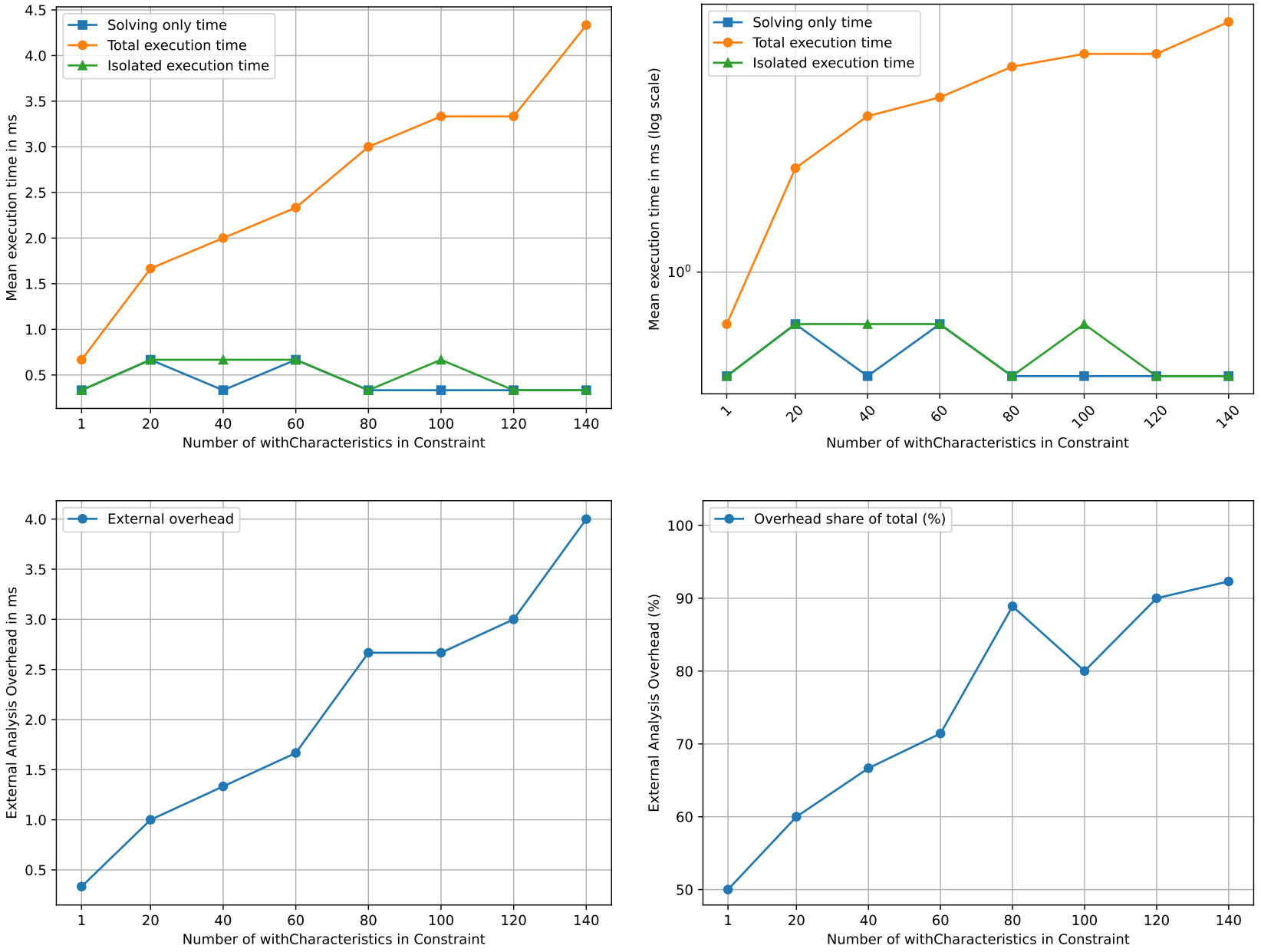

Figure 7.17: Evaluation result for Amount nodes With Characteristic

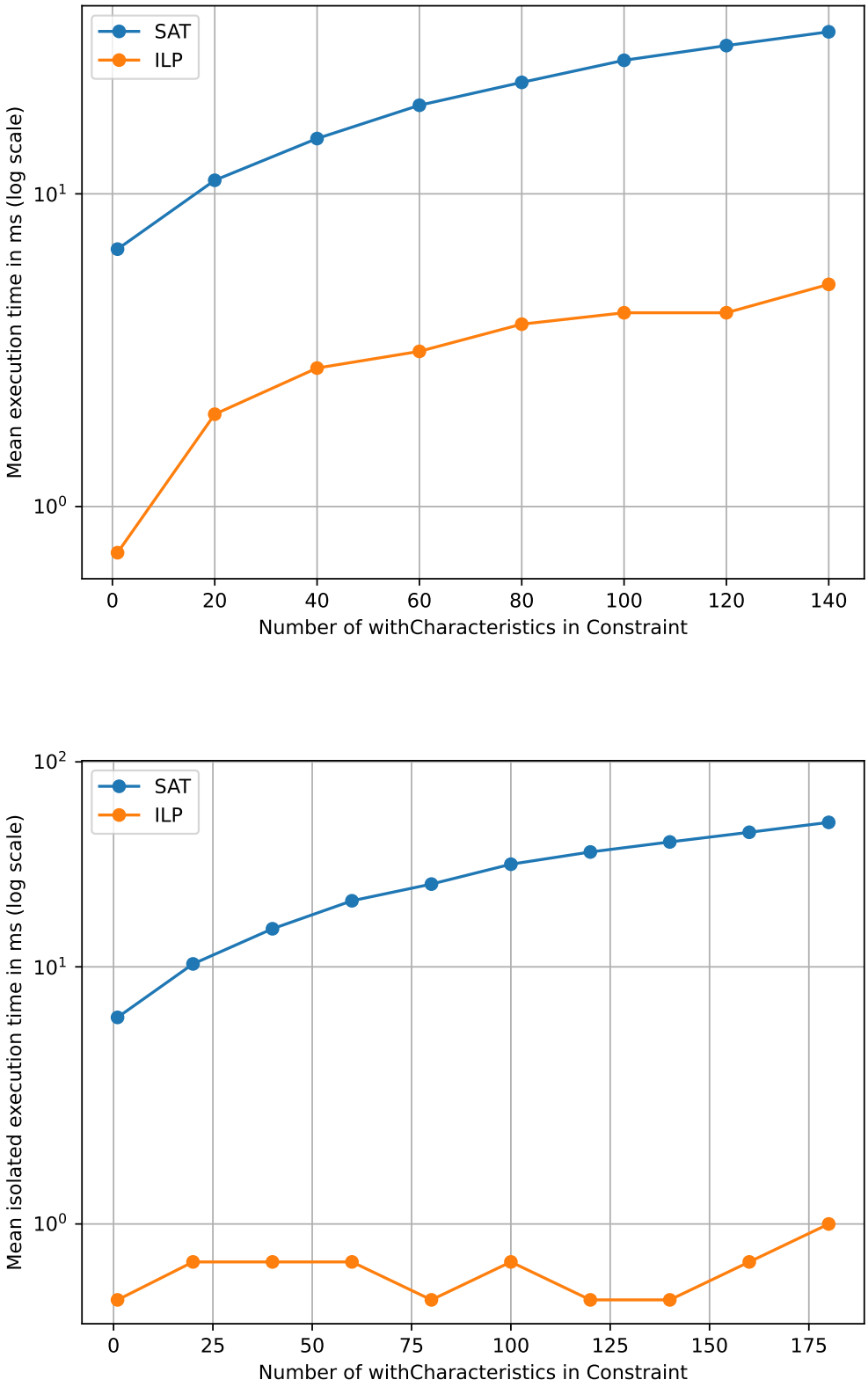

Figure 7.18: Comparison of SAT and ILP approaches for Number of nodes with Characteristic in Constraints

Nodes Without Characteristic Figure 7.19 shows a consistent polynomial growth from approximately 1 ms to 14 ms as `withoutCharacteristic` conditions increase from 1 to 140. The linear-scale plot reveals that the total and isolated curves rise in parallel, with the solving-only curve being constant. The absolute overhead stabilises between 0.5 ms and 4 ms, while the overhead-share plot decreases from approximately 80% at low condition counts to around 30% at 140, mirroring the pattern seen for `withoutLabel`. The structural parallel between these two negative-condition sub-dimensions is particularly informative. Both sub-dimensions exhibit an increase in value from approximately ∼1 ms to ∼12–14 ms , accompanied by a decline in overhead share. This parallel is discussed in more detail below.

Figure 7.20 places both approaches in the 10^0–10^1 ms band throughout the tested range, with the ILP approach remaining at the lower end. The isolated comparison is consistent with the withoutLabel case, the SAT baseline seems to grow at the same speed as the ILP approach, with a slight gain on the ILP end.

Discussion. The linear growth in this sub-dimension follows the same mechanism as `withoutLabel`, applied to the node-characteristic domain. A `withoutCharacteristic` condition specifies a characteristic c whose presence on a node would make that node compliant. For each violation, the Problem Space Exploration phase generates one characteristic-addition candidate per violating node. With u `withoutCharacteristic` conditions and violating Nodes N, the number of new variables is again $O(u \cdot N)$, producing the same linear growth rate as observed for `withoutLabel`. The symmetric behaviour of the two negative-condition sub-dimensions establishes a general principle: *negative conditions drive mitigation candidate space growth, while positive conditions drive/reduce violation identification.* This distinction is not an artefact of the experimental setup but reflects an inherent structural property of the ILP formulation and the DSL.

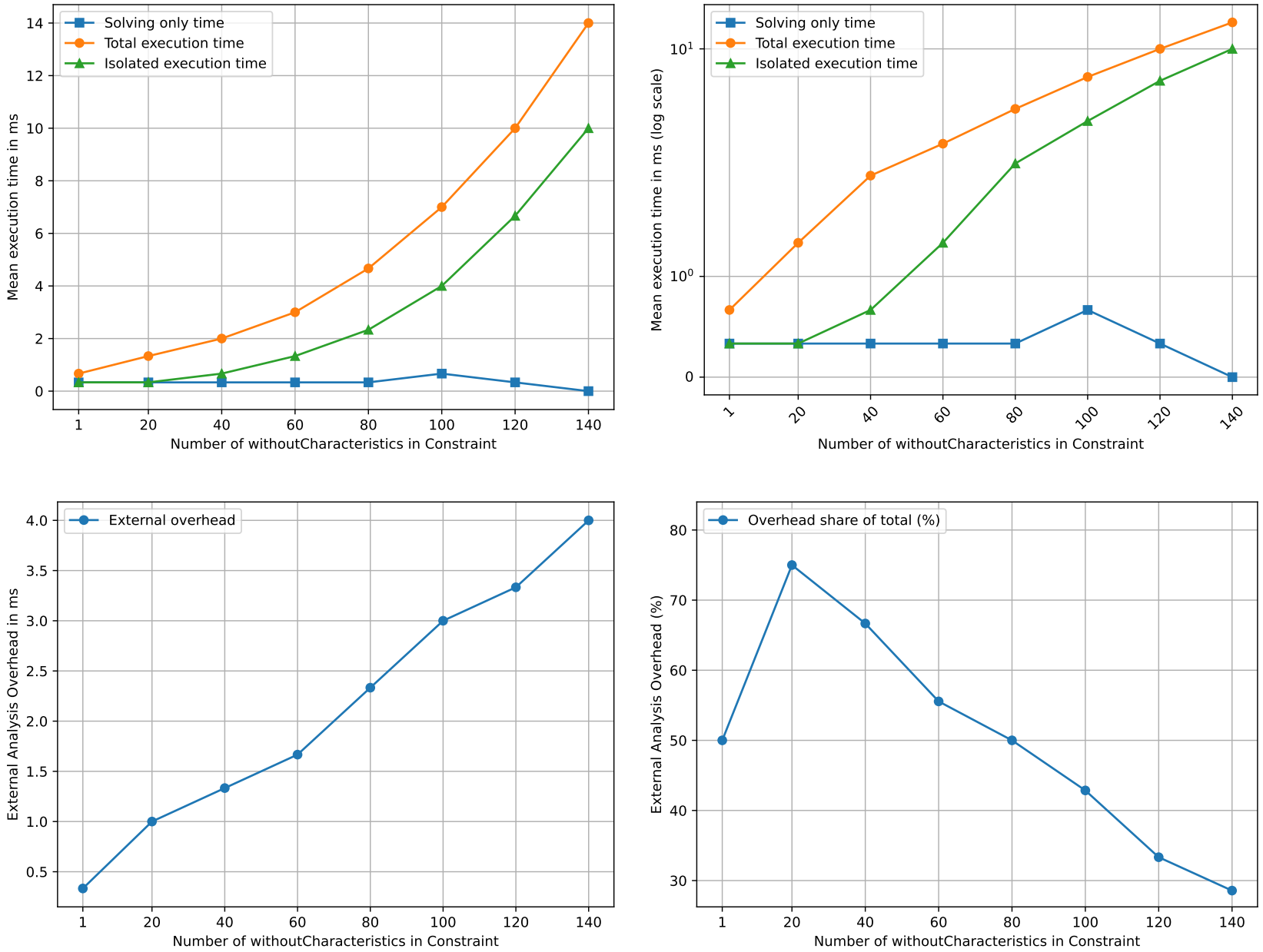

Figure 7.19: Evaluation result for Amount nodes Without Characteristic

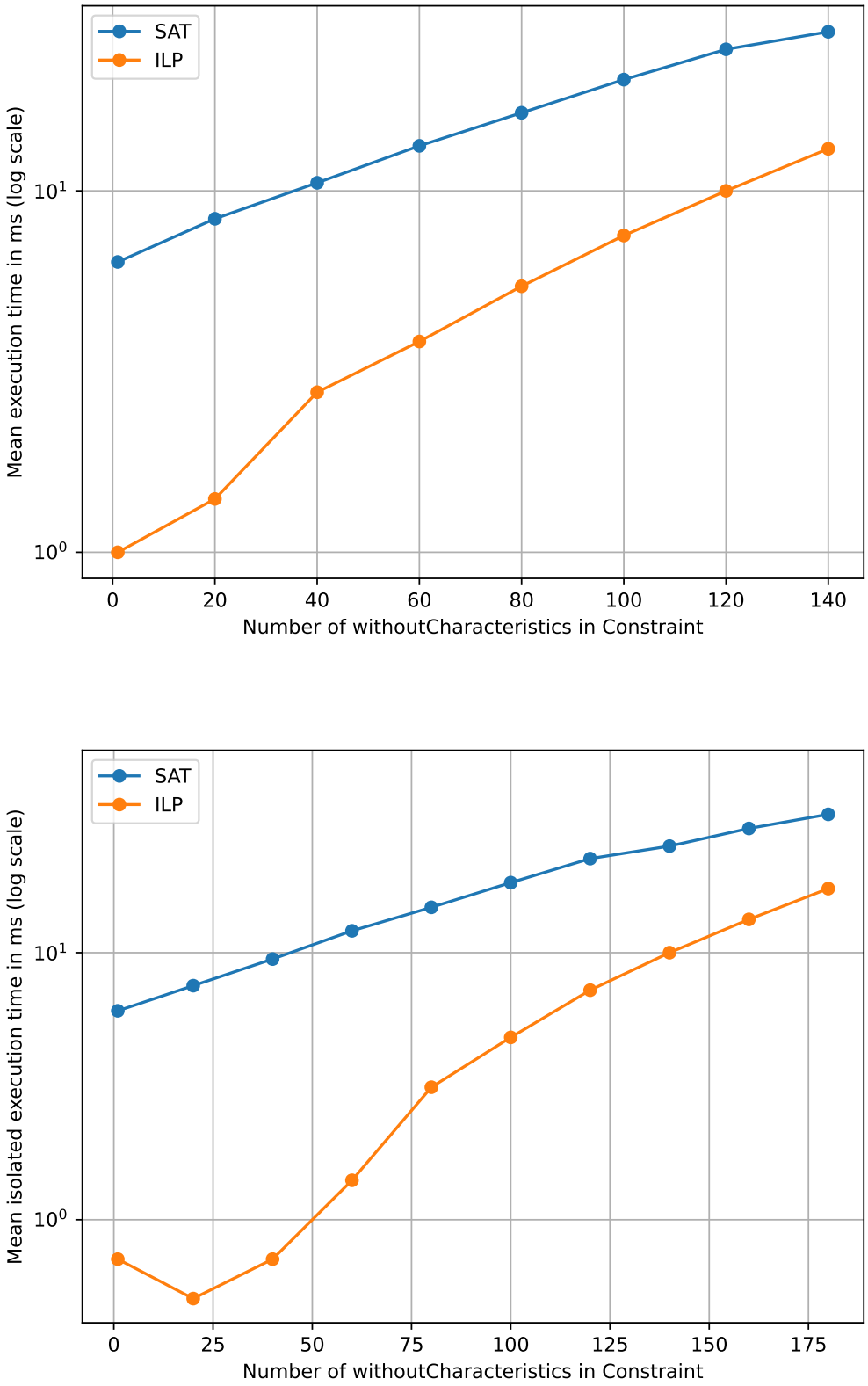

Figure 7.20: Comparison of SAT and ILP approaches for Number of nodes without Characteristic in Constraints

Findings on Q4.5 – All Constraint Dimensions at Once. Finally, Figure 7.21 shows the ILP runtime when all constraint-related dimensions are scaled together from 1 to 140. The linear-scale plot is the most dramatic in the entire evaluation: total execution time rises steeply from near 0 ms at dimension value 1 to approximately 1.2×10^7 ms at 140, corresponding to roughly 3.3 hours per instance. The growth is clearly super-linear: the curve is polynomial on the linear scale and appears as an approximately straight line with a steep slope on the logarithmic scale, suggesting exponential growth in the range of $O(n^2)$ to $O(n^3)$ or faster. The growth mostly stems from the isolated execution, since it tracks the total curve through the entire graph, while the solving-only component seems to be constant. The absolute overhead plot confirms this transition: the DFA analysis cost grows to approximately 800 ms at dimension 140, which is negligible relative to the 10^7 ms ILP solving cost at that point. The overhead-share plot shows the DFA contribution declining from approximately 50% at low dimension values to below 10% at the end. This marks the point at which the ILP solver, rather than the upstream analysis, becomes the bottleneck.

Figure 7.22 compares both approaches for dimension values 1 to 20. In this lower range, ILP seem to scale in a flat line, indicating low exponential growth. SAT, on the other hand, has a very steep/exponential curve indicating super-exponential growth, exceeding measurable results past a scaling of 6.

Discussion. The super-linear growth in this scenario is the direct result of the multiplicative interaction between all five growth mechanisms active simultaneously. For a combined dimension value of d, there are d `withoutLabel` conditions, d `withoutCharacteristic` conditions, and d constraints. The number of ILP variables grows as $O(d^2 \cdot L)$: each constraint contributes $O(2d)$ new label-addition and characteristic-addition candidates per node and incoming Flow, and there are $O(d)$ constraints each producing violations. The number of pairwise contradiction constraints between candidates grows quadratically in the variable count, yielding $O(d^4 \cdot L^2)$ potential constraint pairs. While many of these are pruned during the preprocessing phase of the ILP solver, the dense interaction forces the problem definition algorithm to evaluate significantly more candidate pairs than in any single-dimension experiment, which produces the observed exponential runtime curve. This behaviour is a direct consequence of the NP-hardness of the underlying weighted set cover formulation: when violations, candidates, and inter-candidate relations all grow together, the ILP instance transitions from a well-structured, easily solvable problem to a combinatorially hard one. For practical use, this means that architects should avoid models in which all constraint dimensions simultaneously exceed 40. In practice, however, this situation does not arise, since common microservice security practices described by Schneider et al. [43] can typically be specified with constraints of length 4 rather than 140, as demonstrated by Niehues et al. [34].

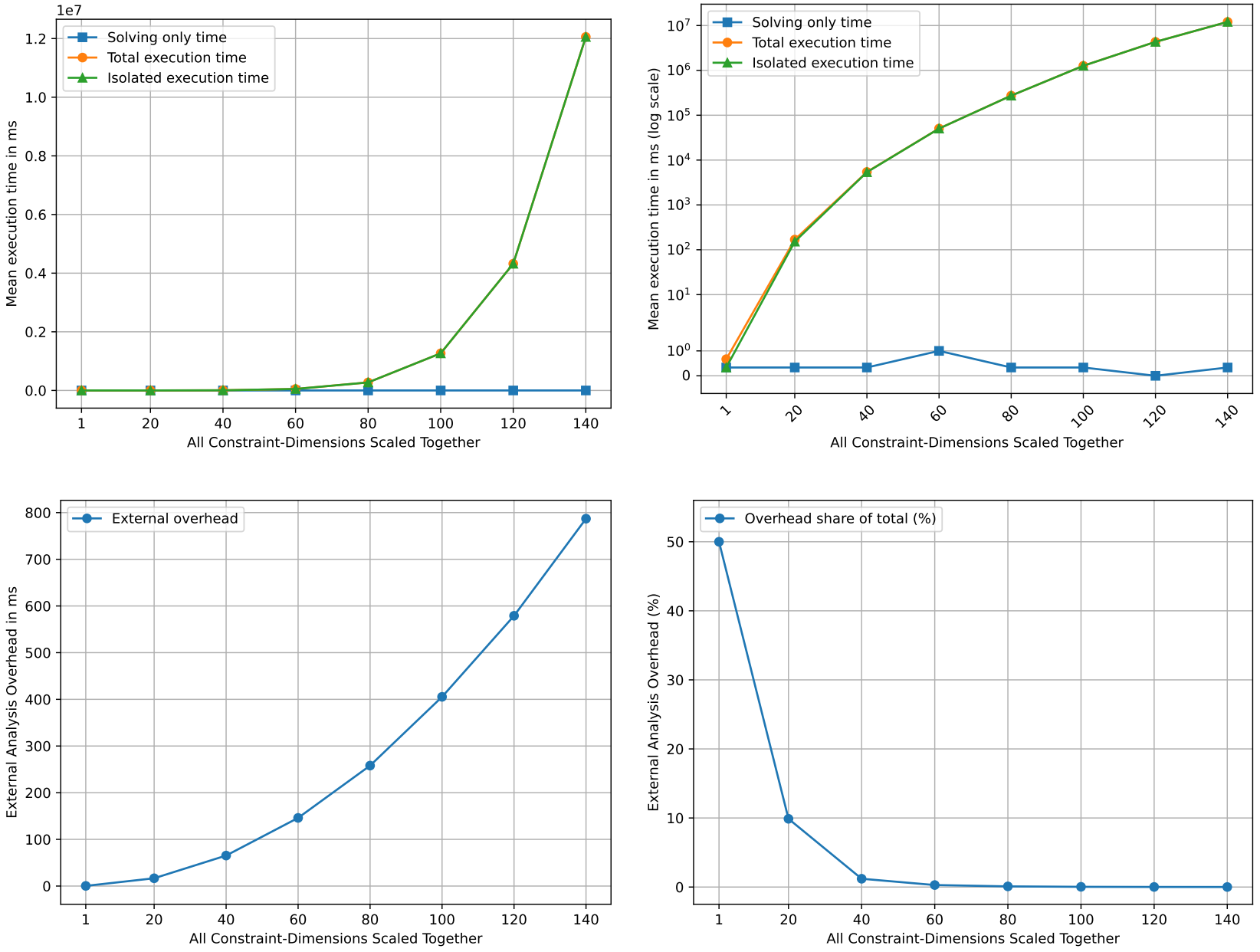

Figure 7.21: Evaluation result for All Constraint dimensions

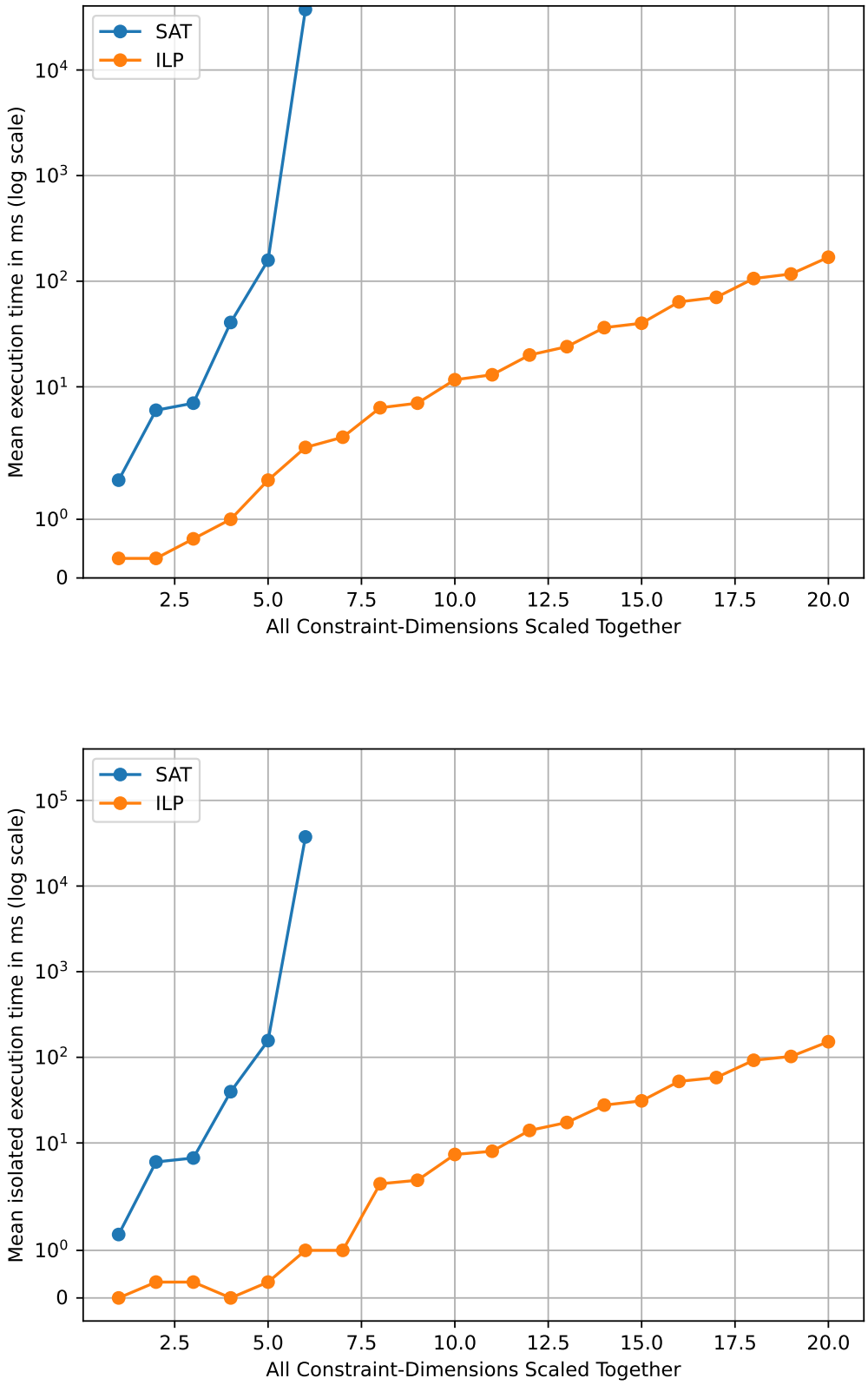

Figure 7.22: Comparison of SAT and ILP approaches for all Constraint Dimensions

Cross-Dimensional Discussion. Considered across all eight scaling dimensions, the results reveal two orthogonal structural axes that fully explain the observed runtime behaviour of the ILP-based approach.

The first axis separates *positive* from *negative* constraint conditions. Positive conditions (`withLabel`, `withCharacteristic`) narrow the set of TFGs or nodes that qualify for violations. Negative conditions (`withoutLabel`, `withoutCharacteristic`) instead specify properties whose *addition* resolves a violation, and the Problem Space Exploration phase generates $O(w \cdot F)$ and $O(u \cdot N)$ new variables for each, respectively. The empirical result is equally clean: the two positive sub-dimensions yield flat or negligible runtime curves, while the two negative sub-dimensions yield consistent near-linear growth, establishing the principle that *negative conditions drive candidate space growth while positive conditions scope violation identification.*

The second axis separates *DFA-dominated* from *ILP-dominated* dimensions. For the number of TFGs, the DFA analysis accounts for 40%–100% of total runtime, because the shared-variable structure of the ILP prevents solver cost from scaling with TFG count. For TFG length and the negative-condition sub-dimensions, both DFA and ILP components grow linearly, but with different constants, producing the observed gradual shift of the overhead-share trend. For the combined-dimension experiment, the ILP solver becomes the dominant bottleneck once the multiplicative interaction of d constraints, d `withoutLabel` conditions, and d `withoutCharacteristic` conditions drives the variable count to $O(d^2 \cdot L)$ and the contradiction set to $O(d^4 \cdot L^2)$, at which point the DFA overhead of approximately 800 ms at $d = 140$ is negligible relative to the 10^7 ms ILP solving cost.

Across all dimensions, the ILP approach consistently outperforms the SAT baseline. In scenarios where the ILP solver does not represent a bottleneck, such as in the TFG amount and positive conditions, the advantage is attributed to the shared-variable formulation and the violation-centric problem space. These elements prevent the undesirable growth in problem size. In scenarios where the ILP solver is the primary bottleneck, characterised by TFG length, negative conditions, and their combination, the superiority of the ILP relaxation over SAT becomes evident. The ILP relaxation enables the branch-and-cut algorithm to efficiently prune infeasible regions through the utilisation of explicit linear inequalities, thereby circumventing the exhaustive exploration of exponentially many clause combinations that would otherwise be necessitated by backtracking. The constraint-count experiment provides the strongest evidence: the SAT baseline becomes infeasible beyond 17 constraints, while the ILP remains flat in the 10^0–10^1 ms range throughout the same scaling. This confirms that the ILP formulation not only scales better in the average case but also avoids the hard-instance pathology that is structurally inherent to the CNF encoding used by the SAT baseline.

7.4 Threats to Validity

We discuss the threats to the validity of our evaluation. Following the scheme proposed by Runeson et al. [39], we address internal, external, and construct validity as well as reliability separately. Validity in this context refers to the trustworthiness of the results and the degree to which they reflect the investigated phenomena rather than the researchers' subjective decisions [39].

Internal Validity Threats to internal validity concern whether the observed evaluation results depend solely on the investigated factors, without interference from uncontrolled third variables.

One such threat arises in the cost evaluation (G3). The invasiveness metric is defined by the authors and assigns fixed weights to different types of architectural modifications. Because the same authors who defined the cost model also designed the mitigation strategies and evaluated the outcomes against this metric, there remains a risk of confirmation bias in how mitigation quality is assessed. To reduce this threat, the evaluation in this thesis normalises all atomic mitigation steps to a uniform unit cost and compares approaches purely by counting the number of applied changes, rather than by assigning differentiated costs. This avoids favouring particular strategy types through subjective weighting, but it does not fully eliminate bias, as the same authors still chose the space of available strategies and the decision to treat all atomic changes as equally costly.

A further threat concerns the scalability evaluation (G4). The synthetic DFD models used in the scalability experiments are generated by systematically varying a single dimension at a time while holding all others constant. The generation procedure was designed by the author, which introduces the risk that the generated structures inadvertently favour the ILP formulation. However, since the SAT baseline is evaluated on the same synthetic instances under identical conditions, the relative performance comparison between both approaches remains meaningful even if the absolute runtime figures are specific to the generated model class.

Finally, for the effectiveness evaluation (G1), both the constraints and the model variants were taken from the publicly available MicroSecEnD dataset, which was not created by the authors of this thesis. This reduces the risk that the scenarios were selected to match the approach's capabilities, and thus constitutes a strong safeguard against internal validity threats in this goal.

External Validity Threats to external validity impact the generalizability of the evaluation results beyond the studied scenarios.

The most significant threat arises from the exclusive use of the MicroSecEnD dataset for effectiveness (G1) and cost (G3) evaluation. This dataset comprises 17 DFD models of real-world, open-source microservice applications. While these models are grounded in real implementations, they represent a specific class of architectures. Results may not generalise directly to monolithic systems, embedded systems, or architectures with substantially different connectivity patterns, constraint profiles, or data sensitivity requirements. Strictly speaking, the effectiveness and cost results are valid only for the studied scenarios and should be interpreted within this scope, but the large variation in violation counts between these architectures also indicates that they cover substantially different structural situations.

A related threat concerns the scalability evaluation (G4). The synthetic models are designed to isolate individual scaling dimensions, and their parameter ranges were chosen to be practically relevant. However, real-world architectures may exhibit simultaneous growth across multiple dimensions in ways that the isolated scaling experiments do not fully capture. The combined-dimension experiment (Q4.5) partially addresses this by scaling all dimensions jointly, but the specific interaction structure of real systems remains untested.

Construct Validity Threats to construct validity concern whether the selected questions and metrics appropriately operationalise the overarching evaluation goals.

For the cost goal (G3), the chosen metric (the sum of modification counts) is a proxy for the genuine engineering concept of invasiveness. While it captures the number and type of changes, it cannot fully account for semantic context: removing a flow may be architecturally trivial in one system and critical in another. A user study involving software architects assessing the acceptability of repair proposals would provide stronger construct validity, but was outside the scope of this thesis. The provision of a configurable cost model partially compensates for this gap.

For the extensibility goal (G2), the evaluation is purely argumentative rather than quantitative. We demonstrate that label-addition alone is inherently insufficient for a class of structural violations by logical reasoning over the monotonicity properties of label propagation. While this argument is logically complete, it does not empirically quantify the frequency with which such violations occur in practice, nor does it measure the additional runtime overhead introduced by structural strategies. This constitutes a limitation of the construct validity for (G2).

The application of the Goal-Question-Metric approach [5] to structure all four evaluation goals mitigates construct validity threats by ensuring that each metric is explicitly linked to a stated evaluation purpose and viewpoint.

Reliability Reliability concerns the repeatability of the evaluation and the degree to which results are independent of the specific researcher conducting it [39]. In order to facilitate reproducibility, the evaluation design is fully specified in Section 7.1 and 7.2 prior to the presentation of results, including all metrics, constraint definitions and model setups.

The underlying ILP solver operates deterministically: given identical inputs, it produces identical outputs. This ensures that all optimisation results reported in Section 7.3 are fully reproducible. The implementation, along with all DFD models, constraint definitions, and scalability test cases used in this evaluation, is publicly available as described in Section 7.6.

7.5 Assumptions and Limitations

In this section, we discuss the assumptions made during the design and realisation of the proposed mitigation approach, together with the limitations of its current implementation.

Inherited Assumptions from DFA Cycle Resolution The proposed approach does not itself impose an acyclicity constraint on the input DFD. The underlying DFA framework supports cyclic architectures through a dedicated cycle-resolution preprocessing step, as described in Arp et al. [2]. However, because the mitigation approach relies on this external cycle-resolution mechanism as a direct upstream dependency, the assumptions and approximations introduced therein transitively apply to the correctness of the resulting mitigations. Any semantics that are altered or abstracted away during cycle resolution may cause the resulting TFGs to represent a subtly different system than intended. The validity of the computed mitigations is therefore contingent not only on the correctness of the ILP formulation, but also on the fidelity of the upstream cycle-resolution step.

Boundary of Automated Strategy Derivation Automated derivation of mitigation strategies is restricted to constraints expressible in the `neverFlows`-style DSL. Constraints that exceed this expressive boundary (for example, cross-TFG reachability conditions) require the user to supply all applicable mitigation strategies manually. This creates a partial automation boundary: the degree to which the tool operates without human input is directly coupled to the expressiveness of the DSL. Any real-world constraint that falls outside this boundary converts the tool from a fully automated system into a semi-automated one, placing additional expertise demands on the user. This limitation is inherent to any DSL-driven approach and could be addressed in future work by extending the DSL's expressiveness or by developing heuristic strategy inference for a wider class of constraint patterns.

Assumption of Strategy Completeness The automated mitigation approach guarantees the elimination of all detected violations if and only if, for every violation identified by the DFA phase, at least one applicable mitigation strategy has been provided by the user or derived automatically from the constraint analysis. If this precondition is not met, for instance, because no strategy exists for a given violation, the ILP problem becomes infeasible for that violation and the solver will report no solution.

Assumption of Non-Deletability of Data Characterisations A more specific assumption governs the scope of automatically derived deletion strategies. The approach treats data characterisations: labels that describe the semantic properties of data elements, such as `personal` or `encrypted`, as representing core domain information that must not be modified by automated repair. Consequently, the automated strategy derivation only generates deletion strategies targeting *node characteristics*, never *data characterisations*. This assumption is well-motivated: removing a data label such as `personal` would alter the semantic identity of the data element rather than addressing an architectural oversight, and could silently obscure a genuine compliance violation rather than resolving it.

Nevertheless, this assumption constrains the automated strategy space. Scenarios in which reclassifying a data element, for instance, removing an incorrectly assigned sensitivity label, would constitute a semantically valid and less invasive repair are excluded from automated consideration. Future work could relax this assumption by introducing a per-constraint annotation that explicitly designates which data characterisations are eligible for automated removal, thereby extending coverage while preserving semantic safety.

Manual Specification of Node and Flow Strategies Structural strategies, including the addition of new nodes, the insertion of sink nodes, and the removal of existing nodes or flows, cannot be inferred from DSL constraints and must be specified manually by the user in every case. The ILP solver selects the most cost-effective subset of the provided strategies, but it cannot autonomously generate strategies that were not supplied. The completeness of the repair space explored during optimisation is therefore bounded by the completeness of the user-provided strategy pool. If a structurally necessary mitigation is absent from the supplied set, the solver is unable to discover it, and the resulting solution may be suboptimal or, in the worst case, unable to eliminate all violations.

Performance Assumption: Contradiction-Free Constraint Sets The current implementation performs exhaustive pairwise contradiction detection during the Problem Space Definition phase, ensuring that all conflicts between candidate mitigations are encoded in the ILP. The effectiveness evaluation demonstrates that this is necessary but sufficient to guarantee complete violation elimination in a single solver run across all studied scenarios. However, the pairwise nature of contradiction detection introduces a growth in the number of contradiction pairs that, in the worst case, scales super-linearly with the number of candidate mitigations. The scalability results show that this becomes the dominant cost driver when all constraint dimensions are scaled jointly, with runtime reaching approximately 3.3 hours at dimension value 140.

A direction for future work arises from this observation. If a given constraint set can be assumed or formally verified to contain no contradictions, the contradiction detection phase can be bypassed entirely for those cases. This would yield significant performance improvements, specifically in the configurations where the current approach exhibits its most severe runtime growth. Developing a pre-verification mechanism, or providing a sound approximation thereof, constitutes a promising avenue to extend the practical scalability of the approach.

7.6 Data Availability

All artefacts produced as part of this thesis are made publicly available in the accompanying replication package [1]. The package includes the mitigation approach implementation, the DFA framework, all test models, and the visualisation code used to produce the evaluation results.

8 Conclusion

To conclude this thesis, we summarise our approach and findings in Section 8.1. We give an outlook on future work in Section 8.2.

8.1 Summary

The goal of this thesis was to answer two questions: which discrete optimisation method is best suited for automated confidentiality repair in DFD-based software architectures, and whether such a method can be realised as a practical, extensible, and cost-aware framework. Both questions can now be answered concretely.

To answer the first question, we conducted a systematic survey of candidate discrete optimisation methods. We evaluated ILP, B&B, and evolutionary algorithms against four criteria: optimality, runtime performance, extensibility, and reproducibility. The survey demonstrated that ILP uniquely satisfies all four criteria simultaneously. It provides provable optimality guarantees, scales efficiently in practice through branch-and-cut solvers, supports declarative extensibility by allowing new constraints and variables to be added without modifying the solver, and produces fully deterministic results.

To answer the second question, we developed a concrete ILP-based mitigation framework integrated into the ARCoViA toolchain. The framework extends the existing DFA-based analysis pipeline with a five-stage repair loop, encoding all candidate mitigations as a binary ILP weighted set cover problem. Unlike prior work, which is restricted to additive label changes, the framework supports the full range of structural modifications: label additions and deletions, node insertions and removals, and flow deletions. Custom user-defined strategies and costs are supported without modifying the solver, as a direct consequence of ILP's declarative problem formulation.

The implementation confirmed the survey result empirically, but also revealed where the complexity actually concentrates. The bottleneck is not the ILP solver (solving times are modest across all studied dimensions), but the Problem Space Exploration phase, which generates the variables and contradiction constraints the solver operates on. When all constraint dimensions are scaled jointly, the quadratic growth of contradiction pairs dominates runtime, reaching approximately 3.3 hours at a scaling of 140. This is the primary scalability ceiling of the current approach, and it is a structural property of exhaustive pairwise contradiction detection, not of ILP itself.

The cost evaluation produced the most significant finding. Across 51 model-constraint pairs from the MicroSecEnD benchmark, the ILP approach accumulated a total repair cost of 271 compared to 1,005 for the human baseline, a 73% reduction that is both statistically significant and large in effect size. A further 66% reduction over the SAT-based baseline demonstrates that encoding repair as a cost-minimisation problem, rather than a satisfiability instance, allows the solver to exploit shared mitigations across violations. These are repairs that a human architect, or a SAT solver operating violation by violation, would not naturally find. One limitation bounds these conclusions: the correctness of repairs in cyclic architectures depends on the cycle-resolution step of the upstream DFA framework.

The results establish that automated, cost-minimal repair of confidentiality violations in DFD-based architectures is tractable with ILP, and that the approach closes a concrete gap of prior work. The proposed approach is the first to combine fully automated, optimal, and structurally flexible mitigation with practical scalability for realistic DFD models. By embedding the tool into the ARCoViA framework, it is directly accessible to architects working within the existing DFD-based analysis ecosystem, reducing the overhead of translating identified violations into concrete model corrections.

8.2 Future Work

The approach presented in this thesis is ready to be used in its current form. However, during its design, realisation, and evaluation, we identified several directions in which it can be extended and improved.

Extending DSL Expressiveness. Automated strategy derivation is currently limited to constraints expressible in the `neverFlows`-style DSL. Constraints that extend beyond this pattern, such as cross-TFG reachability conditions or constraints that reason simultaneously about multiple data flows, fall outside the automated derivation scope and require manual strategy specification. Extending the DSL to support a richer class of constraint patterns and developing corresponding strategy inference rules would reduce the manual effort required for complex constraint profiles and increase the overall degree of automation.

Automated Derivation of Structural Strategies. Node-level and flow-level strategies, like the insertion of new architectural components and the removal of existing flows, currently cannot be inferred from DSL constraints and must always be provided by the user. Future work could explore lightweight static analysis or pattern-based heuristics to automatically suggest structural strategies from the constraint context, for example, by recognising that a constraint requiring an encryption component implies the potential addition of a node with a corresponding stereotype.

Relaxing the Non-Deletability Assumption for Data Characterisations. The current approach treats all data characterisations as non-deletable, generating deletion strategies exclusively for node characteristics. While this assumption is well-motivated by semantic safety, it excludes scenarios in which removing an erroneously assigned data label would constitute a valid and less invasive repair. Future work could introduce a per-constraint annotation mechanism that allows the constraint author to explicitly designate certain data characterisations as eligible for automated removal. This would extend the automated repair space while preserving the semantic integrity of core domain labels.

Addressing Super-Linear Scalability in Combined Dimensions. The scalability evaluation revealed that the runtime grows super-linearly when all constraint dimensions are scaled jointly. This is primarily driven by the quadratic growth of contradiction pairs during the Problem Space Exploration phase. A promising direction for future work is the development of a lightweight consistency pre-check that verifies whether a given constraint set is contradiction-free before entering the exploration phase. For constraint sets that pass this check, the contradiction detection step could be skipped entirely, yielding significant performance improvements precisely in the configurations where the current approach exhibits its most severe runtime growth. Alternatively, incremental or lazy contradiction detection strategies could be investigated, generating only the contradiction constraints that are strictly necessary for the current solver invocation.

Relaxing the Static Architecture Assumption. The approach currently operates on a static DFD snapshot and does not model incremental architectural evolution or runtime variability. Extending the framework towards a change-aware or adaptive mitigation mode, in which previously computed repairs are reused or incrementally updated as the architecture evolves, would improve its applicability to continuously developed systems and reduce the overhead of repeated full re-execution.

Empirical Evaluation of Structural Strategy Coverage. The extensibility evaluation (G2) is purely argumentative, demonstrating the insufficiency of label-addition through logical reasoning rather than empirical measurement. Future work should empirically quantify the proportion of real-world constraints that require structural strategies, and measure the impact of structural strategies on both repair quality and runtime. A controlled user study involving software architects evaluating the acceptability and comprehensibility of the generated repair suggestions would further strengthen the construct validity of this evaluation dimension.

Broadening the Evaluation Dataset. The empirical evaluation relies exclusively on the MicroSecEnD dataset, which covers microservice-based web applications. Evaluating the approach on a broader range of DFD model classes, for example on monolithic architectures, embedded systems, and models with domain-specific compliance requirements, would strengthen the generalizability of the effectiveness and scalability results and may reveal constraint profiles not encountered in the current dataset.

Bibliography

[1] Benjamin Arp. *Supplementary Material for "Automated Mitigation of Confidentiality Violations in Software Architectures using Discrete Optimization"*. Zenodo, Mar. 2026. DOI: 10.5281/zenodo.18873400. URL: https://doi.org/10.5281/zenodo.18873400.

[2] Benjamin Arp et al. "Analyzing Cyclic Data Flow Diagrams Regarding Information Security". In: ISSN: 0720-8928. Gesellschaft für Informatik e.V., 2024. URL: https://dl.gi.de/handle/20.500.12116/45545.

[3] Bengt Aspvall and Richard E Stone. "Khachiyan's linear programming algorithm". In: *Journal of Algorithms* 1.1 (Mar. 1980), pp. 1–13. ISSN: 01966774. DOI: 10.1016/0196-6774(80)90002-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0196677480900024.

[4] Thomas Baeck, D. B. Fogel, and Z. Michalewicz, eds. *Handbook of Evolutionary Computation*. Boca Raton: CRC Press, Jan. 1997. ISBN: 978-0-367-80248-6. DOI: 10.1201/9780367802486.

[5] Victor R. Basili, Caldiera Gianluigi, and H. Dieter Rombach. "THE GOAL QUESTION METRIC APPROACH". In: 1994.

[6] A Biere et al. *Handbook of Satisfiability*. Apr. 2021. ISBN: 978-1-64368-160-3. URL: https://www.iospress.com/catalog/books/handbook-of-satisfiability-2 (visited on 02/18/2026).

[7] Nicolas Boltz et al. "An Extensible Framework for Architecture-Based Data Flow Analysis for Information Security". In: *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium*. Ed. by Bedir Tekinerdoğan et al. Cham: Springer Nature Switzerland, 2024, pp. 342–358. ISBN: 978-3-031-66326-0. DOI: 10.1007/978-3-031-66326-0_21.

[8] Quang-Cuong Bui et al. "APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities". In: *Empirical Software Engineering* 29.1 (Jan. 2024), p. 18. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-023-10415-7. URL: https://link.springer.com/10.1007/s10664-023-10415-7.

[9] V. Chvatal. "A Greedy Heuristic for the Set-Covering Problem". In: *Mathematics of Operations Research* 4.3 (Aug. 1979), pp. 233–235. ISSN: 0364-765X. DOI: 10.1287/moor.4.3.233. URL: https://pubsonline.informs.org/doi/10.1287/moor.4.3.233.

[10] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047. URL: https://dl.acm.org/doi/10.1145/800157.805047.

[11] K. Deb. "Multi-objective optimization using evolutionary algorithms". In: 2001. URL: https://www.semanticscholar.org/paper/Multi-objective-optimization-using-evolutionary-Deb/f51afa2745947c11300c65ce001dc6fb2ddd5c2e (visited on 02/04/2026).

[12] Tom DeMarco. "Structured Analysis and System Specification". In: *Software Pioneers: Contributions to Software Engineering*. Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer, 1978, pp. 529–560. ISBN: 978-3-642-59412-0. DOI: 10.1007/978-3-642-59412-0_33. URL: https://doi.org/10.1007/978-3-642-59412-0_33.

[13] Kevin Feichtinger et al. "It's your loss: classifying information loss during variability model roundtrip transformations". In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*. Vol. A. SPLC '22. New York, NY, USA: Association for Computing Machinery, Sept. 2022, pp. 67–78. ISBN: 978-1-4503-9443-7. DOI: 10.1145/3546932.3546990. URL: https://dl.acm.org/doi/10.1145/3546932.3546990.

[14] Vijay Ganesh and Moshe Y. Vardi. "On the Unreasonable Effectiveness of SAT Solvers". In: *Beyond the Worst-Case Analysis of Algorithms*. Ed. by Tim Roughgarden. 1st ed. Cambridge University Press, Dec. 2020, pp. 547–566. ISBN: 978-1-108-63743-5 978-1-108-49431-1. DOI: 10.1017/9781108637435.032. URL: https://www.cambridge.org/core/product/identifier/9781108637435%23c25/type/book_part.

[15] D. Garlan et al. "Rainbow: architecture-based self-adaptation with reusable infrastructure". In: *Computer* 37.10 (Oct. 2004), pp. 46–54. ISSN: 1558-0814. DOI: 10.1109/MC.2004.175. URL: https://ieeexplore.ieee.org/abstract/document/1350726.

[16] Fred Glover. "Tabu Search—Part I | ORSA Journal on Computing". In: 1989. URL: https://pubsonline.informs.org/doi/abs/10.1287/ijoc.1.3.190 (visited on 09/04/2025).

[17] Mark Harman and Bryan F Jones. "Search-based software engineering". In: *Information and Software Technology* 43.14 (Dec. 2001), pp. 833–839. ISSN: 0950-5849. DOI: 10.1016/S0950-5849(01)00189-6. URL: https://www.sciencedirect.com/science/article/pii/S0950584901001896.

[18] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. "Search-based software engineering: Trends, techniques and applications". In: *ACM Comput. Surv.* 45.1 (2012), 11:1–11:61. ISSN: 0360-0300. DOI: 10.1145/2379776.2379787. URL: https://dl.acm.org/doi/10.1145/2379776.2379787.

[19] Jan Jürjens. "UMLsec: Extending UML for Secure Systems Development". In: *UML 2002 — The Unified Modeling Language*. Ed. by Gerhard Goos et al. Vol. 2460. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 412–425. ISBN: 978-3-540-44254-7 978-3-540-45800-5. DOI: 10.1007/3-540-45800-X_32. URL: http://link.springer.com/10.1007/3-540-45800-X_32.

[20] Kyo C. Kang et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". In: (Nov. 1990). Number: CMUSEI90TR21. URL: https://apps.dtic.mil/sti/html/tr/ADA235785/ (visited on 09/14/2025).

[21] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (May 1983). Publisher: American Association for the Advancement of Science, pp. 671–680. DOI: 10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[23] Bernhard Korte and Jens Vygen. "Integer Programming". In: *Combinatorial Optimization: Theory and Algorithms*. Ed. by Bernhard Korte and Jens Vygen. Berlin, Heidelberg: Springer, 2018, pp. 103–132. ISBN: 978-3-662-56039-6. DOI: 10.1007/978-3-662-56039-6_5. URL: https://doi.org/10.1007/978-3-662-56039-6_5.

[24] Anne Koziolek. "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes". Google-Books-ID: PNoiAwAAQBAJ. PhD thesis. KIT Scientific Publishing, Jan. 2014. URL: https://publikationen.bibliothek.kit.edu/1000024955.

[25] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. "PerOpteryx: automated application of tactics in multi-objective software architecture optimization". In: *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. Boulder Colorado USA: ACM, June 2011, pp. 33–42. ISBN: 978-1-4503-0724-6. DOI: 10.1145/2000259.2000267. URL: https://dl.acm.org/doi/10.1145/2000259.2000267.

[26] E. L. Lawler and D. E. Wood. "Branch-and-Bound Methods: A Survey". In: *Oper. Res.* 14.4 (Aug. 1966), pp. 699–719. ISSN: 0030-364X. DOI: 10.1287/opre.14.4.699. URL: https://doi.org/10.1287/opre.14.4.699.

[27] Claire Le Goues. "Automatic Program Repair Using Genetic Programming". PhD thesis. University of Virginia, Apr. 2013. DOI: 10.18130/V3KZ3C. URL: https://libraetd.lib.virginia.edu/public_view/h128nd96g.

[28] H. W. Lenstra. "Integer Programming with a Fixed Number of Variables". In: *Mathematics of Operations Research* 8.4 (1983), pp. 538–548. URL: http://www.jstor.org/stable/3689168.

[29] Stephen J. Maher, Ted K. Ralphs, and Yuji Shinano. *Assessing the Effectiveness of (Parallel) Branch-and-bound Algorithms*. arXiv:2104.10025 [cs]. Apr. 2021. DOI: 10.48550/arXiv.2104.10025. URL: http://arxiv.org/abs/2104.10025.

[30] George Nemhauser and Laurence Wolsey. "Linear Programming". In: *Integer and Combinatorial Optimization*. John Wiley & Sons, Ltd, 1988, pp. 27–49. ISBN: 978-1-118-62737-2. DOI: 10.1002/9781118627372.ch2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118627372.ch2.

[31] George L. Nemhauser and Laurence A. Wolsey. *Integer programming*. Working paper 841. Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), Jan. 1988. URL: https://EconPapers.repec.org/RePEc:cor:louvrp:841 (visited on 02/18/2026).

[32] Nils Niehues, Benjamin Arp, and Robert Heinrich. "Efficient Repair of Confidentiality Violations in Software Architectures". In: *Proceedings of the 23rd IEEE International Conference on Software Architecture (ICSA)*. To appear. 2026.

[33] Nils Niehues, Sebastian Hahner, and Robert Heinrich. "An Architecture-Based Approach to Mitigate Confidentiality Violations Using Machine Learning". In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. ISSN: 2835-7043. Mar. 2025 under review, pp. 107–118. DOI: 10.1109/ICSA65012.2025.00020. URL: https://ieeexplore.ieee.org/abstract/document/10978930.

[34] Nils Niehues et al. "Integrating security-enriched data flow diagrams into architecture-based confidentiality analysis". In: *Softwaretechnik-Trends Band 44, Heft 4*. Gesellschaft für Informatik eV, 2024. URL: https://dl.gi.de/items/52606a6b-0bf4-4bb9-a856-6663e9c69d52 (visited on 02/23/2026).

[35] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, Jan. 1998. ISBN: 978-0-486-40258-1.

[36] Sven Peldszus et al. "Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2019, pp. 23–33. DOI: 10.1109/MODELS.2019.00-18. URL: https://ieeexplore.ieee.org/abstract/document/8906984.

[37] Stanley Reiter and Gordon Sherman. "Discrete Optimizing". In: *Journal of the Society for Industrial and Applied Mathematics* 13.3 (Sept. 1965), pp. 864–889. ISSN: 0368-4245, 2168-3484. DOI: 10.1137/0113056. URL: http://epubs.siam.org/doi/10.1137/0113056.

[38] Francessca Rossi, Peter von Beek, and Toby Walsh. "Chapter 4 Constraint Programming". en-US. In: *Foundations of Artificial Intelligence*. Vol. 3. ISSN: 1574-6526. Elsevier, Jan. 2008, pp. 181–211. DOI: 10.1016/S1574-6526(07)03004-0. URL: https://www.sciencedirect.com/science/chapter/bookseries/abs/pii/S1574652607030040.

[39] Per Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples*. Google-Books-ID: T7rXoaxqPIAC. John Wiley & Sons, Mar. 2012. ISBN: 978-1-118-18100-3.

[40] Spyridon Samonas and David Coss. "THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY". In: *Journal of Information Systems Security* (2014).

[41] Simon Schneider and Riccardo Scandariato. "Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java". In: *Journal of Systems and Software* 202 (Aug. 2023), p. 111722. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111722. URL: https://www.sciencedirect.com/science/article/pii/S0164121223001176.

[42] Simon Schneider et al. *How Dataflow Diagrams Impact Software Security Analysis: an Empirical Experiment*. arXiv:2401.04446 [cs]. Jan. 2024. DOI: 10.48550/arXiv.2401.04446. URL: http://arxiv.org/abs/2401.04446.

[43] Simon Schneider et al. "microSecEnD: A Dataset of Security-Enriched Dataflow Diagrams for Microservice Applications". In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. ISSN: 2574-3864. May 2023, pp. 125–129. DOI: 10.1109/MSR59073.2023.00030. URL: https://ieeexplore.ieee.org/document/10174132.

[44] Felix Schwickerath et al. "Tool-supported architecture-based data flow analysis for confidentiality". In: *arXiv preprint arXiv:2308.01645* (2023). URL: https://arxiv.org/abs/2308.01645.

[45] Stephan Seifermann, Maximilian Walter, and Sebastian Hahner. *Auxiliar Material for Tutorial on Identifying Confidentiality Violations in Architectural Design Using Palladio*. Sept. 2021. DOI: 10.5281/ZENODO.5086778. URL: https://zenodo.org/record/5086778.

[46] Stephan Seifermann et al. "Detecting violations of access control and information flow policies in data flow diagrams". In: *Journal of Systems and Software* 184 (Feb. 2022), p. 111138. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111138. URL: https://www.sciencedirect.com/science/article/pii/S0164121221002351.

[47] Laurens Sion et al. "Solution-aware data flow diagrams for security threat modeling". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1425–1432. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167285. URL: https://dl.acm.org/doi/10.1145/3167132.3167285.

[48] Joshua Sprey et al. "SMT-based variability analyses in FeatureIDE". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 1–9. ISBN: 978-1-4503-7501-6. DOI: 10.1145/3377024.3377036. URL: https://dl.acm.org/doi/10.1145/3377024.3377036.

[49] International Organization for Standardization. *ISO/IEC 27000:2018 E Information technology - Security techniques - Information security management systems - Overview and vocabulary*. Tech. rep. 2018. URL: https://www.iso.org/standard/73906.html (visited on 08/15/2025).

[50] Katja Tuma, Riccardo Scandariato, and Musard Balliu. "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. Mar. 2019, pp. 191–200. DOI: 10.1109/ICSA.2019.00028. URL: https://ieeexplore.ieee.org/document/8703905.

[51] Council of European Union. *REGULATION EU 2016/679 General Data Protection Regulation*. Doc ID: 02016R0679-20160504. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/2016-05-04/eng (visited on 09/04/2025).

[52] Vijay V. Vazirani. "Set Cover". In: *Approximation Algorithms*. Ed. by Vijay V. Vazirani. Berlin, Heidelberg: Springer, 2003, pp. 15–26. ISBN: 978-3-662-04565-7. DOI: 10.1007/978-3-662-04565-7_2. URL: https://doi.org/10.1007/978-3-662-04565-7_2.

[53] Stef Verreydt et al. "Relationship-based threat modeling". In: *Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems*. EnCyCriS '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 41–48. ISBN: 978-1-4503-9290-7. DOI: 10.1145/3524489.3527303. URL: https://dl.acm.org/doi/10.1145/3524489.3527303.

[54] Maximilian Walter, Robert Heinrich, and Ralf Reussner. "Architectural Attack Propagation Analysis for Identifying Confidentiality Issues". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. Mar. 2022, pp. 1–12. DOI: 10.1109/ICSA53651.2022.00009. URL: https://ieeexplore.ieee.org/abstract/document/9779702.

[55] Maximilian Walter, Robert Heinrich, and Ralf Reussner. "Architecture-Based Attack Path Analysis for Identifying Potential Security Incidents". In: *Software Architecture*. Ed. by Bedir Tekinerdogan et al. Cham: Springer Nature Switzerland, 2023, pp. 37–53. ISBN: 978-3-031-42592-9. DOI: 10.1007/978-3-031-42592-9_3.

[56] Westley Weimer et al. "Automatically finding patches using genetic programming". In: *2009 IEEE 31st International Conference on Software Engineering*. ISSN: 1558-1225. May 2009, pp. 364–374. DOI: 10.1109/ICSE.2009.5070536. URL: https://ieeexplore.ieee.org/abstract/document/5070536.

[57] Danny Weyns. "Software Engineering of Self-adaptive Systems". In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyochul Kang. Cham: Springer International Publishing, 2019, pp. 399–443. ISBN: 978-3-030-00262-6. DOI: 10.1007/978-3-030-00262-6_11. URL: https://doi.org/10.1007/978-3-030-00262-6_11.

[58] Danny Weyns et al. "A survey of formal methods in self-adaptive systems". In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. C3S2E '12. New York, NY, USA: Association for Computing Machinery, June 2012, pp. 67–79. ISBN: 978-1-4503-1084-0. DOI: 10.1145/2347583.2347592. URL: https://dl.acm.org/doi/10.1145/2347583.2347592.

[59] Laurence A. Wolsey. *Integer Programming*. Google-Books-ID: knH8DwAAQBAJ. John Wiley & Sons, Sept. 2020. ISBN: 978-1-119-60652-9.