Hochschule
Bonn-Rhein-Sieg
University
of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Project Report

Course of Studies: Cybersecurity & Privacy

Developing a Trust-Based Knowledge-Based Collaborative Filtering Hybrid Recommender System to share vulnerability management decisions in DevGuard

submitted from

Dennis Tuan Anh Quach

First Examiner	Prof. Dr. Luigi Lo Iacono
Second Examiner	Prof. Dr. Hannes Tschofenig
Submitted on	2026-03-26

Declaration

I declare that I have written this report independently. All passages that are taken verbatim or paraphrased from published or unpublished works by third parties have been identified as such. All sources and instruments that I have used for the work are listed. In particular, the following AI systems were also used to create this work:

- ChatGPT-5.2, Claude Sonnet 4.5 and Claude Opus 4.6 models were used for the initial formulation, based on bullet points specified by me throughout the entire work.

I am aware that content generated by AI systems cannot replace careful scientific work, which is why all such content has been critically reviewed and finalized by me. The work has not yet been the subject of another examination with the same content or in essential parts.

Bonn, 2026-03-26 Signature

Contents

List of Figures

List of Tables

List of abbreviations

API Application Programming Interface

BFT Byzantine Fault Tolerant

BGP Byzantine Generals Problem

CF Collaborative Filtering

CISA Cybersecurity and Infrastructure Security Agency

CVE Common Vulnerabilities and Exposure

CV Crowdsourced Vexing

DoS Denial of Service

ECDSA Elliptic Curve Digital Signature Algorithm

HTTP Hypertext Transfer Protocol

KB Knowledge-Based

OIDC OpenID Connect

OPA Open Policy Agent

PPCF Privacy-Preserving Collaborative Filtering

PoC Proof of Concept

PoS Proof-of-stake

PoW Proof-of-work

RS Recommender System

SBOM Software Bill of Materials

TB Trust-Based

TLS Transport Layer Security

TOTP Time-based One-Time Password

UI User Interface

VEX Vulnerability Exploitability eXchange

Abstract

Software projects increasingly rely on large dependency ecosystems, where recurring vulnerabilities force maintainers to repeat similar vulnerability assessments across organizations. This project report addresses that redundancy in the context of the open-source DevGuard platform by designing and implementing a crowdsourced mechanism for sharing vulnerability management decisions through Vulnerability Exploitability eXchange (VEX) rules. While prior work suggested a Trust-Based (TB) Knowledge-Based (KB) Collaborative Filtering (CF) hybrid recommender approach, a practical system analysis showed that DevGuard lacks the rating data required for CF. Therefore, this work derives a custom algorithm based on trust-weighted majority voting, tailored to DevGuard's centralized architecture.

The proposed Crowdsourced Vexing (CV) system aggregates existing VEX assessments for a given Common Vulnerabilities and Exposure (CVE) and dependency path and recommends the most plausible rule. Security requirements were derived via attack tree analysis and integrated into the algorithm through in-scope mitigations, including minimum organization age checks, strict input validation, vote deduplication, minimum voter thresholds, deterministic tie handling, and exponential decay of repeated votes from the same creator to reduce profile injection impact.

A Proof of Concept (PoC) was implemented in the DevGuard backend and evaluated with 52 tests covering behavioral correctness, adversarial scenarios, and edge cases; all tests passed. The evaluation indicates that the approach is effective as an initial decision-support mechanism, while also revealing residual risks, particularly coordinated low-trust collusion, tie-induced deadlocks, and open questions around robust automatic trust score derivation. Overall, the report demonstrates the feasibility of trust-weighted crowdsourced VEX recommendations and outlines concrete priorities for production hardening.

1 Introduction

As software projects grow in complexity, security vulnerabilities and bugs are bound to find their way into the generated codebase and infrastructure, be it through package dependencies or human error [3], [4]. These security flaws usually need to be addressed by trained cybersecurity experts during the whole process of development, but modern industry tends to shift the responsibility of managing these issues to the software engineers, adding on a task that they are, more than often, not equipped to handle properly [5], [6], [14]. To help software developers address vulnerability management for projects, sophisticated tools have been created. One of those tools is the open-source DevGuard Project—invented and maintained by L3montree GmbH [11], [12]. DevGuard, at its core, provides features to simplify the workflow of detecting and handling critical security issues in applications during the development stage [11], [12].

A pattern that arose during the development of DevGuard is that when similar software projects contain similar dependency tree structure and when security flaws appear in those dependencies, these issues can be treated in similar ways. This is because the dependency tree of a project will determine the chain of questions and decisions a maintainer has to apply to conclude a treatment for a given issue. If projects now have similar dependency trees, these decision trees will be inherited. Therefore, similar projects will deterministically arrive at similar solutions. Knowing that, it would be highly beneficial to gather decisions from users for occurring issues through a crowdsourced approach and communicate those crucial and majority-established vulnerability management decisions between maintainers of these software projects. It is now proposed to build a system that implements these features as core functionalities within the framework of DevGuard to aid developers in deciding the right course of action for handling emerging vulnerabilities.

1.1 Previous Work

In previous work, efforts have been made to find a suitable approach in literature and current research to model the system concept [16]. Recommender System (RS)s were chosen due to their similarities to the proposed system. These systems use special types of algorithms to filter an oversaturated pool of information for the most precise answer to a request, solving problems like information overload [8], [9], [15]. Within a system where the validity of information is determined by the amount of usage of that information, a RS could calculate and provide what statements should be correct and communicate those decisions. The work has concluded by choosing a Trust-Based (TB) Knowledge-Based (KB) Collaborative Filtering (CF) Hybrid RS approach as a suitable candidate to represent the system [16]. This current work will now focus on reviewing the concluded result and consider challenges of a practical context to then construct a Proof of Concept (PoC) implementation within DevGuard.

2 Scope

Within the scope of this work, interaction with concepts and system components will be limited to specific regions of the DevGuard framework. The following sections will explain the relevant considered areas for context.

2.1 DevGuard

DevGuard is an open-source tool designed to support modern software development and supply chain security. It provides organizations with the means to continuously monitor, assess, and mitigate vulnerabilities across software components, third-party libraries, and internal projects. By integrating automation, collaboration, and standardized vulnerability reporting, DevGuard enables teams to maintain a secure development lifecycle while reducing manual overhead. The platform emphasizes transparency, offering stakeholders clear insights into which components are at risk and how those risks are being managed. In this way, DevGuard helps organizations proactively reduce the attack surface of their software while ensuring compliance with evolving cybersecurity standards [17], [19].

2.2 Vulnerability Management in DevGuard

DevGuard's Vulnerability Management toolchain streamlines the process of identifying and mitigating software vulnerabilities. The system continuously scans project dependencies, monitors for new Common Vulnerabilities and Exposure (CVE)s, and prioritizes issues based on factors such as severity, exploitability, and contextual impact on specific components. Instead of merely listing vulnerabilities, DevGuard provides actionable intelligence, indicating whether a vulnerability is relevant to the specific software environment and suggesting mitigation strategies. By combining automated detection with structured reporting, DevGuard allows security teams to quickly focus on high-priority issues, track remediation efforts, and maintain an auditable record of risk management activities. This approach ensures that organizations can respond effectively to emerging threats without overwhelming developers with unnecessary alerts [17], [19].

2.3 VEX

The Vulnerability Exploitability eXchange (VEX), as defined by the Cybersecurity and Infrastructure Security Agency (CISA), is a standardized framework designed to articulate the impact and exploitability status of specific software vulnerabilities in a machine-readable format. Unlike traditional vulnerability reporting, which often overwhelms users with raw CVE lists without context, VEX enables organizations to communicate precisely whether a given vulnerability affects a particular product, component, or deployment scenario. The standard acknowledges that the mere presence of a CVE in a Software Bill of Materials (SBOM) does not necessarily imply actual risk. Vulnerable code may be excluded in builds, unreachable at runtime, or mitigated through compensating controls. By expressing assessments such as `Not Affected`, `Affected`, `Fixed`, or `Under Investigation` along with justifications, VEX transforms vulnerability management from an undifferentiated enumeration of potential issues into targeted, context-aware documentation of true exploitable impact. This capability not only streamlines internal security decision-making but also facilitates automated ingestion of vulnerability posture by downstream consumers, auditors, and supply chain partners, thereby reducing redundant analysis and improving transparency across software ecosystems [7], [10], [18].

2.4 Definition: Vexing

In the context of VEX, this work will define the term *vexing* as the process of conducting a VEX-compliant vulnerability assessment. This involves systematically analyzing software components, determining which known vulnerabilities are actually relevant, and documenting the findings in a standardized, machine-readable format. Vexing represents a meticulous, context-driven approach to vulnerability management, emphasizing accuracy, clarity, and actionable insight [18].

2.5 Vex rules

VEX rules constitute a formal mechanism for codifying and automating recurring vulnerability mitigation decisions based on the VEX paradigm. Within DevGuard, a VEX rule is defined by three fundamental components: (1) a specific CVE identifier, indicating the vulnerability of interest; (2) the actual VEX rule itself, a dependency path pattern, expressed with support for wildcards and suffix matching, that determines which dependency tree and context the rule applies to; and (3) an actionable outcome, typically designating the vulnerability as a false-positive or marking it for risk acceptance without manual intervention. By mapping these rules against the dependency graph of an application during each scan, systems can automatically apply previously validated decisions whenever matching patterns are encountered, thereby eliminating repetitive manual triage. The path pattern logic supports nuanced scoping—for example, differentiating between vulnerabilities that only arise when a package is reached through specific ancestors versus those that universally apply regardless of path. As rule applications are idempotent, they integrate safely into continuous integration pipelines and large-scale scanning workflows. Ultimately, VEX rules not only streamline internal vulnerability operations but also serve as interoperable artifacts that can be imported from or exported to external VEX documents, enhancing collaboration and reducing duplicated security assessments across projects [18].

Structurally, a VEX rule follows a dependency-chain abstraction that models how vulnerabilities propagate through function calls between packages. The rule typically contains a wildcard element (`*`) representing any arbitrary set of dependencies above a certain point in the dependency tree, followed by an ordered sequence of dependencies describing the call chain leading to the vulnerable component. An illustrative pattern can be expressed as `*` $\rightarrow$ `A` $\rightarrow$ `B` $\rightarrow$ `C` $\rightarrow$ `D`, where `D` denotes the package that contains the vulnerable code associated with the specified CVE. In this structure, each dependency after the cutoff point represents a concrete function-call relationship in which a package imports the next dependency and invokes functionality that may eventually reach the vulnerable code [18].

A critical concept in this structure is the *cutoff point*, which is the dependency immediately following the wildcard. This dependency imports the next element in the chain but does not invoke any function that would lead to the vulnerable execution path. In the example above, package `A` imports `B` but does not call any function that would trigger the vulnerability located in `D`. Consequently, all dependencies represented by the wildcard (`*`)—that is, every package closer to the dependency tree root—are also considered unaffected by the vulnerability, since the execution path is effectively terminated at the cutoff point [18].

Conversely, the dependencies that follow the cutoff point represent an active propagation chain. For instance, `B` imports `C` and invokes a function that leads further down the chain, while `C` imports `D` and calls a function that ultimately reaches the vulnerable code within `D`.

The rule therefore captures the precise boundary between benign dependency inclusion and exploitable execution paths. By formally encoding this boundary within the dependency pattern, VEX rules allow automated analysis systems to determine when vulnerabilities are technically present in the dependency graph but cannot be triggered by the application's actual call structure. This structural representation significantly improves the precision of automated vulnerability triage and reduces false-positive findings in complex dependency ecosystems [18]. Table 1 shows a few examples for possible VEX rule patterns.

Type	**Pattern**	**Explanation**
Wildcard Cutoff	`*` → `A` → `B` → `C` → `D`	The wildcard represents any dependency chain above the cutoff point. Dependency `A` imports `B` but does not invoke any function that triggers the vulnerability further down the chain. Dependencies `B` and `C` propagate function calls until the vulnerable package `D` is reached. All dependencies represented by the wildcard are therefore considered unaffected.
Short Propagation Chain	`*` → `A` → `B`	Represents a minimal chain where `B` contains the vulnerability and `A` imports it without triggering the vulnerable execution path.
Direct Dependency Cutoff	`ROOT`	Represents a vulnerability that directly located in the project
Global Vulnerability Rule	`*` → `D`	Represents a vulnerability located in dependency `D` (last dependency) that is considered unreachable regardless of the calling dependency chain. The wildcard captures all possible upstream paths.

Table 1: Example structural patterns of VEX rules in dependency chains

2.6 Workflow

As DevGuard contains plenty of tools and many workflows, it is important to narrow down the scope of what this work is concentrating on. The proposed system intends to improve the vexing workflow for vulnerabilities in dependencies of software projects. Currently, DevGuard enables maintainers to conduct vexing and afterwards encourages the developer to create VEX rules to streamline the process. If the maintainer does not import an externally created VEX rule, the vexing process itself has to be done manually. To improve this meticulous process, the proposed system intends to calculate and propose a fitting VEX rule to the user, ready for application. Therefore, the workflow that will be covered can be summarized in the steps below. Anything that is not covered by these steps will be considered as out-of-scope and, under circumstances, will not be reflected in the analysis or implementation process of this work.

1. The user requests a proposal (VEX rule) for an assessment (vexing) as the result of a trigger.
   (a) The user interacts with the frontend to input his request (*)
   (b) The request is sent to the backend and received by the algorithm.

2. The algorithm calculates a proposal.
    (a) The algorithm validates the received input and adheres to system constraints.
    (b) The algorithm acquires additional data from sources (e.g., databases) (*)
    (c) The algorithm calculates a result.
    (d) The algorithm returns the result.
    (e) The response is returned to the user.
3. The user receives a proposal.
    (a) The user accepts or rejects the proposal

A *Trigger* is defined as something that causes the user to act in the system and require a proposal, e.g., in the context of DevGuard, an emerging vulnerability in the project. Steps that are marked with (*) are optional and might not occur in every execution of the workflow.

2.6.1 System Data

To validate if the proposed system integrates correctly into the DevGuard framework and ensure its applicability, it is necessary to define the parameters of the surrounding system. These can include the input, the output, static data that is accessible by the system, and entities that act in the system. Given these parameters, mappings to proposed solutions can be established. The existence of such mappings demonstrates structural compatibility between the algorithm and the system, indicating that the algorithm can process the system's data and produce viable results. Figure 1 illustrates this logic. As the proposed system is supposed to integrate into DevGuard itself, DevGuard's current data models, features and functions will dictate the parameters that will be available to the algorithm. Table 2 and Figure 2 show which parameters are stipulated by DevGuard.

Parameter	Datatype	Entity Type
Users	-	Actor
CVE	`String`	Trigger
VexRules	`Array<VexRule>`	Inputdata
DependencyPath	`Array<String>`	Inputdata
Trustscores	`Array<Float>`	Inputdata
VexRule	`VexRule`	Outputdata

Table 2: Required parameters for the proposed system

Users represent the actual actors who interact with the proposed system. They are included as a parameter not because their individual actions directly influence the algorithm's computation, but because they define the surrounding environment in which the system operates. Users interact with the system through the DevGuard frontend and are subject to constraints such as authentication requirements, role-based access control, and visual feedback mechanisms that govern how recommendations are presented and accepted. The CVE serves as the trigger that initiates the recommendation workflow. An emerging or newly detected CVE within a project's dependencies constitutes the incentive for the user to request a VEX rule proposal from the system. The CVE identifier thereby defines the specific vulnerability for which the algorithm must determine a crowd-endorsed

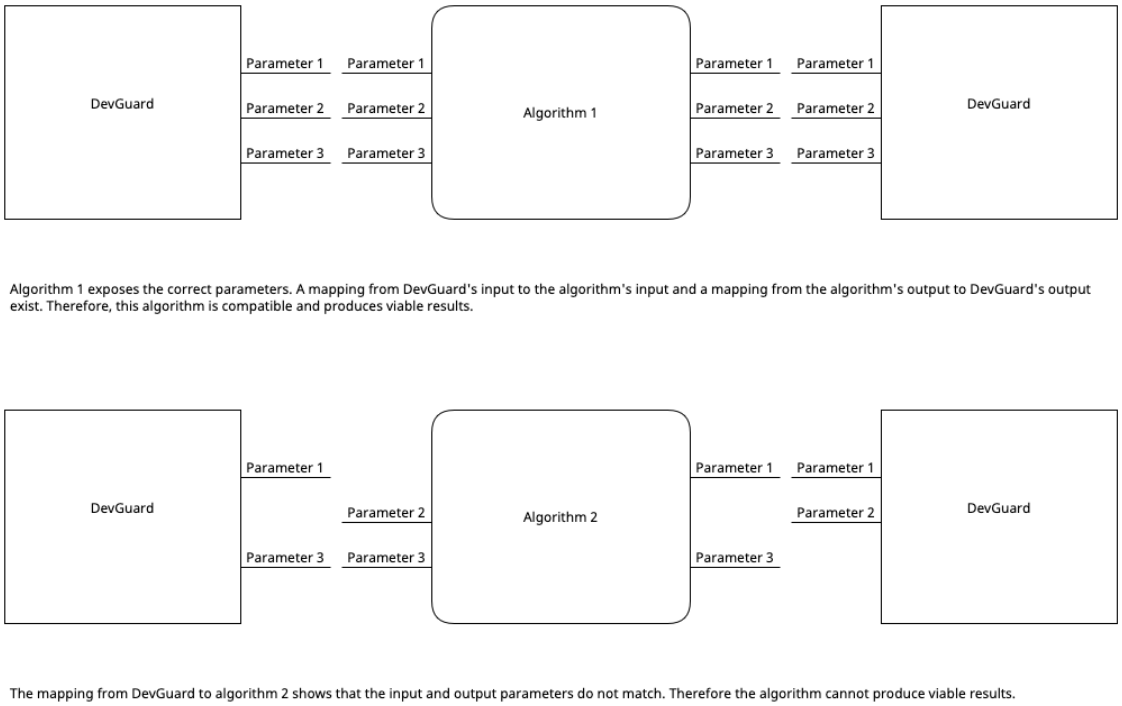

Figure 1: Illustration of mapping logic

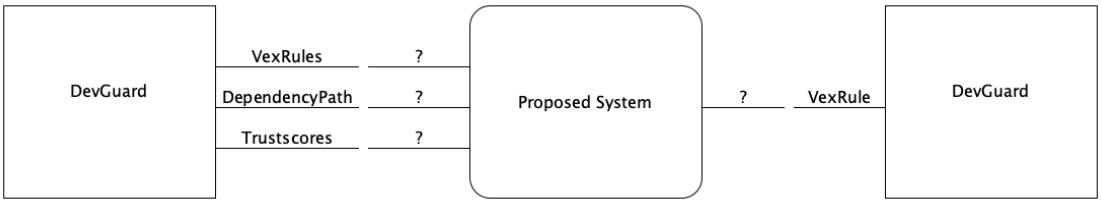

Figure 2: Parameters stipulated by DevGuard

assessment. The input set of VEX rules comprises the complete collection of VEX rules stored within DevGuard at the moment of recommendation, encompassing all rules created across all organizations and projects—not only those whose path patterns match the requesting project's dependency path. The algorithm itself is responsible for filtering this comprehensive set to identify the rules that are relevant to the given dependency context and CVE. The dependency path represents the specific chain of dependencies from the project. For an emerging vulnerability, DevGuard specifies and links the affected path of a project rather than the whole dependency tree to granularly differentiate assessment cases. It is expressed as an ordered sequence of dependency identifiers starting with the own project as the root and ending at the affected dependency. Further, it serves as the contextual reference against which VEX rule path patterns are matched during the filtering phase of the algorithm. Trust scores provide floating-point values corresponding to all entities within DevGuard that carry a trust score—specifically, organizations and projects. These scores determine how much trust is assigned to an entity and will later influence the algorithm's output. Finally, the output of the algorithm is a single VEX rule that represents the crowd-endorsed recommendation for the given CVE and dependency path.

3 System analysis

As mentioned previously, the proposed TB KB CF Hybrid RS solution is solely based on recent literature and research [16] and has not considered the practical context it will be implemented in. When approaching the implementation, it is now required to align the theory with reality. Therefore, in this section, the proposed system will be reviewed and contextualized for practical implementation.

3.1 Reviewing the Proposed System

To summarize the contents of the previous work, it can be said that the proposed solution intends to create a system that aggregates security-relevant information and calculates which information is fitting for a given recommendation problem through a CF approach, while also protecting the mechanism through the use of a TB and Privacy-Preserving Collaborative Filtering (PPCF) extension [16]. Furthermore a KB is planned in as a fallback, when CF suffers from cold-start and sparsity problems. Though this approach is founded on the fairly sophisticated assumption that RS and the proposed system share a quite similar basis [16], it appears that the proposed TB KB CF Hybrid RS and RS approach in general run into discrepancies with the DevGuard framework in practice. Creating a system mapping to assess compatibility between the proposed system and DevGuard reveals multiple problems that can be divided by the underlying RS-Types.

3.1.1 Collaborative Filtering

Taking a look at the mapping between DevGuard and CF (Figure 3), it can be seen that most of the parameters of CF align into accordance with the framework. Since DevGuard intends to recommend VEX rules to the developer, they can be considered as both the item base (input) itself and also the recommended item (output). Furthermore, though it may seem strange at first, the dependency path of the given project acts as a surrogate for the user in the proposed system. This is because the user normally identifies uniquely for whom the recommendation is for and also portrays one central object to base the similarity comparison upon. Both of this properties are provided by the dependency path object from the DevGuard framework. What is important now is the crucial parameter `Rating`. Ratings are essential to the CF RS, because they are the primary source of information in the algorithm [16]. As the mapping shows, this parameter finds no according input value from DevGuard. This is because DevGuard itself does not provide some sort of rating feature or substitute feature for VEX rules, which can be used to aggregate rating information. This missing information makes CF and DevGuard incompatible. A workaround that could be applied here, would be to count every application of a VEX rule as a rating, but that would reduce the information gained from the rating to a binary domain, which would then again reduce CF itself to a majority-based voting system.

3.1.2 Knowledge-Based Recommender System

Figure 4 shows that a KB RS integrates more suitably into the DevGuard framework, as it requires only a few parameters to establish the knowledge base. The items will again consist of VEX rules, where the path information of the VEX rule will make up the item features. User requirements are, also again, reflected by the dependency path information,

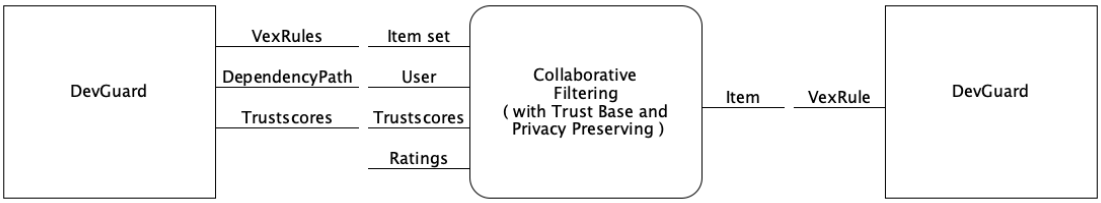

Figure 3: Mapping between DevGuard and Collaborative Filtering

since the dependency path (in combination with a CVE) dictates which items (features) will be needed. Now, although the mappings resolve for a KB system, a KB system alone wouldn't be sufficient to provide a foundation to implement a TB or PPCF approach, since these rely on an intensified involvement of users in the decision process, which was only provided by CF. Therefore, with a KB system alone, it is not possible to prevent profile injection attacks and privacy concerns, which makes a standalone KB RS not viable for integration.

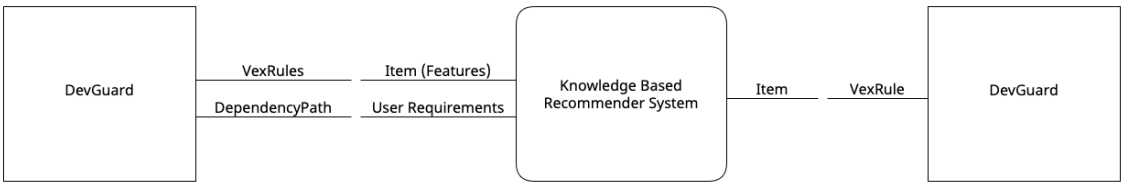

Figure 4: Mapping between DevGuard and Knowledge-Based Recommender Systems

3.2 Alternative Approaches

Section 3.1 has revealed crucial discrepancies in the established theory. This begs the question if RSs in general are even a suitable choice for the proposed system, if not even the most suitable algorithm by established theory is applicable in practice. As there are further approaches beyond the realm of RSs, the following sections will explore other options for suitable candidate approaches.

3.2.1 Proof-of-Stake

Proof-of-stake (PoS) is a consensus mechanism that enables a decentralized network to agree on a single history of transactions by having participants lock up economic value as collateral rather than expend computational energy in comparison to Proof-of-work (PoW). In PoS, nodes known as *validators* must commit (stake) currency into the protocol. This stake acts as a bond that can be reduced or destroyed if the validator acts dishonestly, thereby aligning incentives for honest behavior. Validators are selected pseudo-randomly to propose and validate transactions and blocks, and their votes on the validity of blocks form the basis of the chain's fork-choice rule, which prioritizes the branch supported by the greatest accumulated stake weight (at least 51%). A robust PoS system also incorporates rewards for active, correct participation and penalties for inactivity or conflicting behavior, ensuring that consensus reflects the views of stakeholders with the most at risk. Additionally, protocols often implement mechanisms to achieve finality, meaning that once enough validators attest to a state transition, that state becomes extremely costly to reverse without incurring large economic losses for an attacker, thereby securing the network against common attack vectors [20].

Once again, it is possible to create a mapping between the approach and DevGuard to validate compatibility. Table 3 shows such a mapping. It can be seen that all parameters can be resolved between the systems. In this mapping, DevGuard users correspond to the validator, user, and node entities in PoS, as they would participate in the validation and consensus process. A CVE serves as the transaction request that triggers the consensus mechanism, while VEX rules constitute the ground truth that validators must agree upon. The dependency path acts as the transaction itself, containing the contextual information for decision-making. Furthermore, trust scores naturally map to the currency concept in PoS and can be used as the stake that users commit to the system. The final recommended VEX rule represents the common opinion achieved through consensus.

However, despite the apparent compatibility of parameter mappings, fundamental architectural requirements for a PoS implementation are not satisfied by the DevGuard framework. The primary obstacle lies in the fact that DevGuard operates as a centralized system, whereas PoS inherently requires a decentralized architecture where participants collectively maintain and distribute a shared ground truth. In a PoS system, each validator must independently hold and verify the complete state of the blockchain, enabling trustless consensus through redundancy and cryptographic verification. The current DevGuard architecture does not support such distributed state management. Users are represented merely as identifiers within a centralized database and lack the necessary infrastructure to function as autonomous nodes capable of maintaining consensus state, validating transactions, or participating in fork-choice algorithms.

A further fundamental incompatibility concerns the mapping of trust scores to the currency concept in PoS. While the parameter mapping in Table 3 suggests a structural correspondence between the two, this equivalence does not hold at a functional level. In established PoS systems such as Ethereum, the staked currency—Ether—possesses intrinsic economic value: it can be exchanged for goods, services, or other currencies, and its monetary worth provides a tangible incentive for validators to accumulate and preserve it [20]. The prospect of financial reward for honest participation and the risk of losing economically valuable collateral through slashing penalties create a robust incentive alignment that underpins the security guarantees of the protocol [20]. In contrast, the trust score within DevGuard carries no such direct economic or utilitarian value. It functions as an internal metric of behavioral reliability rather than a transferable asset, and consequently, ordinary users lack a compelling incentive to maximize or protect their trust score. While certain categories of participants—such as established security researchers or organizations with reputational interests—may derive indirect benefit from maintaining a high trust score, the majority of users in a general-purpose vulnerability management platform would not perceive a meaningful consequence from its reduction. Without this incentive structure, the staking mechanism that constitutes the core security guarantee of PoS—namely, that validators risk losing something of value when they act dishonestly—cannot be meaningfully replicated within DevGuard.

Consequently, the PoS approach is not suitable for integration into the DevGuard system. Implementing such a mechanism would necessitate not only a fundamental overhaul of DevGuard's architecture, transforming it from a centralized platform into a fully decentralized network with peer-to-peer communication, distributed ledger technology, and sophisticated consensus protocols, but also the introduction of an incentive model that endows the staked quantity with sufficient value to deter adversarial behavior. This transformation would require not only substantial development effort but also a complete redesign of data storage, user authentication, and information propagation mechanisms. Given the scope and objectives of this project, such an extensive architectural restructuring is neither feasible nor aligned with the practical implementation goals. Therefore, while PoS

presents a theoretically interesting approach to achieving consensus in crowdsourced systems, its application remains impractical within the constraints of the existing DevGuard framework.

DevGuard Parameter	PoS Parameter	Entity Type
Users	Validator, User, Node	Actor
CVE	Transaction Request	Trigger
VexRules	Ground Truth	Inputdata
DependecyPath	Transaction	Inputdata
Trustscores	Currency (Amount for every participant)	Inputdata
VexRule	Common Opinion	Outputdata

Table 3: Required parameters for the proposed system

3.2.2 Byzantine Generals Problem

The Byzantine Generals Problem (BGP) describes the challenge of achieving reliable consensus in a distributed system where some participants may behave arbitrarily or maliciously. In this model, independent actors (the "generals") must agree on a single coordinated action, yet some actors may disseminate false or inconsistent information, thereby undermining agreement. The problem demonstrates that consensus is achievable only if the number of faulty nodes f satisfies $n \geq 3f + 1$ in a synchronous system with authenticated messaging. This theoretical result provides a foundation for designing Byzantine Fault Tolerant (BFT) algorithms, which ensure that honest nodes converge on the same correct value despite adversarial behavior. In a crowdsourced information system, contributors may provide erroneous or intentionally misleading data. By applying BFT-inspired mechanisms—such as redundancy, majority or supermajority voting thresholds, and iterative message exchange protocols—the system can algorithmically filter out inconsistent responses and converge on a value supported by a sufficiently large subset of mutually consistent contributors. Thus, the Byzantine Generals framework offers a principled method for constructing robust aggregation algorithms that tolerate misinformation while preserving correctness guarantees under bounded adversarial assumptions [1].

However, applying a consensus algorithm based on the BGP to the DevGuard system presents several fundamental incompatibilities. First, consensus algorithms are designed to produce a binding agreement among all honest participants, meaning the system automatically converges on a single action that all nodes must adopt. This behavior is inherently at odds with the proposed system's objective, which is to suggest a VEX rule that the user may then accept or reject based on their specific circumstances. A consensus mechanism would not afford the user this choice, but would instead enforce the collectively determined outcome. Moreover, it is important to note that BFT consensus guarantees agreement—that all honest nodes arrive at the same value—but it does not guarantee that the agreed-upon value is objectively correct. It merely ensures consistency across participants, which alone does not fulfill the requirement of providing accurate vulnerability assessments.

Second, and more critically, the BGP is formulated exclusively for decentralized systems in which nodes communicate directly with one another through peer-to-peer messaging. The core Byzantine fault model assumes that a malicious actor can send contradicting messages to different peers simultaneously, thereby sowing inconsistency across the network [1]. In DevGuard's centralized architecture, this communication direction is structurally not implemented. Users do not exchange information directly with one another, nor do

they rely on aggregating messages received from individual peers directly so far. Instead, each user submits a single assessment to the central server, and the server distributes exactly that assessment uniformly to all other participants without inconsistency. Since all communication is mediated through a single authoritative node, the possibility of a user disseminating contradictory information to different recipients does not arise. The very threat model that BFT algorithms are designed to mitigate is therefore inapplicable to DevGuard's architectural context to a certain point, rendering a Byzantine fault tolerant consensus approach unsuitable for integration into the system.

3.2.3 Custom Algorithms

The preceding analysis has demonstrated that none of the established approaches examined— CF, KB RSs, PoS consensus mechanisms, and BFT algorithms derived from the BGP —are fully compatible with the architectural and functional constraints of the DevGuard framework. CF fails due to the absence of a rating mechanism within DevGuard, reducing any workaround to a binary voting scheme and minimizing the benefits of CF. A standalone KB RS, while structurally mappable, lacks the multi-user interaction model required to support TB and PPCF extensions, leaving the system vulnerable to profile injection attacks and privacy concerns. PoS, despite yielding a complete parameter mapping, demands a decentralized network architecture that is fundamentally incompatible with DevGuard's centralized design. Also DevGuard is unable provide an incentive through trust scores to encourage correct behaviour for a secure application of PoS. Similarly, BFT consensus protocols presuppose peer-to-peer communication and a distributed threat model that does not arise in a centralized system where all user interactions are mediated by a single authoritative server. Given these incompatibilities, it becomes evident that adopting an existing algorithmic paradigm without substantial modification is not feasible. Consequently, given the constraints imposed by the DevGuard framework, this work attempts to design a custom algorithm that is tailored to the specific data model, architectural properties, and functional requirements of the system. Such an algorithm can selectively incorporate proven principles from the analyzed approaches, such as trust-weighted aggregation, knowledge-based matching on dependency path features, and majority-based consensus logic, while avoiding the structural prerequisites that render the original frameworks inapplicable. As this constitutes a novel and context-specific design effort, the resulting implementation will inherently require iterative refinement and is expected to evolve through successive evaluation cycles, even beyond this work, rather than arrive at a fully mature solution within the initial iteration. Nevertheless, this tailored approach ensures that the resulting system remains architecturally consistent with DevGuard's centralized infrastructure, operates effectively on the available data parameters outlined in Table 2, and fulfills the objective of providing accurate, crowd-informed VEX rule recommendations to the end user, while also establishing a secure application environment.

4 System definition

Before proceeding to the implementation, it is necessary to establish a clear and rigorous definition of the system to be constructed. A well-defined system specification serves as a foundational reference throughout the remainder of this work, ensuring that design decisions, implementation choices, and evaluation criteria remain aligned with the intended goals. Without such a definition, approaching the implementation directly would increase

the risk of architectural inconsistencies, ambiguous requirements, and undetected design flaws. Therefore, this section formalizes the system's structure, its security considerations, and its core mechanisms, providing a coherent blueprint that guides the subsequent development process.

4.1 Threat Modeling

Given that the proposed system operates in the domain of vulnerability management, where the integrity and trustworthiness of information are of critical importance, it is essential to consider potential attack vectors and adversarial scenarios before committing to an implementation. To systematically identify and reason about threats to the system, it was decided to conduct an attack tree analysis as the primary requirement analysis and design step. Attack trees provide a structured, hierarchical representation of the various ways in which an adversary could compromise system objectives, enabling the derivation of concrete countermeasures from identified threat paths [2], [13]. By modeling potential attacks in this manner, design decisions can be grounded in a threat-informed rationale rather than ad-hoc assumptions, thereby strengthening the security posture of the resulting system.

The attack tree was constructed through a systematic collaborative brainstorming process. Known attack patterns from related domains—such as profile injection in recommender systems [16] and common web application vulnerabilities—were mapped onto the specific architectural characteristics and trust assumptions of the proposed system to derive context-specific threat scenarios. It should be noted that the resulting attack tree does not claim exhaustiveness; rather, it is intended to capture the most relevant and plausible threats given the system's centralized architecture, its reliance on crowd-contributed assessments, and the security-critical nature of its output. Threat paths that presuppose capabilities or system configurations outside the scope of the DevGuard framework were deliberately excluded to maintain analytical focus.

By decomposing the root objective—manipulation of the majority voting system—into these progressively more concrete attack paths, the analysis exposes the specific threat scenarios that the system design must account for. The attack tree presented in Figure 5 structures the space of adversarial strategies along three primary branches, each targeting a distinct layer of the system: (1) compromising voters, (2) manipulating the communication between voters and the system, and (3) compromising the voting algorithm itself. The identified threat categories are summarized below.

- **Compromise Voters.** An adversary may seek to influence voting outcomes by gaining control over a sufficient fraction of the participating entities. This branch encompasses several sub-strategies: hijacking honest voters through stolen credentials or malware, mass-creating voter accounts (i.e., Profile Injection attacks) to artificially inflate representation, removing honest voters from the system by forcing account bans, and exploiting predictable voting behavior by exposing the system's recommendation goal to inattentive users.
- **Manipulate Communication between Voters and System.** Rather than compromising voters directly, an attacker may intercept or disrupt the communication channel between voters and the centralized system. This branch includes man-in-the-middle attacks aimed at modifying vote messages in transit, replay attacks that re-submit previously valid votes, and denial-of-service strategies that crash voting

nodes, disconnect honest voters from the network, or block their messages, for instance through resource overload or forced token expiration.

- **Compromise Voting Algorithm.** A third category targets the algorithm implementation itself. By exploiting software bugs-such as authentication validation flaws, insufficient input validation, or rounding errors—an adversary may directly manipulate the aggregation logic.

From these threat paths, concrete mitigations were derived. Each mitigation was classified according to its implementation scope: mitigations that can be directly enforced within the algorithm logic of the proposed system are marked as *in scope* and will be integrated into the algorithm design itself. Mitigations that depend on platform-level capabilities—such as authentication infrastructure, network security, or account management policies—fall outside the algorithm's direct control and are designated as system-level requirements that must be satisfied by the DevGuard framework. Table 4 lists the derived mitigations together with their originating threat context and scope classification.

ID	Root	Closest Parent	Attack	Mitigation	In Scope
1	Compromise Voters	Hijack honest voters	Stolen credentials / tokens	Enforce two-factor authentication for DevGuard users	No
2	Compromise Voters	Hijack honest voters	Stolen credentials / tokens	Implement token expiry	No
3	Compromise Voters	Hijack honest voters	Install malware	Enforce system standards for the use of DevGuard	No
4	Compromise Voters	Hijack honest voters	—	Provide best-practice guidelines for environment, setup, and security measures	No
5	Compromise Voters	Hijack honest voters	—	Implement account recovery	No
6	Compromise Voters	Hijack honest voters	—	Implement emergency account lockdown	No
7	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Limit the number of organizations and projects a single user can create	No
8	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Increasingly decrease the value of each vote created by an organization or project of the same user	Yes
9	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Implement CAPTCHA to prevent automated bot creation	No
10	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Require a minimum organization age to be able to vote	Yes
11	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Value trust of organization based on age	Yes
12	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Require user/organization verification through an external provider	No
13	Compromise Voters	Mass-create voter accounts	Mass-create voter accounts	Implement trust score to devalue votes of shallow accounts	Yes
14	Compromise Voters	Remove honest voters	Force voter ban	Enforce multi-factor checks for account disabling	No

Continued on next page

Table 4 — continued from previous page

ID	Root	Closest Parent	Attack	Mitigation	In Scope
15	Compromise Voters	Remove honest voters	Force voter ban	Require a minimum number of voters for a decision; disabling the recommendation when too few voters remain	Yes
16	Compromise Voters	Exploit predictable voting	Expose goal to blunt voters	Implement visual indicators for crucial acceptance actions	No
17	Compromise Voters	Exploit predictable voting	Expose goal to blunt voters	Implement multi-step dialogue for accepting risks and recommendations	No
18	Manipulate Communication	Man-in-the-middle attack	Modify sent vote messages	Implement message signing for backend	No
19	Manipulate Communication	Man-in-the-middle attack	—	Enforce TLS for all requests	No
20	Manipulate Communication	Replay old votes	Replay old votes	Ensure that votes are unique by using a combination of data structure fields to identify each vote and removing duplicates	Yes
21–26	Manipulate Communication	Crash / disconnect / block nodes	Denial of Service (DoS) / resource overload	Implement rate limiting and load balancing	No
27	Manipulate Communication	Block honest nodes	Force token expiration	Enforce multi-factor checks for account disabling	No
28	Compromise Algorithm	Exploit software bug	Authentication validation	Enforce two-factor authentication for DevGuard users	No
29	Compromise Algorithm	Exploit software bug	Authentication validation	Use well-tested authentication framework	No
30	Compromise Algorithm	Exploit software bug	Lack of input validation	Implement choosable options instead of free-form input	Yes
31	Compromise Algorithm	Exploit software bug	Rounding issues	Use standardized cutoff; test with extreme values; define deterministic tie-breaking rules	Yes

Table 4: Derived mitigations from the attack tree analysis. Mitigations marked *In Scope = Yes* will be addressed in the algorithm design; all others are treated as platform-level requirements for DevGuard.

As Table 4 shows, a subset of mitigations falls within the direct control of the algorithm and can be incorporated into its design and implementation. These in-scope mitigations (IDs 8, 10, 11, 13, 15, 20, 30, 31) address concerns such as devaluing votes from shallow or mass-created accounts through trust score weighting, enforcing minimum voter thresholds, vote deduplication, and ensuring deterministic input validation and tie-breaking behavior. The remaining mitigations constitute platform-level requirements—such as two-factor authentication, token expiry, Transport Layer Security (TLS) enforcement, CAPTCHA, rate limiting, Hypertext Transfer Protocol (HTTP) message signing, adoption of a well-tested authentication framework, and account management policies—that DevGuard must provide as part of its infrastructure. While these platform-level measures are essential to

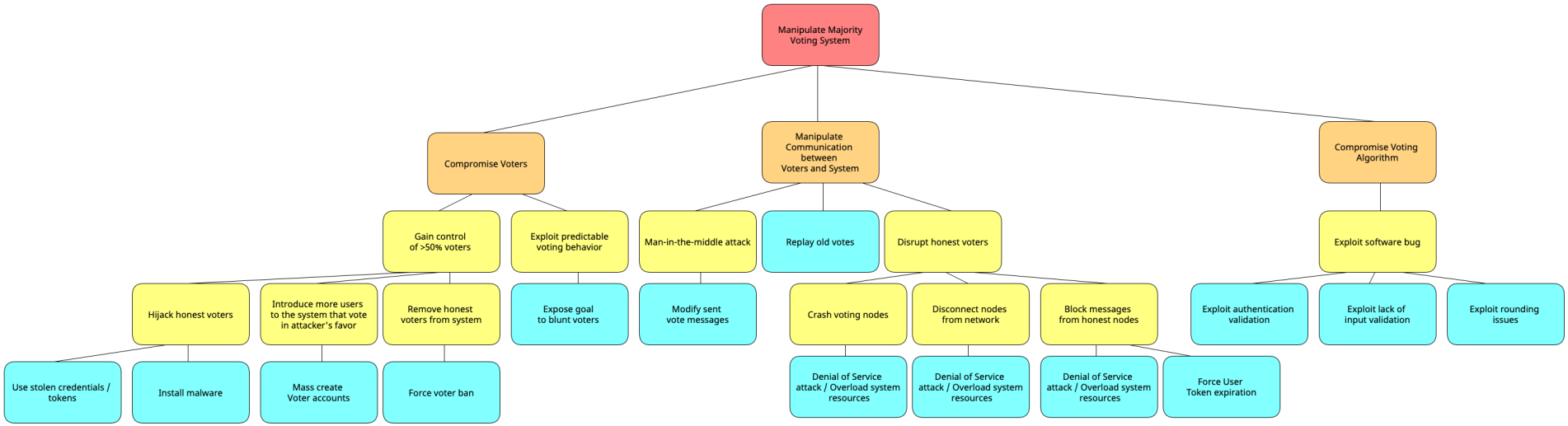

Figure 5: Attack Tree for Proposed System

the overall security posture of the crowdsourced vexing system, their implementation lies outside the scope of the algorithm itself and will therefore be treated as prerequisites that the DevGuard framework is expected to fulfill. The satisfaction of these requirements is assumed for the remainder of this work.

4.2 Proposed Algorithm

Building upon the findings of the system analysis (Section 3.1) and the threat model established in the preceding section, this section defines the algorithmic foundation of the new system. The algorithm must operate within the constraints of DevGuard's centralized architecture, leverage the available data parameters outlined in Table 2, and incorporate the in-scope threat mitigations identified in Table 4.

4.2.1 Majority-Based Voting as Foundation

The system analysis conducted in Section 3.1 revealed that the CF approach, when applied to DevGuard's data model, inherently reduces to a binary voting scheme due to the absence of a rating mechanism. Specifically, since DevGuard does not provide granular rating information for VEX rules, any attempt to implement CF would effectively count each application of a VEX rule as a single, unweighted vote, thereby collapsing the collaborative filtering computation into a majority-based voting system. Rather than treating this reduction as a deficiency, this work adopts majority-based voting as the explicit algorithmic foundation, recognizing that it constitutes the most natural and structurally compatible mechanism for the given data model.

This choice is not merely a consequence of data constraints but is also grounded in established theoretical principles. The analysis of the BGP demonstrated that majority and supermajority voting thresholds are foundational mechanisms for achieving consensus under adversarial conditions in BFT algorithms [1]. While the full Byzantine consensus protocol is inapplicable to DevGuard's centralized architecture, as discussed in the system analysis, the underlying principle—that the correct outcome can be reliably identified when a sufficient majority of honest participants agrees—remains directly applicable to the crowdsourced vexing context. In a system where multiple organizations independently assess the exploitability of the same vulnerability, a majority vote over their assessments provides a robust aggregation strategy that tolerates a bounded proportion of incorrect or malicious input, consistent with the fault tolerance guarantees of majority-based consensus theory.

4.2.2 Trust-Weighted Voting

While majority-based voting provides a sound foundation for aggregation, unweighted voting alone is susceptible to several of the threats identified in the attack tree analysis (Figure 5). In particular, Profile Injection attacks—where an adversary mass-creates organizations or projects—and collusion strategies can compromise a simple majority vote by inflating the number of assessments from adversarial entities. To address this, the proposed algorithm augments the majority-based voting mechanism with a trust score that assigns a weight to each vote based on the trustworthiness of the submitting entity.

In the proposed system, trust scores are assigned at the level of organizations and projects. Each organization and each project registered within DevGuard is associated with a trust score, expressed as a floating-point value. When an entity submits a VEX rule assessment, the trust score of that entity is applied as a weighting factor in the voting aggregation. Consequently, assessments from entities with higher trust scores exert greater influence on the final recommendation, while assessments from low-trust entities have proportionally reduced impact. Even if colluding or mass-created low-trust entities outnumber honest participants in absolute terms, their diminished aggregate weight should prevent them from overriding the assessments of trusted contributors.

The trust score thus transforms the algorithm from a simple majority vote into a *trust-weighted majority vote*. For a given CVE and dependency context, let $V = \{v_1, v_2, \ldots, v_n\}$ denote the set of n submitted VEX assessments, where each v_i corresponds to a particular outcome (path), and let $t_i \in [0, 1]$ denote the trust score of the entity that submitted v_i. The aggregated weight for a candidate outcome o is defined as:

$$W(o) = \sum_{i=1, v_i=o}^{n} t_i$$

That is, $W(o)$ sums the trust scores of all assessments that voted for outcome o. The recommended outcome is the one that maximizes $W(o)$ across all possible assessment outcomes, subject to the constraint that a minimum number of votes has been received (mitigation 15 in Table 4). This threshold prevents the system from issuing recommendations when too few voters have participated—whether due to a cold-start phase for newly disclosed vulnerabilities or due to honest voters having been removed from the system—thereby ensuring that recommendations are only issued when they rest on a sufficiently broad basis of community input. In the case of a tie, deterministic tie-breaking rules are applied in accordance with mitigation 31, ensuring reproducible behavior regardless of input ordering or floating-point edge cases.

4.2.3 Trust Score Management

A critical design consideration is the mechanism by which trust scores are determined and maintained. Two principal strategies are conceivable: (1) *explicit assignment*, where an administrator manually sets the trust score based on organizational reputation, verified identity, or other qualitative criteria; and (2) *implicit derivation*, where the trust score is computed algorithmically based on an entity's historical behavior, accuracy track record, or community endorsement patterns.

While implicit trust score derivation is conceptually appealing and would enable the system to adapt dynamically to changing participant behavior, identifying the appropriate criteria, data sources, and computational methods for reliable implicit trust derivation constitutes a substantial research problem in its own right. The selection of suitable features—such as historical assessment accuracy, contribution frequency, domain expertise indicators, or peer endorsement signals— each require careful investigation that extends beyond the scope of this work. Furthermore, implicit trust mechanisms introduce their own attack surface, as adversaries may attempt to artificially inflate their trust scores through strategic behavior patterns, which would necessitate additional safeguards. Mitigation 11 from Table 4, which proposes valuing trust based on organization age, exemplifies the kind of implicit signal that could inform future derivation mechanisms.

Therefore, this work employs explicitly assigned trust scores as the initial approach. Administrators within DevGuard can assign and modify trust scores for organizations and projects through the platform's management interface, providing direct and auditable control over the weighting of contributions. This design choice ensures that the trust mechanism remains transparent and resistant to automated manipulation during the initial deployment phase. The exploration of implicit trust score derivation mechanisms, including the identification of robust criteria and the development of appropriate validation methodologies for automatic trust assessment, is deferred to future work.

4.3 Crowdsourced Vexing

The preceding sections have established the core components of the proposed system: a threat model that identifies adversarial strategies and derives concrete mitigations (Table 4), a majority-based voting foundation grounded in BFT-inspired consensus principles, a trust-weighted aggregation mechanism, and an explicitly managed trust score framework. Taken together, these components constitute a system in which multiple independent organizations submit VEX assessments for identical vulnerabilities, and an algorithm aggregates these assessments—weighted by the trustworthiness of each contributing entity—into a single, community-endorsed recommendation for a given CVE and dependency context.

In practice, many organizations independently analyze the exploitability of identical vulnerabilities affecting widely used dependencies, resulting in substantial redundant effort across the software supply chain [18]. Since the dependency tree of a project dictates the chain of decisions a maintainer must apply, organizations with similar dependency structures deterministically arrive at similar conclusions. The system defined in the preceding sections directly targets this redundancy: by collecting individual VEX rule assessments and consolidating them through the trust-weighted majority voting algorithm, isolated, per-organization vexing activities are transformed into a shared and reusable knowledge base from which community-level recommendations can be derived. The in-scope threat mitigations—including trust score weighting (mitigations 8, 11, 13), minimum voter thresholds (mitigation 15), vote deduplication (mitigation 20), and deterministic tie-breaking (mitigation 31)—ensure that this aggregation process remains resilient against adversarial manipulation.

This approach—the aggregation of distributed VEX assessments into collectively validated, trust-weighted recommendations—will hereafter be referred to as *Crowdsourced Vexing (CV)*, and the implementation of this concept within the DevGuard framework constitutes the central objective of this work [18].

5 Implementation

Having established the algorithmic foundation and the threat model in the preceding sections, this section describes how the proposed CV system was realized within the DevGuard framework. The implementation translates the system definition—comprising the trust-weighted majority voting algorithm, the in-scope mitigations derived from the attack tree analysis (Table 4), and the data parameters stipulated by DevGuard (Table 2)—into a functional component integrated into the DevGuard backend. The following subsections detail how each mitigation was incorporated into the algorithm, describe the overall processing logic, and assess the status of the out-of-scope mitigations within the DevGuard ecosystem.

5.1 Realization of the Algorithm

The CV algorithm was implemented as a self-contained module within the DevGuard backend using the Go programming language. The module exposes a single entry point that accepts the dependency path of the requesting project, the CVE under evaluation, the complete set of existing VEX rules from the database, and the corresponding organizational and project metadata—including trust scores, creation dates, and ownership information. From these inputs, the algorithm produces either a recommended VEX rule, representing the crowd-endorsed assessment for the given vulnerability and dependency context, or an error indicating that the conditions for issuing a recommendation are not met. Figure 6 shows that this customized approach creates a compatible mapping for DevGuard.

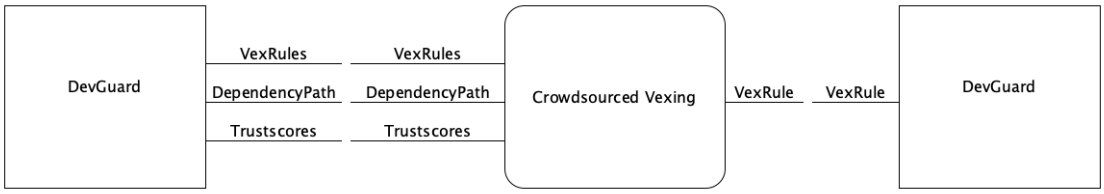

Figure 6: Mapping between DevGuard and Crowdsourced Vexing

The processing logic proceeds through several distinct phases: entity resolution, filtering and validation, trust-weighted vote aggregation, and outcome selection. Each phase incorporates one or more of the in-scope mitigations identified in Table 4, ensuring that the resulting recommendation is resilient against the adversarial strategies modeled in the attack tree (Figure 5). The following subsections describe the algorithm's behavior, organized by the mitigation each component addresses.

5.1.1 Entity Resolution and Path Matching

For each VEX rule in the input set, the algorithm first resolves the chain of ownership by tracing the rule's asset identifier to its parent project and, subsequently, to the owning organization. This resolution is necessary because trust scores are assigned at the organization and project level, and because deduplication and age-based filtering operate on these entities rather than on individual VEX rules. If any link in the chain cannot be resolved—for instance, if the asset, project, or organization record is absent from the input data—the rule is discarded and logged, ensuring that incomplete or orphaned data does not corrupt the aggregation.

Following entity resolution, the algorithm evaluates whether the VEX rule's path pattern matches the dependency path of the requesting project. The path pattern is expressed as an ordered sequence of dependency identifiers, optionally containing wildcard elements. Matching is performed using a suffix-based approach, provided through an algorithm by DevGuard: the algorithm tests the pattern against all trailing sub-sequences of the dependency path, starting from the shortest suffix of length equal to the number of literal (non-wildcard) elements in the pattern and extending towards the full path. A match is declared if the pattern can be aligned to any such suffix. The wildcard elements—specifically, the reserved token $*$—matches zero or more consecutive path elements, permitting patterns to abstract over multiple intermediate dependencies without enumerating them explicitly. Rules whose path patterns do not match the dependency path under this procedure are excluded from further consideration, as they pertain to dependency contexts that are

not relevant to the requesting project. This filtering step inherently prevents adversaries from injecting votes through VEX rules that reference fabricated or unrelated dependency paths.

5.1.2 Minimum Organization Age (Mitigations 10 and 11)

Before a VEX rule is admitted into the voting pool, the algorithm verifies that the owning organization satisfies a minimum age requirement. Organizations whose creation date falls within a configurable threshold—set to 30 days in the current implementation—are excluded from participation. This measure directly addresses mitigations 10 and 11 from Table 4: it prevents freshly created organizations from immediately influencing recommendations, thereby imposing a temporal barrier against mass-creation attacks where an adversary rapidly registers multiple organizations to inflate voting power. The age threshold reflects the principle that organizations with a longer operational history carry a higher baseline of credibility, as the effort required to maintain an organization over an extended period raises the cost of sustaining a coordinated attack. While mitigation 11 envisions a more nuanced trust valuation based on organization age, the current implementation treats age as a binary gate; extending this to a continuous, age-dependent trust factor is deferred to future iterations, consistent with the trust score management strategy outlined in Section 4.2.3.

5.1.3 Input Validation (Mitigation 30)

Each VEX rule's assessment field is validated against a fixed set of permissible values—specifically, the assessments `false-positive` and `affected`. Rules carrying any other assessment value, including empty strings, case variants, or arbitrary text, are rejected and excluded from the voting process. This strict input validation implements mitigation 30 from Table 4, which requires choosable options instead of free-form input, effectively eliminating the possibility of exploiting input validation flaws or injecting unexpected values into the aggregation logic. By restricting the assessment domain to a predefined set of options, the algorithm ensures that the voting space remains well-defined and that no adversarial payload can propagate through the assessment field.

5.1.4 Trust-Weighted Vote Aggregation with Exponential Decay (Mitigations 13 and 8)

The aggregation mechanism at the core of the CV algorithm follows a weighted majority voting scheme based around VEX rules as the aggregation index. In contrast to an unweighted majority vote—where each vote contributes equally and the outcome is determined solely by count-the algorithm assigns each vote a confidence value instead, which is derived from two independent, multiplicative components.

The first component is the trust score t, which captures the credibility of the submitting entity and ensures that assessments from established participants carry greater weight than those from unverified or shallow accounts (mitigation 13). Before incorporating a vote into the aggregation, the algorithm will execute the previously mentioned resolution of the trust score for the submitting entity (organization or project). The base trust score for a given vote, expressed as a floating-point value in the range $[0, 1]$, is computed as the maximum of the organization's and the project's trust scores: $t = \max(t_{org}, t_{proj})$. This

formulation ensures that an entity benefits from whichever trust level is higher, accommodating scenarios where a well-regarded organization hosts projects of varying maturity, or where individual projects within a less-established organization have independently earned elevated trust. The second component is a diminishing factor that attenuates repeated contributions from the same user, preventing mass-created organizations from circumventing trust weighting through volume (mitigation 8). The algorithm tracks whether the creator of the submitting organization has already contributed votes through other organizations within the same aggregation context. Since a single user may create multiple organizations, unchecked repeated participation would enable an adversary to overtake voting through mass-created entities with low trust, circumventing the trust weighting by substituting quantity for quality. A vote tracker records how many votes have been contributed by the same user as a creator of organizations across all path-assessment combinations during the current aggregation. Let k denote the number of prior votes that the user has already contributed under the entity identity i at the time the current vote is processed, and let d denote a configurable diminishment constant. The confidence value c_k for each vote is then computed as:

$$c_k = t_i \cdot d^k$$

The first vote from a given user ($k = 0$) receives the full trust score as its confidence, since $d^0 = 1$. Each subsequent vote is attenuated by an additional factor of d: the second vote ($k = 1$) contributes $t_i \cdot d$, the third ($k = 2$) contributes $t_i \cdot d^2$, and so forth. In the current implementation, d is set to 0.1, producing aggressive attenuation in which each additional vote from the same user carries only one tenth of the weight of the preceding one. This rapid decay renders mass-voting strategies ineffective, as the aggregated vote of a user follows under

$$\sum_k c_k$$

and a user's third vote contributes merely 1% of the weight of the first, and a fourth vote only 0.1%.

The use of a diminishment constant offers several advantages. First, the decay behavior is uniform and predictable across all trust levels, simplifying both the analysis and the operational reasoning about the system's resilience to manipulation. Second, the total cumulative influence of any single user, regardless of the number of organizations created, is bound by the convergent geometric series $\sum_{k=0}^{\infty} t \cdot d^k = \frac{t}{1-d}$, which for $d = 0.1$ yields an upper bound of approximately $1.111 \cdot t$. Consequently, this means that even with an unlimited number of organizations, a low-trusted users' aggregated contribution can never materially exceed the weight of a single full-trusted entity's vote, rendering the creation of additional organizations inefficient. On the other side, this implementation still allows for entities of the same trust-level to compete with each other using volume of votes, as small changes within the same relative region, still produce significant impact. For example, a user A with a trust score of 0.3 will not be able to out-vote a user B with a trust score of 0.9 as, per the algorithm, his aggregated contribution would be $0.333333... < 0.9$. In contrast, a user C, also with a trust score of 0.9, could compete with user B by volume since the relative difference between 0.999 and 0.9999 can actually impact the vote.

The resulting confidence values of j votes are summed for n possible candidate outcomes o_n—identified by the combination of path pattern and assessment type in a VEX rule—as

$$o_n = \sum_j \sum_k c_{jk}$$

and the outcome with the highest aggregated weight is selected as the final recommendation.

5.1.5 Vote Deduplication (Mitigation 20)

To prevent replay-style attacks—where the same assessment is submitted multiple times within the same organizational and project context—the algorithm enforces uniqueness constraints on the voter set. DevGuard itself should already prevent this by restricting the creation of VEX rules to unique contexts when using direct DevGuard features, but this does not consider requests that are replayed externally into the system through vulnerabilities. For each path-assessment combination, the algorithm maintains a record of which organization-project-asset combinations have already contributed to a vote. If a subsequent VEX rule, originating from the same combination for the same path and assessment, is encountered, it is silently discarded. This measure implements mitigation 20 from Table 4, which requires that votes are guaranteed unique through a combination of data structure fields. Notably, the deduplication operates at the level of organization-project-asset combination rather than individual VEX rules, preventing an entity from artificially inflating its vote count by creating multiple references for the same path pattern vote.

5.1.6 Minimum Voter Threshold (Mitigation 15)

After all valid votes have been collected and aggregated, the algorithm verifies that the total number of valid votes meets a configurable minimum threshold—set to four in the current implementation. If fewer valid votes have been recorded, the algorithm refuses to issue a recommendation and returns an error instead. This threshold implements mitigation 15 from Table 4, which requires a minimum number of voters for a decision and disables the recommendation when too few voters remain. The threshold serves a dual purpose: during a cold-start phase, when a newly disclosed vulnerability has not yet attracted sufficient community attention, it prevents premature recommendations based on an unrepresentative sample. Similarly, if honest voters have been removed from the system—whether through account compromise or administrative action—the threshold ensures that the remaining voter base is too sparse to produce a reliable recommendation. In such cases, the algorithm defaults to a safe state of non-recommendation rather than emitting a potentially manipulated output.

5.1.7 Deterministic Outcome Selection (Mitigation 31)

The final phase of the algorithm selects the candidate outcome with the highest aggregate trust-weighted value as the recommended VEX rule. The selection iterates over all path-assessment combinations that received at least one valid vote and identifies the one whose accumulated weight is maximal. Candidates are sorted by their aggregate value, and the entry with the greatest accumulated weight is selected as the recommendation. In the event of a tie—where the two highest-ranked candidates share an identical aggregate weight—the algorithm withholds the recommendation entirely and returns an empty result, rather than imposing an arbitrary ordering between equally supported outcomes.

This conservative tie-breaking strategy reflects the principle that an inconclusive vote does not constitute sufficient evidence to endorse any particular assessment. Selecting one of the tied candidates arbitrarily would introduce a bias that is not grounded in the available evidence and could mislead users into acting on an assessment that carries no stronger support than its competing alternative. Instead, the absence of a recommendation signals to the requesting system that manual assessment should be conducted. This also serves as an implicit incentive for the broader community to contribute additional votes that may break the deadlock in future queries, thereby improving the quality of the available data over time.

This approach addresses mitigation 31 from Table 4, which calls for a standardized cutoff and deterministic tie-breaking rules. By defining the tie-breaking rule as a deliberate non-recommendation, the algorithm guarantees a deterministic and consistent outcome in all cases: given the same set of input votes, the algorithm will always either return the uniquely highest-weighted candidate or produce no recommendation, without any dependence on evaluation order or arbitrary ordering heuristics.

5.2 Out-of-Scope Mitigations

The mitigations classified as out of scope in Table 4 represent platform-level requirements that cannot be enforced within the CV algorithm itself but are essential to the overall security posture of the system. As established in the system definition, the satisfaction of these requirements is assumed for the correctness-guarantees of the proposed algorithm. The following assessment examines which of these mitigations are already provided by the DevGuard framework and which remain unimplemented at the time of writing.

5.2.1 Mitigations Satisfied by the DevGuard Framework

Several out-of-scope mitigations are already satisfied by DevGuard's existing infrastructure and can therefore be considered as fulfilled prerequisites for the CV system.

- **Two-Factor Authentication (Mitigations 1 and 28).** DevGuard supports multi-factor authentication through its integration with Ory Kratos, which provides Time-based One-Time Password (TOTP)-based verification as well as passkey and WebAuthn support. The identity server is configured to require the highest available authentication assurance level for sensitive operations, ensuring that user accounts are protected against credential theft and unauthorized access—the primary attack vectors addressed by these mitigations.
- **Authentication Framework (Mitigation 29).** DevGuard integrates Ory Kratos as its authentication backend rather than maintaining a custom authentication implementation. By delegating user registration, login, session management, and identity verification to this externally maintained and widely scrutinised open-source framework, the attack surface attributable to authentication logic is substantially reduced. Relying on a component that is subject to continuous security review by a broad community of practitioners directly satisfies the requirement of mitigation 29 to employ a well-tested authentication framework that is free from exploitable implementation flaws.
- **Token Expiry (Mitigation 2).** Session and flow lifespans are enforced through the Ory Kratos configuration, with login and registration flows limited to ten-minute

windows and privileged sessions expiring after fifteen minutes. These constraints limit the temporal window during which stolen session tokens remain valid, directly addressing the threat of token-based account hijacking.

- **Account Recovery (Mitigation 5).** DevGuard provides account recovery functionality through Ory Kratos, which supports code-based recovery via email. This mechanism enables legitimate users to regain access to compromised accounts, limiting the duration during which an attacker can exercise unauthorized control over a voter identity.

- **User and Organization Verification (Mitigation 12).** DevGuard integrates with external identity providers through OpenID Connect (OIDC), including GitLab OAuth2 support. This external verification mechanism raises the barrier for mass-creating fraudulent user accounts, as each account must be backed by a verifiable external identity, supplementing the in-scope mitigations against Profile Injection attacks.

- **System Standards and Guidelines (Mitigations 3 and 4).** DevGuard provides a compliance module based on Open Policy Agent (OPA) and Rego policies that evaluates assets against established security frameworks. While this module does not constitute dedicated best-practice guidelines for environment security and setup in the traditional sense, it provides a structured mechanism for enforcing and evaluating compliance standards across projects, partially satisfying the intent of these mitigations.

- **TLS Enforcement (Mitigation 19).** DevGuard's production configuration mandates TLS-encrypted connections for its instance domain. While TLS termination is handled at the infrastructure layer through a reverse proxy rather than within the application code itself, the deployment architecture ensures that communication between clients and the server is encrypted in transit, mitigating man-in-the-middle attacks on the communication channel.

- **Message Signing (Mitigation 18).** DevGuard implements HTTP message signing using Elliptic Curve Digital Signature Algorithm (ECDSA) with the P-256 curve, applying digital signatures to outgoing requests that include a content digest and method field. Incoming requests are verified against these signatures in the session middleware, ensuring that any modification of message content between the client and the server is detectable. This mechanism provides an integrity guarantee beyond transport-layer encryption, protecting against scenarios in which TLS termination occurs at an intermediate proxy and thereby satisfying mitigation 18's requirement to prevent man-in-the-middle attacks from undetectably altering vote messages in transit.

- **Rate Limiting (Mitigations 21–26).** DevGuard implements rate limiting for outgoing Application Programming Interface (API) calls to external services and enforces synchronization cooldowns for external entity provider interactions. While comprehensive incoming request rate limiting is not yet deployed as a general middleware, the existing mechanisms provide a foundation for DoS protection that can be extended to cover the CV endpoints specifically.

- **Visual Indicators for Crucial Actions (Mitigation 16).** The DevGuard frontend does not yet provide visual indicators specifically tailored to CV recommendations. However, it already employs extensive visual cueing mechanisms throughout the vulnerability management workflow that constitute a strong foundation for this mitigation. Severity badges use a color-coded scheme—red for critical, orange for

high, yellow for medium, and green for low—to communicate risk levels at a glance. Vulnerability state badges display distinct icons and colors for each state: an open bug icon for unresolved vulnerabilities, a stop icon with purple background for false positives, a speaker-muted icon for accepted risks, and a green check icon for fixed issues. Furthermore, nodes within the dependency graph visualization are highlighted with red borders and a red "Vulnerable" badge when they carry unresolved vulnerabilities, while nodes marked as false positives are rendered with gray borders and a corresponding gray badge. Although these visual indicators are not yet applied to CV-specific recommendation actions, the existing infrastructure demonstrates that the DevGuard frontend already provides the foundational components necessary to implement mitigation 16 for the CV context specifically.

- **Multi-Step Dialogue for Risk Acceptance (Mitigation 17).** The DevGuard frontend already implements structured confirmation workflows for vulnerability assessment actions that substantially satisfy the intent of this mitigation. When a user marks a vulnerability as a false positive, a dedicated dialog is presented that requires multiple deliberate interactions: the user must select a mechanical justification from a predefined set of options—such as "the vulnerable component is not part of the product," "the vulnerable code was excluded," or "the vulnerable code is never executed"—provide a written justification, and optionally select a path pattern for the scope of the VEX rule, before the submission can be finalized. Similarly, when accepting a risk, a separate dialog explicitly warns the user that they are acknowledging the vulnerability and its potential impact, requires a written justification, and links to external documentation for further guidance. These multi-step confirmation flows prevent uninformed or accidental acceptance actions by requiring users to consciously engage with the implications of their decision at multiple stages. While these dialogs are not yet integrated with the CV recommendation acceptance workflow specifically, they demonstrate that DevGuard already possesses the User Interface (UI) components and interaction patterns necessary to fulfill mitigation 17 for crowdsourced recommendations.

5.2.2 Mitigations Not Yet Present in DevGuard

The following mitigations are not yet implemented in the DevGuard framework at the time of writing. These represent requirements that are assumed to be satisfied for the security guarantees established in the system definition and are expected to be addressed by developers of the DevGuard platform in the future.

- **Emergency Account Lockdown (Mitigation 6).** DevGuard does not currently provide a mechanism for users or administrators to immediately suspend a compromised account. Implementing such a feature would enable rapid containment of hijacked voter identities before they can influence ongoing or future CV recommendations. This is considered a necessary addition for production readiness of the CV system.
- **Organization and Project Creation Limits (Mitigation 7).** No quota or rate limit is currently enforced on the number of organizations or projects a single user can create. While the in-scope diminishing returns mechanism (mitigation 8) reduces the marginal value of additional votes from the same user, imposing explicit creation limits at the platform level would provide an additional layer of defense against mass-creation attacks.

- **CAPTCHA Integration (Mitigation 9).** Automated bot-driven account creation is not currently prevented through CAPTCHA or equivalent challenge-response mechanisms. Integrating such a mechanism into the registration flow would complement the minimum organization age requirement (mitigation 10) by preventing the automated creation of dormant accounts that could later be activated for coordinated voting attacks.
- **Multi-Factor Checks for Account Disabling (Mitigations 14 and 27).** DevGuard does not currently require additional authentication factors before disabling user accounts or forcing token expiration. Without such safeguards, an attacker who gains limited administrative access could selectively remove honest voters from the system, thereby shifting the voting balance in favor of adversarial entities. Implementing multi-factor confirmation for these sensitive operations would close this attack path.

While the absence of these mitigations does not compromise the algorithmic integrity of the CV system itself, their implementation would strengthen the platform-level defenses that the algorithm relies upon. As the DevGuard framework continues to mature, the integration of these missing mitigations should be prioritized to ensure that the assumptions underpinning the CV security model are fully satisfied in practice.

6 Evaluation

This section evaluates the correctness and security properties of the CV implementation described in the preceding section. The evaluation proceeds in two parts. First, the test suite developed alongside the implementation is assessed with respect to its coverage of the algorithm's behavioral requirements and the in-scope mitigations derived from the attack tree analysis. Second, a number of design-level concerns that were identified during implementation are discussed, with particular attention to the validity of the voting model and the numerical limitations of the chosen attenuation scheme, to direct future development efforts.

6.1 Tests

To verify that the implementation correctly realizes the intended algorithmic behavior and satisfies the security properties stipulated by the in-scope mitigations in Table 4, a dedicated test suite was developed covering three categories of test cases: behavioral correctness tests, security and mitigation tests, and edge case tests. The complete suite comprises 52 individual test cases, grouped into 21 top-level test functions and organized by concern. All 52 cases passed successfully during execution. It should be noted that, although these tests constitute a comprehensive set intended to support a structured evaluation of the system, the possibility of undetected cases cannot be entirely excluded. Consequently, additional tests should be considered in future work to further enhance the robustness and completeness of the assessment.

6.1.1 Behavioral Correctness Tests

The first category of tests verifies that the algorithm produces correct outputs under well-formed, representative inputs. Four test groups address distinct aspects of the algorithm's expected functional behavior.

- **Uniform Vote.** Eight cases verify that when all submitted VEX rules converge on a single assessment—either `false-positive` or `affected`—the algorithm returns the correct recommendation across dependency paths of varying depth, from shallow two-element paths to seven-element chains. This confirms that the basic voting and path-matching mechanisms operate deterministically regardless of structural path complexity.
- **Higher Trust Score Wins.** Six parametric cases verify that, given two equally sized voter groups holding opposing assessments, the group with the higher trust score consistently prevails. Pairs of trust scores spanning the full range of the $[0, 1]$ domain were tested, including marginal differences such as scores of 0.50 and 0.51, confirming that the weighting mechanism is sensitive to small trust differentials and does not exhibit rounding-induced misclassification at the fringes of the scale.
- **Maximum of Organization and Project Trust Score.** A single case validates the trust score selection semantics: the algorithm must use the maximum of the organization's and the project's trust score when computing vote confidence. The test constructs a scenario where relying solely on the organization trust score would invert the correct outcome, ensuring that the implementation does not silently fall back to organization-level trust in all cases.
- **Quality versus Quantity.** Two cases probe the interaction between trust-based weighting and vote volume: four high-trust voters ($t = 0.9$) accumulate a combined weight of 3.6, which exceeds the weight of ten low-trust voters ($t = 0.1$, combined weight 1.0), confirming that quality can dominate quantity. Conversely, ten moderate-trust voters ($t = 0.3$, combined weight 3.0) are shown to outweigh four high-trust voters ($t = 0.5$, combined weight 2.0), demonstrating that sufficient quantity can legitimately overcome a quality deficit within trust-comparable ranges.

6.1.2 Security and Mitigation Tests

The second category directly tests the in-scope mitigations listed in Table 4. These tests are intentionally adversarial in structure: each case constructs an input that would produce an incorrect or exploitable result if the corresponding mitigation were absent.

- **Minimum Organization Age (Mitigations 10 and 11).** Two cases verify the temporal admission gate. In the first, four high-trust organizations aged seven days—falling below the 30-day threshold—submit votes for the `affected` assessment. Despite their high trust score, these votes are discarded, and the algorithm correctly returns the `false-positive` recommendation produced by the four older, lower-trust organizations. The second case targets the boundary condition: organizations created exactly at the threshold (aged 30 days minus one minute) are accepted, confirming that the comparison is inclusive and that the cutoff does not inadvertently exclude eligible entities at the margin.

- **Minimum Voter Threshold (Mitigation 15).** Three cases validate the cold-start and voter-depletion safeguard. Presenting exactly the minimum of four valid votes results in a successful recommendation. Presenting three votes causes the algorithm to return an error and presenting zero votes likewise results in an error. These cases confirm that the threshold acts as a hard gate regardless of trust levels.

- **Replay Protection (Mitigation 20).** Three cases examine deduplication behavior. When the same organization-project-asset combination submits five identical VEX rules through distinct assets, only one vote is recorded, causing the total count to fall below the threshold and triggering an error. A complementary case demonstrates that five distinct projects within the same organization each contribute independently, confirming that deduplication operates at the organization-project-asset combination level rather than the organization level alone. A third case verifies that identical deduplication semantics apply on deep dependency paths, ruling out path-length-dependent anomalies in the voter bookkeeping logic.

- **Assessment Input Validation (Mitigation 30).** Seven cases submit rules carrying invalid assessment values: an empty string, the label `not-affected`, the capitalized variants `AFFECTED` and `False-Positive`, a generic malformed string, a cross-site scripting payload, and a SQL injection attempt. It should be noted here that not all of these cases intend to propose a serious threat to the system but rather test if the input has to stay within the set bounds. In every case, no valid votes are accumulated and the algorithm returns an error, confirming that the assessment filter is strictly enforced and that no adversarial input can be injected into the vote aggregation through the assessment field.

- **Trust Score Weighting (Mitigation 13).** A single case verifies that organizations carrying a trust score of zero contribute zero weight to the vote, ensuring that shallow accounts receive no influence over the recommendation even when present in sufficient numbers to satisfy the voter threshold structurally.

- **Negative Trust Scores.** A complementary case verifies that negative trust scores—which would represent a configuration error—are detected and rejected by the algorithm before any aggregation takes place, preventing adversarially crafted input data from introducing negative vote weights that could invert aggregation outcomes.

- **Deterministic Tie-Breaking (Mitigation 31).** A single case constructs a symmetric input: two voter groups of equal size and equal trust score vote for opposing assessments, producing identical aggregate weights. The algorithm is verified to return an empty recommendation—that is, neither assessment is endorsed—rather than selecting arbitrarily between the two, confirming that the conservative tie-breaking policy is enforced.

- **Diminishing Returns (Mitigation 8).** Two cases validate the exponential decay mechanism. The first constructs a scenario of five same-creator organizations voting for one assessment against five distinct-creator organizations voting for the opposing assessment: the distinct-creator group accumulates a strictly higher aggregate weight because each of its members contributes an unattenuated first vote, while the same-creator group's contributions decay geometrically. The second case confirms that users on the same trust-level can still compete with each other using vote volume, by showing that different amounts of votes with the same trust score, shift the outcome in favor for the larger aggregation.

- **Unrelated Paths Ignored.** Two cases verify that votes associated with VEX rules whose path patterns do not match the queried dependency path are excluded

from the aggregation entirely. Both shallow and deep path scenarios are exercised, confirming that path filtering is applied uniformly regardless of the structural length of the queried path.

6.1.3 Edge Case Tests

The third category addresses boundary conditions and outlier inputs that may arise in practice.

- **CVE Mismatch.** A case confirms that VEX rules referencing a different CVE identifier than the queried one produce no valid votes, ensuring that the algorithm cannot be influenced by data belonging to unrelated vulnerability records.
- **Missing Entities.** Four cases verify graceful handling of incomplete input data: absent assets, absent projects, and absent organizations each cause the corresponding VEX rules to be silently discarded, while passing an empty rule set produces an error. No panic or underdefined behavior occurs in any of these scenarios.
- **Path Non-Matching.** Two cases exercise partial-match semantics: a scenario where all votes target a non-matching path results in an error, while a scenario where only a subset of votes fail to match still produces the correct recommendation from the remaining valid matches, provided the minimum threshold is met.
- **Branched Dependency Paths.** Two cases ensure that votes for one branch of a dependency tree do not influence recommendations for a structurally distinct branch sharing a common ancestor. The algorithm is further verified to handle independent recommendation queries for multiple branches without interference, confirming correctness across concurrent scoping requests.
- **Very Small Trust Scores.** A stress case presents 100 voters with trust score 0.01 opposing a single voter with trust score 0.99. The combined weight of the low-trust cohort ($100 \times 0.01 = 1.00$) exceeds the single high-trust voter's weight of 0.99, and the algorithm correctly returns the majority assessment. This confirms that the system remains functional even when trust scores are minimal, while also demonstrating that mass-creation attacks by a sufficient number of collaborating low-trust actors are not fully preventable by trust weighting alone. The test thereby serves as an empirical lower bound on the trust differential required to resist a coordinated low-trust coalition.

6.1.4 Security Assessment

Based on the test results, the implemented CV algorithm satisfies all in-scope security mitigations derived from the attack tree analysis. The combination of the minimum organization age filter, the minimum voter threshold, the trust-weighted confidence computation, the exponential decay mechanism for same-creator organizations, the deduplication of organization-project-asset combinations, strict assessment input validation, and the conservative tie-breaking policy collectively address the threat vectors modeled in Figure 5. Every adversarial test case produced the expected safe outcome—either the correct assessment, an empty non-recommendation, or an explicit error—without the algorithm emitting a manipulated or undefined result.

As an additional empirical indicator at platform level, the codebase was also evaluated by the DevGuard scanner pipeline during repository push events. According to the scanner implementation and documentation, this pipeline covers software composition and container dependency analysis (Trivy-based), static source-code analysis (Semgrep-based), infrastructure configuration analysis (Checkov-based), and secret leakage detection (Gitleaks-based) [17]. For the evaluated revision, these scans produced no findings. Although a zero-finding result cannot be interpreted as a formal proof of absence of vulnerabilities, it provides orthogonal evidence that no known issues detectable by the configured scanner stack were present at scan time.

The most critical residual risk surface lies in the interplay between trust scores and collaborative adversarial behavior. As the very-small-trust-score edge case demonstrates, a coalition of sufficiently many individual low-trust actors can overpower even a high-trust honest voter when their combined weight exceeds the honest participant's individual contribution. The convergence bound imposed by the exponential decay mechanism applies only within the scope of a single creator's organizations; it does not constrain collusion between distinct creators. Mitigating this attack vector to the degree required for full resistance would require complementary platform-level controls—such as the organization creation limits identified as mitigation 7 in Table 4 and the external verification requirement of mitigation 12—which remain unimplemented at the time of writing. Though this is a crucial concern, it does not render the system nor the trust score redundant, as the application can still serve as an impediment for attackers, while trusted entities accumulate in size to maintain the validity of the system.

6.2 Concerns for future development

Beyond the properties evaluated through the test suite, two design-level concerns warrant discussion: the validity of applying VEX rules as voting acts, and the numerical precision constraints of the chosen diminishment scheme.

6.2.1 Application of VEX Rules as Votes

The current model treats every VEX rule created by a user—whether authored independently or adopted from a crowd-sourced recommendation—as a vote that contributes to future aggregations involving the same dependency path and CVE. This raises a concern about self-reinforcement: if a user accepts a recommendation and applies it, that application creates a new VEX rule that feeds back into the pool of votes, effectively numerically reinforcing the recommendation that was already dominant. Over time, this feedback loop could amplify a majority position to the point where dissenting assessments become arithmetically irrelevant, even if additional evidence emerges that the dominant assessment was incorrect.

However, closer inspection of the trust-weighting mechanics suggests that this concern is mitigated to a substantial degree by the structural properties of the trust score distribution. A user who accepts a crowd-sourced recommendation and applies it thereby contributes a vote weighted by their own trust score. If that user's trust score is low—which is likely for a new or inactive participant—the feedback contribution is correspondingly small and cannot materially shift the aggregate. If, conversely, the user holds a high trust score, this implies a demonstrated history of reliable security assessments within the platform. It is therefore reasonable to assume that such a user would not blindly adopt a recommen-

dation without independent verification, and that their vote reflects genuine concurrence rather than uncritical acceptance. Under this assumption, VEX rule applications by high-trust users would function as valid, independently grounded endorsements rather than as mere echoes of the recommendation engine. Nevertheless, this reasoning rests on a behavioral assumption rather than a structural guarantee, and a more conservative design could exclude applied recommendations from the vote pool—or weight them with a reduced confidence factor—to avoid any feedback-induced bias. Furthermore, self-reinforcing rules assume an initial state where users are presented with only a few contradicting options, so that selection and adoption of a singly existing rule become more trivial. Considering that there are existing mitigations in the system, which counteract a marginal recommendation volume (e.g. minimum voter threshold), this assumption is not always fulfilled and the risk for this scenario is reduced.

6.2.2 Numerical Precision of the Exponential Decay Mechanism

The exponential decay mechanism computes confidence values of the form $c_k = t_i \cdot d^k$, where k is the number of prior votes by the same creator and $d = 0.1$ is the diminishment constant. For large values of k, this expression produces numbers in the range of floating-point underflow: a creator's tenth vote contributes a weight of $t \cdot 10^{-9}$, which, for a trust score of $t = 0.9$, evaluates to approximately 9×10^{-10}. While IEEE 754 double-precision arithmetic can represent values as small as approximately 5×10^{-324}, the effective precision of such numbers is limited near the underflow boundary, and accumulated rounding errors may introduce non-determinism when comparing aggregate weights that differ only by contributions from very high k values.

This concern is most acute in configurations involving many organizations created by the same user—a scenario that the diminishing returns mechanism itself is designed to discourage. In practice, the aggressive decay rate of $d = 0.1$ renders contributions beyond the fifth or sixth vote negligible relative to the overall aggregate weight, and the test suite did not reveal any observable rounding-induced misclassification. Nevertheless, the numerical behavior in the extreme has not been formally verified, and it represents a latent risk in deployments where adversaries deliberately construct inputs at the boundary of numerical precision to provoke inconsistent outcomes.

6.2.3 Deterministic Tie-Breaking and Informational Deadlock

The conservative tie-breaking policy— whereby the algorithm withholds any recommendation whenever the two highest-ranked candidates share identical aggregate weight—guarantees determinism and prevents the arbitrary imposition of an unsupported assessment. However, this policy carries an operational cost: when a deadlock occurs, the requesting user receives no actionable output, despite the fact that a significant body of community data may be present in the system. This absence of guidance can be disorienting, particularly for inexperienced users who may not understand why a recommendation is unavailable when many VEX rules for the queried vulnerability already exist.

Moreover, a strict tie can occur not only under conditions of sparse data, where few voters have converged yet, but also under data-rich conditions, if the trust-weighted contributions of opposing assessments happen to cancel exactly. While the probability of an exact floating-point tie decreases as the number of voters grows, it remains non-vanishing for any finite participant population, and it can be reproduced deterministically by an adversary

with knowledge of the trust score distribution. In the worst case, an adversary who knows the current aggregate weights for competing assessments could introduce precisely calibrated votes to force and perpetuate a deadlock, preventing the system from ever issuing a recommendation.

A possible mitigation for this scenario would be to introduce a tiebreaker heuristic—for example, preferring the `affected` assessment over `false-positive` in the event of an exact tie, under the principle that a conservative security posture favors treating a vulnerability as present over treating it as absent. However, as discussed in the implementation section, introducing such a bias would imply endorsing an assessment that carries no stronger evidential support than its competitor, which conflicts with the epistemological goals of the system. The previously mentioned decision for this problem—incentivizing the user to conduct a manual assessment—still remains the most viable option.

7 Conclusion

This work has addressed the problem of redundant vulnerability assessment effort distributed across independently maintained software projects. Motivated by the observation that projects with structurally similar dependency trees will deterministically arrive at identical VEX assessments, a CV system was designed and implemented as a PoC within the DevGuard framework. The goal of this system is to aggregate individually authored VEX rules into collectively validated, trust-weighted recommendations, thereby transforming isolated per-organization vexing activities into a shared and reusable knowledge base.

To realize this goal, the work proceeded through three distinct phases. First, a practical system analysis revealed that the TB KB CF Hybrid RS approach proposed in prior work [16] introduces irreconcilable discrepancies with the DevGuard framework, most critically the absence of a rating mechanism for VEX rules that would be required for CF. This finding motivated the adoption of a simpler, more compatible algorithm: a trust-weighted majority voting scheme grounded in BFT-inspired consensus principles. Second, a threat model was constructed through an attack tree analysis identifying adversarial strategies against the voting process, from which a set of in-scope mitigations, those which can be implemented within the main algorithm, and out-of-scope mitigations, those which have to be realized externally through the DevGuard system, were derived. The algorithm was designed to incorporate all in-scope mitigations—including trust score weighting with exponential decay, minimum organization age filtering, minimum voter thresholds, vote deduplication, strict assessment input validation, and deterministic tie-breaking. Third, the algorithm was implemented as a self-contained module within the DevGuard backend, and its correctness was verified through a dedicated test suite comprising 52 test cases organized across behavioral, security, and edge-case categories. All 52 cases passed successfully, confirming that the implementation satisfies both its functional requirements and the security properties derived from the attack tree analysis. Notably, a residual risk remains, as the test suite can not cover all imaginable cases.

Beyond the test results, the evaluation identified multiple design-level concerns for future development. First, treating every VEX rule application as a vote creates a potential self-reinforcing feedback loop in which a dominant assessment becomes arithmetically entrenched. While trust weighting substantially attenuates this risk, the reasoning rests on a behavioral assumption rather than a structural guarantee. Second, the exponential decay formulation produces confidence values of the form $c_k = t \cdot d^k$ that decrease rapidly to-

ward the floating-point underflow boundary for creators with many organizations, where accumulated rounding errors may introduce non-determinism when comparing aggregate weights that differ only in contributions from very high k values. Third, the conservative tie-breaking policy, while epistemologically sound, opens the possibility of adversarially induced deadlock, where a well-informed actor introduces calibrated votes to permanently suppress a recommendation. Although deliberatly chosen to incentivise interaction with the system to gather more data tie-breaking data, future iterations should evaluate whether additional safeguards against tie induction are warranted.

In further considerations, several aspects of the system remain open for future development. The most consequential open question concerns the mechanism for determining trust scores. In the current implementation, trust scores are assigned explicitly by administrators, which ensures transparency and resistance to automated manipulation during the initial deployment phase but does not adapt to changes in participant behavior over time. The development of an implicit, algorithmically derived trust score represents a substantial research problem in its own right: suitable criteria must be identified, validated, and protected against manipulation. Several signals identified in this work point toward candidates for such derivation. Mitigation 11 from the attack tree analysis—which proposes valuing the trust of an organization based on its age—provides an example: organizations with a longer operational history carry a higher baseline credibility, as sustained engagement implies legitimate investment in and continued use of the platform. Further candidate criteria include historical assessment accuracy relative to subsequently confirmed ground truth, contribution frequency and engagement patterns within the vulnerability management workflow, domain expertise indicators such as the volume and diversity of analyzed CVEs, and peer endorsement signals such as the rate at which other high-trust organizations adopt assessments originating from a given entity. Each of these dimensions requires an investigation into its predictive validity and its availability as a data source within the DevGuard framework. Until such a derivation mechanism is established, the explicit assignment model serves as a necessary baseline that future work can build upon.

Beyond trust score derivation, several platform-level mitigations identified in the attack tree analysis remain unimplemented at the time of writing. Organization and project creation limits (mitigation 7), CAPTCHA integration (mitigation 9), and multi-factor checks for account disabling (mitigations 14 and 27) would each strengthen the platform-level defenses that the algorithm relies upon and should be prioritized as the DevGuard framework continues to mature. The implementation of these mitigations, combined with the development of robust implicit trust score criteria, would bring the system closer to full production readiness and significantly reduce the residual risk surfaces identified during the evaluation. But as these features are beyond the scope of this work, they will be left as future tasks for the developers of DevGuard.

The source code of the CV implementation is publicly available as part of the open-source DevGuard project and can be inspected at `https://github.com/l3montree-dev/devguard` [17]. Contributions, issue reports, and further development efforts are welcomed by the project maintainers at L3montree GmbH [12].

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[2] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, vol. 24, no. 12, pp. 21–29, Dec. 1999.

[3] M. Lipow, "Number of faults per line of code," *IEEE Transactions on software Engineering*, no. 4, pp. 437–439, 2006.

[4] A. Kurmus, A. Sorniotti, and R. Kapitza, "Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs," in *Proceedings of the Fourth European Workshop on System Security*, 2011, pp. 1–6.

[5] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong? a qualitative usability study," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 311–328.

[6] M. Tahaei, A. Jenkins, K. Vaniea, and M. Wolters, ""i don't know too much about it": On the security mindsets of computer science students," in *International Workshop on Socio-Technical Aspects in Security and Trust*, Springer, 2019, pp. 27–46.

[7] Cybersecurity and Infrastructure Security Agency (CISA), *Vulnerability exploitability exchange (vex) status justification document*, Accessed: 2026-02-23, 2022. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/VEX_Status_Justification_Jun22.pdf.

[8] C. Guan and K. K. F. Yuen, "The cognitive comparison enhanced hierarchical clustering," *Granular Computing*, vol. 7, pp. 637–655, 3 Jul. 2022, ISSN: 23644974. DOI: 10.1007/S41066-021-00287-X/TABLES/17. [Online]. Available: https://link.springer.com/article/10.1007/s41066-021-00287-x.

[9] Q. Shambour, M. M. Abualhaj, and A. A. Abu-Shareha, "A trust-based recommender system for personalized restaurants recommendation," *International Journal of Electrical and Computer Engineering Systems*, vol. 13, pp. 293–299, 4 Jun. 2022, ISSN: 1847-7003. DOI: 10.32985/IJECES.13.4.5. [Online]. Available: https://ijeces.ferit.hr/index.php/ijeces/article/view/947.

[10] Cybersecurity and Infrastructure Security Agency (CISA), *Minimum requirements for vulnerability exploitability exchange (vex)*, Accessed: 2026-02-23, 2023. [Online]. Available: https://www.cisa.gov/sites/default/files/2023-04/minimum-requirements-for-vex-508c.pdf.

[11] L. Dev, *Devguard*, Accessed: 2025-11-23, 2023. [Online]. Available: https://github.com/l3montree-dev/devguard.

[12] L. Dev, *L3montree*, Accessed: 2025-11-23, 2023. [Online]. Available: https://l3montree.com/.

[13] J. Bruner, *A beginner's guide to attack tree threat modeling*, Accessed: 2026-02-18, 2024. [Online]. Available: https://riskytrees.com/blog/a-beginners-guide-to-attack-tree-threat-modeling.

[14] R. Ghosh, S. De, and M. Mondal, "" i wasn't sure if this is indeed a security risk": Data-driven understanding of security issue reporting in github repositories of open source npm packages," *arXiv preprint arXiv:2506.07728*, 2025.

[15] E. A. Marand, A. Sheikhahmadi, M. Challenger, P. Moradi, and A. Khalilipour, "Recommender systems for unified modeling language and vice versa - a systematic literature review," *IEEE Access*, vol. 13, pp. 23 426–23 460, 2025, ISSN: 21693536. DOI: 10.1109/ACCESS.2025.3535527.

[16] D. T. A. Quach, "Identifying and evaluating suitable crowd-sourcing recommender algorithms for sharing vulnerability management decisions between organizations," Unpublished manuscript, 2025, 2025.

[17] L. Dev, *Devguard*, Accessed: 2026-02-23, 2026. [Online]. Available: https://github.com/l3montree-dev/devguard.

[18] DevGuard Documentation, *Vulnerability mitigation strategies & vex rules*, Accessed: 2026-02-23, 2026. [Online]. Available: https://docs.devguard.org/explanations/vulnerability-management/mitigation-strategies.

[19] DevGuard Project. "Devguard documentation." Accessed: 2026-02-23. (2026), [Online]. Available: https://docs.devguard.org/.

[20] Ethereum Foundation, *Proof-of-stake (pos)*, Accessed: 2026-03-01, 2026. [Online]. Available: https://ethereum.org/developers/docs/consensus-mechanisms/pos/.