



A large scale analysis of code security in public repositories

Ciprian Oprea^{1,2} · Dominic Octavian Grigoruț² · Haralambos Mouratidis^{3,4} · Eftychia Lakka⁶ · Ourania Manta⁷ · Angelos Mavrias⁷ · Marinós Tsantekidis⁸ · Nikolas Filippatos⁹ · George Daniil⁹ · Ionel-Alexandru Gal^{11,12} · Dragoș Gavriluț^{1,10} · Christos Kargatzis⁸ · Sotiris Ioannidis^{5,6}

Received: 30 October 2025 / Accepted: 6 December 2025
© The Author(s) 2026

Abstract

Cybersecurity is a growing concern for the software development sector, as insecure code and insecure software development practices lead to dangerous vulnerabilities in the released products. We propose CONSOLE, designed to boost cybersecurity for software development with an automated, unified platform of tools and services. The proposed framework integrates static and dynamic code analysis tools, real-time protection mechanisms, and adaptive training services. Multiple open-source tools for Static Application Security Testing offer a wide coverage of supported programming languages and types of vulnerabilities that can be detected. The code analysis features of the proposed platform were used in a large scale analysis on 10,855 public repositories. More than 60% of the analyzed repositories contained verified security issues, which highlights the need for automatic code evaluation as a standard practice in the software development pipeline. While popular software projects are usually more secure, having less defects per Megabyte of code, they are not shielded from security issues. The vulnerabilities present in popular projects are likely to have a greater impact in production environments. The large scale analysis results can be used to improve CONSOLE. The paper proposes a strategy for mitigating the False Positives, focusing on prevalent types of alerts and prioritizing them so the developers will be able to examine the most serious issues first. This paper brings awareness on the security defects found in common software projects and proposes the CONSOLE platform as a solution to automatically detect and correct these defects during the software development lifecycle.

Keywords Code security · Vulnerability · Static Application Security Testing · Repository · Software development

1 Introduction

In recent years, the increasing reliance on open-source components and accelerated software development cycles has made software security a growing concern. Security vulnerabilities [19] embedded in source code, often introduced unintentionally, can lead to critical failures, data breaches, and supply chain compromises. The early identification and mitigation of such vulnerabilities are therefore of paramount importance.

Automated code security evaluation, particularly at scale, plays a vital role in addressing this challenge. Manual code reviews and ad hoc use of static analysis tools have proven insufficient for the complexity and velocity of modern software development. Prior research highlights both the utility

✉ Ciprian Oprea
coprisa@bitdefender.com

¹ Bitdefender, Cluj-Napoca, Romania

² Computer Science Department, Technical University of Cluj-Napoca, Cluj-Napoca, Romania

³ Institute for Analytics and Data Science (IADS), University of Essex, Colchester, UK

⁴ Security Labs Consulting (SLC), Dublin, Ireland

⁵ Department of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece

⁶ Institute of Computer Science Foundation for Research and Technology - Hellas, Heraklion, Greece

⁷ CyberAlytics Limited, Nicosia, Cyprus

⁸ AEGIS IT Research GmbH, Braunschweig, Germany

⁹ University of Patras, Patras, Greece

¹⁰ Faculty of Computer Science, Al. I. Cuza University, Iași, Romania

¹¹ Ringhel Team SRL, Bucharest, Romania

¹² Institute of Solid Mechanics of the Romanian Academy, Bucharest, Romania

and limitations of mining GitHub data to assess software security practices, especially with regard to dataset coverage, labelling quality, and reproducibility challenges [6]. Developers and DevOps teams require comprehensive and integrated approaches that not only scan for security issues but also contextualise, prioritise, and help address them across the full lifecycle of a project.

This need is particularly urgent in the context of *incident response and crisis management* [14]. The ability to detect flaws early, before they become entry points for attackers, is critical for increasing resilience. When incidents do occur, having accurate, structured vulnerability data enables faster triage and root-cause analysis. Furthermore, in critical infrastructure environments, such as healthcare, transportation, and energy, the impact of insecure code can be systemic and cascading [24]. Therefore, proactive, large-scale evaluation mechanisms are essential to ensuring continuity and minimising exposure. Additionally, by integrating contextual training based on real-world analysis, platforms like CONSOLE can also support cyber range environments and readiness exercises.

CONSOLE (Cybersecurity for Resilient Software Development) was designed to meet these demands. It is a unified cybersecurity framework that integrates static and dynamic code analysis tools, real-time protection mechanisms, and adaptive training services for developers. Rather than creating yet another tool, CONSOLE orchestrates 17 state-of-the-art open-source security scanners in a cohesive platform that simplifies usage while maximising coverage. 11 out of the 17 tools were used for the current study. Many of these tools, such as *Bandit* [33], *clj-holmes* [17], and *Cppcheck* [26] [21], have been used independently in the past [32], but CONSOLE enables their combined use in a scalable and actionable framework. The platform also leverages standardised vulnerability classifications such as CWE and CVE to improve the traceability and relevance of its findings [27] [29]. It supports 24 programming languages and technologies, with both cloud-based and on-premises deployment options.

Notably, CONSOLE integrates directly with GitHub repositories, enabling fully automated scanning on each commit or pull request, thus embedding security deeply into CI/CD pipelines. This functionality is now live and in use within pilot projects. Developers can trigger scans either by uploading their code manually or by enabling GitHub monitoring via token-based authentication. Each new commit to a monitored branch results in a new security analysis job, with configurable tool-specific settings such as rule exclusions to reduce false positives (e.g., skipping low-severity Bandit tests for Python).

Beyond scanning source code, CONSOLE also monitors development, build, and deployment environments through its integration with Bitdefender GravityZone. This integra-

tion is now active in the pilot deployments and enables real-time threat detection on developer machines by linking local working directories to centralised policy enforcement. Alerts generated by GravityZone are directly forwarded to the CONSOLE dashboard, providing project managers with visibility over infrastructure level risks that may impact the integrity of the development process. These recent developments bring CONSOLE closer to becoming a turn-key DevSecOps solution, ready for deployment in real-world software production environments.

In this paper, we present the architecture and implementation of the CONSOLE platform, along with insights from its application on 10,855 public GitHub projects. Our large-scale evaluation shows the prevalence of security flaws even in widely adopted projects and demonstrates the effectiveness of CONSOLE in identifying and organising security-relevant issues through its clue-based reporting framework. We also analyse false positive trends, tool complementarities, and performance characteristics. These findings underline the importance of automation in ensuring secure-by-design principles and provide practical guidance for incident preparedness in critical infrastructure environments.

The next section presents the related work, with similar code analysis tools and frameworks and with similar studies of the code security in public repositories. Section 3 presents the architecture of the CONSOLE platform, including the requirements analysis and describing the Security Clue concept, a unified format for reporting code security issues. The code analysis module, including the open-source tools used in this study is detailed in Section 4. Our research methodology, presented in Section 5, focuses on evaluating the CONSOLE tools capabilities, mitigating the False Positives and assessing the overall security of the analyzed repositories, including a comparative study between popular and obscure software projects. The experimental results are detailed in Section 6, followed by the conclusions and future work in the last section.

2 Related work

This paper is based on our research dedicated to understanding and mitigating security risks in software development through code analysis. This section reviews key studies that have explored various aspects of code security, from the reliability of patches to vulnerabilities in modern development and deployment practices, all seen from the point of view of how software analysis tools can mitigate code vulnerabilities.

A critical problem, investigated by Meli et al. (2019) [28], is secret leaks in the code. Their analysis showed that exposed credentials were found in more than 100,000 GitHub repositories. It is alarming that 81% of these secrets, such as API keys or private keys, remain accessible for long periods of

time, due to the immutable property of Git history, while detection tools have been shown to be only partially efficient. This represents a direct and persistent vulnerability, often added unintentionally by developers.

In addition to the actual code, modern development practices added new attack vectors. Infrastructure analysis (Iosif et al., 2021) [20] showed hundreds of thousands of security violations in configuration files, such as weak access controls or a lack of encryption. A key aspect is that no static analysis tool can fully cover the software development life cycle, leaving exposed infrastructure. Similarly, pipeline security CI/CD is weak. Koishybayev et al. (2022) [22] discovered that 99,8% of workflows are overprivileged, allowing for extra permissions, while 99,7% are using external built GitHub actions with high security risk, from unverified creators or even using mutable references, opening for supply-chain attacks. Even established practices, such as code review (Thompson & Wagner, 2017) [43], have limited impact, showing a minor correlation with lowering security issues.

In general, even if vulnerability security patch studies (Li & Paxson, 2017; Woo et al., 2005) [25][48] are essential, the specialized literature proves that the biggest and systemic risks are found in exposed credentials, infrastructure configuration, and CI/CD pipelines. The current approach is fragmented, focusing on isolated issues. This lack of holistic vision underscores the urgent need for an integrated and complete analysis tool capable of evaluating multiple sides of code security. Our research paper proposes to fill this gap by developing and validating such a solution.

In [5] the authors propose a pipeline designed to automate several security checks that are required by the DevSecOps procedure as part of the CI/CD flow. Their framework evaluates dependency scanning, container scanning (open ports scanning via *nmap*) and SSL/TLS. Additionally, a set of static tools were also inserted into the CI/CD flow. The entire automation was performed based on GitHub action. While this hybrid approach (static and dynamic) is good for identifying various types of errors, the paper lacks a large set of repositories where the proposed framework could be tested. A similar approach was performed in [23], where the authors integrate Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and Software Composition Analysis (SCA) into CI/CD pipeline. Similarly to the previously cited paper, this lacks a convincing dataset used to validate how effective this framework is and it is more oriented on some specific uses-cases such as retail and e-commerce.

In terms of larger datasets, a notable mention is [8] where more than 900,000 GitHub repositories (and approximately 4.7 millions files) were crawled in order to identify potential SQL injection attacks. While the paper focuses on only one type of vulnerability it does this by analyzing files of different

types (such as PHP, Java and C#) and provide some measurements that highlight the importance of security in CI/CD pipeline (e.g., according to their finding: 63% of the analyzed web application contain SQL string concatenations and 38% of the PHP application were potentially exploitable).

In [4] the authors propose a model-based secure development/testing approach that integrates threat modelling, SAST, DAST, with different MITRE tags into CI/CD pipelines designed for DevSecOps lifecycles. While the proposed model is sound and take into consideration various tools and different types of attacks (such as brute force, spear phishing, credential theft, container escape, etc.), it only validates the methodology with one tool (Juice Shop) and a limited number of security tests (148).

A more comprehensive tests were performed in [12] where the authors evaluated more than 40,000 GitHub repositories from different perspectives including time series to analyze how commits that alter a repository security vary, security alerts analysis as well as static analysis of close to 500,000 repository snapshots written in JavaScript and Python. The study highlights that identifying a potential vulnerability is just part of the process, one also need to act on it and as such the need for an automated workflow that takes all of this into account.

In [3] the author evaluates the efficiency of multiple vulnerability scanners in CI/CD (including container/image scanners and dependency auditing) and discusses what such an implementation implies (including advantages such as cost, proactive security as well as different challenges related to performance, tools compatibility, and false positive and/or negative mitigation). While the paper lacks a solid database where the ideas could be tested, it does come up with an automated flow for different aspects related to the security of CI/CD pipelines.

Another relevant report is [9] where the authors present a comprehensive study that includes best practices for using various security controls as part of a CI/CD pipeline. The report covers various cloud infrastructures, different testing methodologies and standards and provides several ideas on how adding SAST and DAST to DevOps pipelines changes both process and governance. This is meant as technical report, and as such, it does not have a solid dataset to test, however, it highlight how relevant the security for CI/CD pipelines is from both a government and industry point of view.

In [45] the authors conduct a study of the usage of Static Application Security Testing (SAST) tools and identify the main concerns associated with them: high false-positive rates, unclear prioritization, and alert fatigue. The study relies on various academic papers and present a survey based on them. The results show that False Positives and a poor/unclear output are the main detractors when using a SAST tool.

While a substantial body of research has been devoted to the identification of vulnerabilities in smart contracts, the majority of these efforts focus predominantly on blockchain-specific code. Representative examples include Mythril [40], Oyente¹, Slither [11], Echidna [13], and Manticore [30], which employ static analysis, symbolic execution, and property-based testing for Ethereum smart contracts. More recent work has explored formal verification methods, such as Solc-verify [16], and AI-assisted analysis tools, such as GPTScan [42], to detect complex logic vulnerabilities. In contrast, CONSOLE constitutes a generic security analysis platform capable of evaluating smart contracts in conjunction with traditional software projects implemented across multiple programming languages. This comprehensive scope enables CONSOLE to provide unified coverage over heterogeneous codebases, integrating multiple static and dynamic analysis tools to detect a wide range of vulnerabilities, including but not limited to those inherent to smart contracts, thereby underscoring the novelty of our approach relative to prior work that generally focuses on a single language or execution environment. Yet, smart contracts verification design patterns, such as AdapT [15] can improve the predictability of the verification logic for the CONSOLE tools, reducing the redundancy and improving the consistency.

3 The CONSOLE architecture

3.1 CONSOLE requirements

CONSOLE followed a Systems Engineering approach to capture and manage stakeholder requirements and needs, ensuring that they are implemented and validated effectively. The process had two main activities designed to ensure a stakeholder-focused approach. During the first activity, an online questionnaire was developed and distributed to ensure engagement and input from stakeholders that were external to the project, while as part of the second activity, an internal to the project process was followed to understand the needs and requirements of the CONSOLE pilot partners. As part of the first activity, a stakeholders' needs questionnaire was developed (and hosted at the EUSurvey platform), consisting of two sections and 17 questions aiming to capture the main needs of the stakeholders with regards to the CONSOLE objectives. We received 65 response, which confirmed that resilient software development is extremely important for organisations, while at the same time most of them do not have in place staff with relevant expertise, or they do not provide relevant training. The survey also confirmed that most organisations do not use any tools to support resilience as

part of their software engineering processes. These are gaps that CONSOLE platform aims to fulfil.

The second activity focused on eliciting the needs and requirements of the internal partners, and in particular the pilot partners who are potential users of the CONSOLE platform. An initial list of 40 requirements was considered, refined, and checked for duplicates. The process resulted in 33 requirements, which were prioritised following the MOSCOW method. Out of the 33 requirements, 10 were prioritised as Must Have, 13 as Should Have and 10 as Could Have.

3.2 The high-level architecture of CONSOLE

Definition methodology

The architectural definition of the CONSOLE platform followed a comprehensive methodology that integrated research, innovation, and validation activities. This process combined the specification of cybersecurity tools and services, their development by SMEs, and their subsequent validation within pilot scenarios. The approach was user-centric and stakeholder-oriented, ensuring the continuous participation of all relevant actors during design, deployment, and evaluation. Furthermore, the project adopted an iterative and phased development strategy, allowing for incremental refinement based on the feedback of the stakeholders. An agile and market-oriented perspective was also embedded, aligning technical outputs with real operational needs.

Defining the platform architecture required establishing a solid foundation early in the project. This process began with the derivation of a unified set of requirements that captured the full spectrum of functionalities and methodologies that the platform would comprise (see Section 3.1). To ensure completeness, this process incorporated results from state-of-the-art analysis, user and pilot requirements, and qualitative, legal, ethical, and privacy considerations. This knowledge base guided the specification of key architectural aspects, including communication topologies, module interactions, deployment criteria, and user interaction mechanisms.

Each partner contributed to the architectural design by describing their respective components, functional and non-functional requirements, and dependencies, as well as the necessary integration interfaces. The resulting reference architecture serves as a blueprint for implementation, integration, and evaluation across the project. Once established, the architecture guided the specification and development of the components, in close coordination with end-users to ensure alignment with the initial concept and practical validation objectives.

¹ <https://github.com/enzymefinance/oyente>

Platform architecture

The reference architecture of the CONSOLE platform provides a structured view of the core components and their interactions, supporting a unified cybersecurity ecosystem. It defines both functional and communication relationships across modules while enabling secure and scalable operations. Figure 1 presents the reference organization of the platform.

To complement the high-level reference architecture and provide a more formal specification of the solution, a Unified Modeling Language (UML) diagram is also introduced. While Figure 1 offers a holistic conceptual view of the CONSOLE platform, the UML model focuses specifically on the Code Analysis subsystem. This focused representation enables a clearer and more rigorous depiction of the internal service interactions, tool integrations, and data flows that underpin the analysis pipeline, without introducing complexity from unrelated platform modules.

Figure 2 presents the corresponding UML component diagram. It models the structural relationships among the key elements of the Code Analysis module and the interfaces through which findings are communicated to the broader platform. Components related to Endpoint Security and Training are intentionally omitted, as they fall outside the scope of the present work.

The architecture organizes the system into four conceptual blocks, each encapsulating related functionalities:

(i) Management Services (MS): This core block provides the backbone of the platform. It includes Identity Management, CONSOLE Database, API Gateway, and Orchestration services. Together, these ensure coordinated process execution, data management, and secure communication among all platform components. The Identity Management Service enforces Role-Based Access Control (RBAC) across user types; the Database serves as a unified data repository; the API Gateway provides a consistent communication interface; and the Orchestration Service governs workflow execution across modules.

(ii) Incident Central System (ICS): This block encompasses the components responsible for monitoring, detecting, and managing security incidents within the software product ecosystem. It integrates two major subsystems: *Endpoint Security* and *Code Analysis*. The Endpoint Security subsystem monitors developer workstations, build servers, and deployment environments via local agents, producing alerts and feeding them to the dashboard. The Code Analysis subsystem comprises static and dynamic analysis tools, detecting vulnerabilities both at source and runtime levels. A multi-tool approach was adopted to maximise coverage and cross-validate findings, leveraging complementary techniques such as signature-based analysis and fuzzing. The experimental results presented in this research paper were obtained solely

by the Code Analysis module, using a subset of 11 static analysis tools. While this setup was performed for research purposes, using all CONSOLE modules will increase the security and usability.

(iii) Graphical User Interface (GUI): The GUI block represents the user-facing layer of the platform. It includes the CONSOLE Dashboard, which provides real-time visibility into security events, vulnerabilities, and code quality metrics, and the Training Platform, which delivers personalised learning content. The dashboard also allows redirections to specialised tools for advanced forensics and response actions, extending the platform's analytical capabilities.

(iv) User Environment (UE): This block encapsulates the various user roles interacting with the platform—developers, DevOps engineers, and administrators—whose feedback informed both the architecture and interaction design. The architecture thus ensures usability and relevance across diverse user workflows.

The integration of these conceptual blocks enables a comprehensive view of software security throughout its life-cycle, from code authoring and testing to deployment and maintenance. The reference architecture also supports interoperability with external tools, scalability across different deployment contexts, and adaptability to evolving threat landscapes. In this way, the CONSOLE platform aims to serve as a practical, modular, and user-oriented cybersecurity ecosystem for SMEs and DevSecOps environments.

3.3 The security clues

One of the main strengths of CONSOLE is the ability to orchestrate many security tools (that will be described in Section 4) to cover a wide range of security flaws and to support the most common programming languages and technologies. Yet, each tool is unique, having a different output format, different configuration options, and different ways of reporting the flaws. To overcome this issue, CONSOLE uses a common representation for the security reports, called *security clue*. A security clue is a JSON encapsulating the information about a security flaw, such as the file and line number where the flaw was detected, the type of the flaw, a description of the flaw, and other relevant information. The security clues are generated by the code analysis module and are used to populate the CONSOLE database.

The first 5 fields from Table 1 are for bookkeeping and are added automatically from the code analysis module. By saving the project ID, GitHub commit ID, job ID and the timestamp of the analysis, one can retrieve specific information from the clues database in the future. The clue name and category provide a quick overview of the nature of the discovered security flaw. Some tools like Insider use MITRE Common Weakness Enumeration [27] to name the clues. For instance, a clue named *CWE-327* identifies the usage

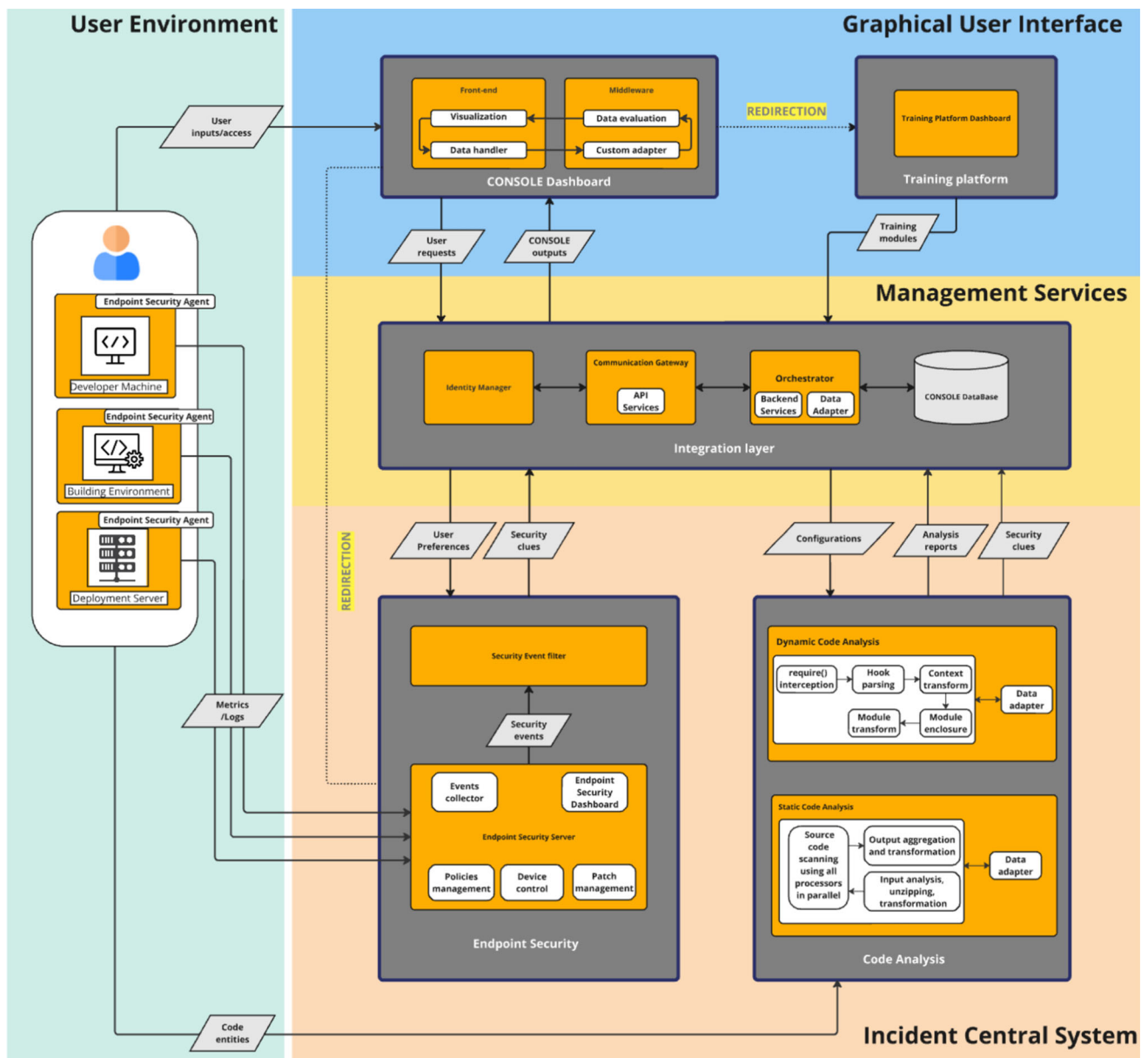


Fig. 1 High-level reference architecture of the CONSOLE platform.

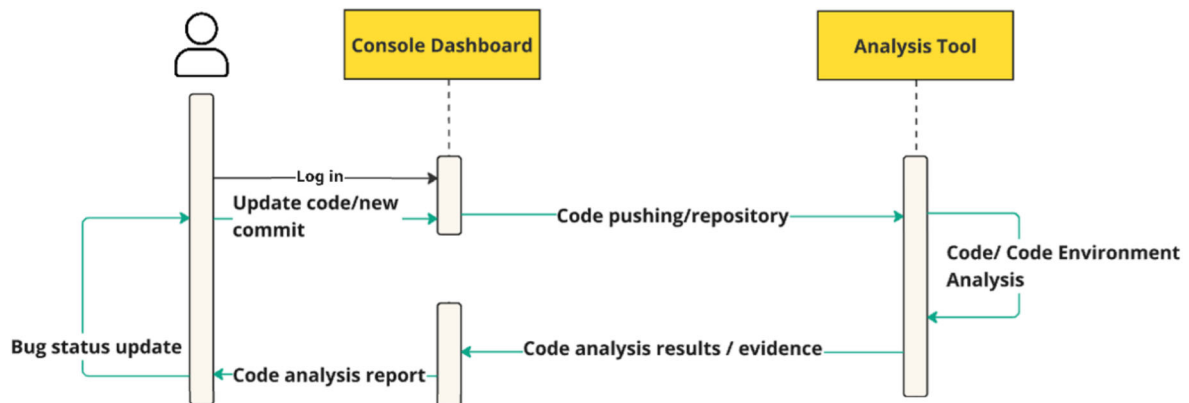


Fig. 2 UML component diagram focusing on the Code Analysis subsystem of the CONSOLE platform.

Table 1 Security clues fields

Field name	Mandatory	Description
project_id	YES	The ID of the project being analyzed
project_version	YES	The GitHub commit ID of the code being analyzed
job_id	YES	Identifier of the analysis job
timestamp	YES	Analysis time
tool	YES	Name of the tool
category	YES	Flaw category
name	YES	Name of the identified security flaw
description	YES	Detailed description of the identified flaw
source_name	NO	Relative path to the source file containing the flaw
source_line	NO	Line number from the source file containing the flaw
severity	YES	An integer from 0 to 10 indicating how serious the flaw is

of a broken or risky cryptographic algorithm [29]. Other tools like Bandit use more descriptive names, such as *hard-coded_sql_expressions*. The clue description is provided by the analysis tool and should provide actionable intelligence to the user, so he can decide if the clue is indeed a security flaw and if it is, then how to fix it. Most clues will be related to some source file and one or more source lines, pinpointing the origin of the security flaw. These fields are however optional, as not all security flaws can be localized to a specific point in the source code. The *severity* field tells us how critical is the discovered security flaw. Since each analysis tool has its own notion of severity, one of the challenges of CONSOLE is to harmonize these severities, such that clues with the same severity should be given the same importance, regardless of the tool that generated the clue. The approach, that will be detailed in Section 5 is based on manual rating for a subset of randomly selected prevalent clues.

4 The code analysis module

The role of the “code analysis module” is to inspect the source code of a software project, find security defects, and report them as semi-structured data, following the Security Clues format. Since CONSOLE must support multiple languages and technologies, this module was designed to be extensible, supporting both open-source and proprietary tools. The current version of CONSOLE supports 17 tools, but for this analysis only 11 open-source tools were used. The tools are detailed in this section, and for each tool, we briefly mention what kind of analysis it does, the supported languages and technologies, the vulnerabilities that can be detected by it and whether the tool is actively maintained or not. While we aim for actively supported tools, even older tools can bring added value to CONSOLE, being able to detect timeless vulnerabilities such as buffer overflows.

```
python3 analyze.py $(PROJECT)
                    $(SETTINGS)
                    $(OUTPUT)
```

Fig. 3 Example of tool adapter calling.

Each of the analysis tools will run in a separate Docker container, from an image that was tailored to support the tool. Since each tool outputs the results in a different format, an adapter was implemented for each tool to run it on a specified project, with the specified configuration, and output all the results as Security Clues. Fig. 3 shows an example of a command for calling a tool adapter inside the container. The placeholder `$(PROJECT)`, `$(SETTINGS)`, and `$(OUTPUT)` will be replaced by the orchestrator by the path to the project, the path to the configuration file and the output path respectively.

Most of the analysis tools do not require internet access while running. For this reason, their containers will run without any network access, eliminating the risk of a 3rd party tool malfunctioning and leaking the source code. There are exceptions to this rule, however, like tools that need to use an on-line database to check the project libraries for known vulnerabilities. The CONSOLE platform allows the developer to select which tools to run, achieving the required level of privacy. Each tool container can be restarted after each analysis or reused to analyze further projects. For the current analysis, the containers were reused in order to minimize the analysis latency.

The orchestrator itself can run as a Python module, directly on the host or it can be dockerized as well, by mounting the Docker pipe inside its container.

The following subsections will present details on each analysis tool. A summary of the used tools, along with the version that was used is displayed in Table 2. The experiments

Table 2 Code analysis tools versions

Tool name	Version	Release date
Bandit	1.8.2	2025.01.13
Clj-holmes	1.4.3	2022.04.18
Cppcheck	2.14.2	2024.06.22
Dawnscanner	2.2.0	2023.04.17
Flawfinder	2.0.19	2021.08.29
Gosec	2.20.0	2024.05.14
Insider	3.0.0	2021.01.26
NjScan	0.4.3	2024.11.14
Progpiloot	1.1.0	2024.05.15
RATS	4ba54ce	2013.12.31
Sobelow	0.13.0	2023.08.06

```
try:
    ...
except Exception:
    pass
```

Fig. 4 Example of Bandit True Positive.

```
'%030x' % random.randrange(16**30)
```

Fig. 5 Example of Bandit False Positive.

in this paper were performed in February 2025, so the release dates of the versions that were used are prior to this date.

4.1 Bandit

Bandit [33] is a static code analysis tool that comes in the form of a Python module. To perform the analysis, it parses the entire project, creates an Abstract Syntax Tree (AST) [41], and runs the appropriate plugins against the AST nodes. It offers support for Python and can identify the following types of vulnerabilities: Cross-Site Scripting (XSS) and Insecure Web Configuration, SQL and Code/Command Injections, Hardcoded Secrets, Insecure Deserialization, Weak Cryptography, XML External Entity, Insecure Transport/SSL, Insecure Temporary Files, Binding to All Network Interface. The tool is actively maintained.

An example of issue properly detected by Bandit is depicted in Fig. 4. The clue `bandit/try_except_pass` indicates that the `except` block catches any generic exception but doesn't handle it.

Bandit has also triggered the False Positive in Fig. 5 as `bandit/blacklist`. Indeed, the random number generation is deterministic (pseudorandom), but the generated values are not used in a security context.

```
...
int* ptr = NULL;
...
*ptr = 1;
}
```

Fig. 6 Example of Cppcheck True Positive.

4.2 Clj-holmes

Clj-holmes [17] is a static code analysis tool that searches for vulnerable code using a pattern language [18]. The code is parsed into an AST then compares parts of the AST with the defined pattern matching rules. Custom rules can be added in order to extend the publicly available capabilities, which include: Command/Code Injection, SQL Injection, Hardcoded Secrets, Weak Cryptography, Insecure Deserialization, Insecure Network Configuration, Bad Authentication. It was designed for Clojure code and the tool had its last release on the 18th of August 2022.

4.3 Cppcheck

Cppcheck [21, 26] scans the C/C++ source code for bugs, undefined behavior, and potential security weaknesses without the need to compile/build the project. It uses syntax parsing and data flow analysis to see the life cycle of variables in functions and loops. It identifies standard issues and coding style issues, but also potential vulnerabilities such as: Memory Leaks, Out of Bounds Access, Null pointer dereference, Uninitialized Variables, Unsafe Functions Usage, Memory Overflows, Miscellaneous Logic and Risky Conditions, Input Formatting. The tool is still maintained.

Fig. 6 contains an example of code flagged by Cppcheck. The code contains a null pointer dereference, which causes an immediate process crash.

Cppcheck has also detected an uninitialized variable error (`cppcheck:uninitvar`) in Fig. 7, which is a False Positive, because the variable `iv_dec` is always set inside the conditional branches that check for client or server. While a developer would figure out that one of `MBEDTLS_SSL_SRV_C` and `MBEDTLS_SSL_CLI_C` is always defined, the static analyzer also considers the case where none of them is defined.

4.4 Dawnscanner

Dawnscanner [31] is a static analysis tool designed to review web applications written in Ruby, offering support for the major MVC frameworks, i.e. Ruby on Rails, Sinatra, and Padrino. The tool parses the `Gemfile.lock` trying to identify the used gems, the Ruby interpreter, and the MVC

```
#if defined(MBEDTLS_SSL_SRV_C)
iv_dec =
    traffic_keys->client_write_iv;
#endif
...
#if defined(MBEDTLS_SSL_CLI_C)
iv_dec =
    traffic_keys->server_write_iv;
#endif
...
memcpy(
    transform->iv_dec,
    iv_dec,
    traffic_keys->iv_len
);
```

Fig. 7 Example of Cppcheck False Positive.

framework, then checks the knowledge base for any vulnerabilities. It can also check the code for sinks to identify Cross-Site Scripting (XSS) and SQL Injections. It had its last release on the 17th of April 2023.

4.5 Flawfinder

Flawfinder [47] is a static analysis tool that scans C/C++ code in order to find potential security vulnerabilities by looking at dangerous function calls and known risky patterns. It uses lexical scanning to find tokens that suggest a potential vulnerability, then estimates the level of risk. This leads to an increase in the number of false positives. The following vulnerabilities are set to be found: Improper Input Validation, Improper Limitation of a Pathname to a Restricted Directory, Improper Neutralization of Special Elements used in an OS Command, Improper Restriction of Operations within the Bounds of a Memory Buffer, Buffer Copy without Checking Size of Input, Buffer Over-read, Uncontrolled Format String, Integer Overflow or Wraparound, Execution with Unnecessary Privileges, Use of a Broken or Risky Cryptographic Algorithm, Concurrent Execution using Shared Resource with Improper Synchronization, Insecure Temporary File, Use of Potentially Dangerous Function, Incorrect Permission Assignment for Critical Resource, Use of Path Manipulation Function without Maximum-sized Buffer, Reliance on Untrusted Inputs in a Security Decision, Inclusion of Functionality from Untrusted Control Sphere. The tool had its last release on the 29th of August 2021.

In Fig. 8, Flawfinder has detected a buffer overflow issue (flawfinder/buffer\strcpy), which occurs when data is written beyond the allocated memory space, potentially leading to security vulnerabilities. Developers should avoid using `strcpy` and used instead more secure functions, such as `strncpy` or `strcpy_s`.

```
strcpy(s->name, cfg.vdi);
```

Fig. 8 Example of Flawfinder True Positive.

```
char buf[3 * MAXLEN];
```

Fig. 9 Example of Flawfinder False Positive.

The buffer overflow detection from Flawfinder can also trigger False Positives. The buffer size from Fig. 9 is sufficient, yet a clue is emitted.

4.6 Gosec

Gosec [44] is a static analysis tool that checks for security-related issues in Go source code. It does that by parsing the source code using the native parser and building an AST for each source file, then checks the AST for insecure patterns using rules implemented as a separate analyzer structure. Custom rules can be implemented to expand the range of vulnerabilities that are identified. Currently, the tool has rules that cover the following vulnerabilities: Hardcoded Secrets, Command Injection, Unvalidated Inputs, Dangerous Functions, Weak Cryptography, TLS Insecurity, Uncontrolled Path Traversal, Files Created With Insecure Permissions, Race Conditions, Denial of Service. The tool is actively maintained.

4.7 Insider

Insider [36] is static analysis tool that focuses on covering the OWASP Top 10 [38] on source code of projects developed using the following technologies: Java (Maven and Android), Kotlin (Android), Swift (iOS), .NET Full Framework, C# and JavaScript (Node.js). It analyses the source code using pattern based rules defined using regular expressions. Its rules cover the following types of vulnerabilities: SQL Injection, Command Injection, Cross-Site Scripting, Insecure Cryptography Usage, Hardcoded Credentials, Improper Error Handling, Insecure API Usage, Resource Leaks and Access Control Issues. The tool had its last release on the 26th of January 2021.

4.8 NjScan

NjScan [2] is a static application testing tool that tries to find insecure code patterns in Node.js projects. For that, it uses a pattern matcher plug-in [1] and a syntax-aware pattern search tool, i.e., `semgrep` [39], on a parsed JavaScript AST. It has rules defined for the following types of vulnerabilities: Weak Cryptography, Hardcoded Secrets, Cross-Site Scripting, Server-Side Request Forgery, SQL Injection, NoSQL

```
&&(this.loginForm.username="hippo4j",
    this.loginForm.password="hippo4j")
```

Fig. 10 Example of NjScan True Positive.

```
<div class="panel-body">
  <pre>
    <?= file_get_contents($file) ?>
  </pre>
</div>
```

Fig. 11 Example of Progpilot True Positive.

Injection, Insecure Configuration, Command Injection, Code Injection, Missing Proper Input Validation, Open Redirect and Cross-Site Request Forgery. The tool had its last release on the 14th of November 2024.

In Fig. 10, NjScan properly detected a security flaw related to hardcoded credentials in the source code (njsscan/node_password).

4.9 Progpilot

Progpilot [35] is a static analysis tool that has as its target to identify security vulnerabilities in PHP code. During the analysis, it builds an AST of the analyzed code and then it tries a set of predefined rules to find malicious data flows from sources, i.e., user inputs to sinks, which are potentially dangerous functions. It specializes in finding common web vulnerabilities such as: Cross-Site Scripting (XSS), SQL Injection (SQLi), Remote Code Execution (RCE), Insecure Deserialization, Insecure Configuration, and Bypass Access Control. The tool is actively maintained.

An example of vulnerability detect by Progpilot is the cross-site scripting (XSS) from Fig. 11. The attacker can exploit it by injecting malicious scripts into web pages viewed by other users. This can lead to unauthorized access to user data, session hijacking, and other security vulnerabilities.

4.10 RATS

RATS (Rough Auditing Tool for Security) [10] is a tool developed for scanning C, C++, Perl, PHP, Python, and Ruby source code, performing a rough audit of it, trying to find security vulnerabilities. It uses a lexical parsing and pattern matching to identify possible insecure parts in the source code. The vulnerabilities it can find cover the following: Buffer Overflows, Race Conditions, Unbounded Inputs, Format String Vulnerabilities, Command Injection, Insecure Temporary Files, Memory Leaks, SQL Injection, Cross-Site

```
system(
    "rm -rf #{@siab_home}/siab.pid"
)
```

Fig. 12 Example of RATS True Positive.

```
while(n < bytes){
    int next =
        read(fd, buffer+n, bytes-n);
    if(next <= 0)
        error("read failed");
    n += next;
}
```

Fig. 13 Example of RATS False Positive.

Scripting, Unsafe Deserialization. The tool had its last release on the 31st of December 2013, yet is still being used.

In Fig. 12, RATS has detected the usage of the `system` function with a dynamically built command (`rats/system`), which may be vulnerable to command injection attacks.

RATS can also trigger False Positives such as the one involving the `read` system call in Fig. 13. While the address where the read data is placed is computed using pointer arithmetic, the code is secure because the buffer limits can't be exceeded.

4.11 Sobelow

Sobelow [37] is a security-focused static analysis tool developed for finding vulnerabilities in Elixir source code that uses the Phoenix framework. It creates the AST for the code under analysis and applies a set of defined rules to pattern match potential issues related to security. It also checks the configuration files to detect dangerous misconfigurations. The vulnerabilities it finds are: Insecure Configuration, Known-Vulnerable Dependency, Cross-Site Scripting, SQL Injection, Command Injection, Denial Of Service, Code Execution, Directory Traversal and Unsafe Deserialization. The tool had its last release on the 6th of August 2023.

5 Research methodology

5.1 Detected flaws and blind spots

The CONSOLE platform is useful for the stakeholders because it detects security defects that could otherwise pass unnoticed and reach the production stage, resulting in insecure software being released. The detection rate (True Positive Rate / Recall/ Sensitivity) is given by the ability of

the underlying tools to detect the security defects. To evaluate it, we need an annotated dataset containing labeled vulnerable code, suitable for Static Application Security Testing (SAST). We will use the *vulnerable-code-examples* repository [34] that contain 44 examples for vulnerable code, from which 26 are suitable for SAST. While the number of tests from this dataset is small, it covers the most common types of vulnerabilities and multiple programming languages such as C, C++, C#, Java, JavaScript, PHP and Python. The evaluation results will be covered in Section 6.3.

5.2 False positives mitigation strategy

One of the main challenges of using SAST tools is managing the high number of alerts [46], many of them being False Positives. Since CONSOLE aggregates multiple tools, with some languages and technologies being covered by more than one tool, reducing the number of False Positives and reducing the overall number of alerts that are being displayed is critical. Naturally, some alert types are encountered more often than others and some alert types are more prone to False Positives.

We will denote by N the total number of tools used in the analysis ($N = 11$) and by n_i the total number of clues types that can be emitted by the tool i ($1 \leq i \leq N$).

Let T_i be the probability that tool i ($1 \leq i \leq N$) will emit a clue on a piece of code that can be handled by that specific tool and C_{ij} the probability that clue j from tool i ($1 \leq j \leq n_i$) will be emitted on a piece of code written in the supported languages. Similarly, we will denote by t_i the probability that a clue emitted by tool i is a True Positive and by c_{ij} the probability that clue j from tool i is a True Positive.

For a given language and/or technology, let $S \subset \{1, 2, \dots, N\}$ be the subset of tools supporting it and let k be the number of code pieces from a project implemented in the given language and/or technology. Then, the expected number of emitted clues is E_1 from Eq. 1, which is the number of Predicted Positives ($PP = TP + FP$).

$$E_1 = k \cdot \sum_{i \in S} T_i = k \cdot \sum_{i \in S} \sum_{j=1}^{n_i} C_{ij} \quad (1)$$

$$E_2 = k \cdot \sum_{i \in S} T_i t_i = k \cdot \sum_{i \in S} \sum_{j=1}^{n_i} C_{ij} c_{ij} \quad (2)$$

The number of expected True Positive clues is E_2 from Eq. 2, which is the number of True Positives (TP). To improve the usability of the CONSOLE platform, we must maximize the Precision, expressed in Eq. 3. Note that the precision (P) is independent on the number of analyzed code pieces k .

$$P = \frac{TP}{TP + FP} = \frac{E_2}{E_1} = \frac{\sum_{i \in S} T_i t_i}{\sum_{i \in S} T_i} = \frac{\sum_{i \in S} \sum_{j=1}^{n_i} C_{ij} c_{ij}}{\sum_{i \in S} \sum_{j=1}^{n_i} C_{ij}} \quad (3)$$

We argue that the Precision (P) is important for usability by considering the clues that the developer must investigate. If the Precision is 100%, all the clues are actual security issues and by fixing them, the code becomes more secure. If the Precision is 50%, then half of the investigated clues will be False Positives, but the other half are relevant. If the Precision drops to 10%, then 9 out of 10 investigated clues will be False Positives and the developer will waste time to decide which clues are actually relevant.

The strategy for mitigating the False Positives is to hide the clue types with a very low probability of being True Positives ($c_{ij} < 5\%$) and adjust the severity for those with low probability ($5\% \leq c_{ij} < 50\%$). This way, the clues displayed to the developers will be re-prioritized, enabling them to focus on the reported issues that are more likely to be security vulnerabilities.

To estimate the True Positive probabilities for every clue type, we have manually analyzed a sample of the results obtained on the experimental dataset (more details on the dataset are provided in Section 6.1). A clue type was included in the sample for manual analysis only if it appears in at least 100 distinct projects. For each clue type we will sample at most 3 projects and at most 3 occurrences of the clue per sampled project. We have selected for manual analysis only prevalent clue types, since the False Positives on them will have a greater impact on usability than rare ones. The number of samples per clue type is limited, to enable the analysis of a wide range of clues. The sample clues were rated using one of the scores below:

- 3 – *SERIOUS*: the clue points to a real security vulnerability that must be addressed by the developers;
- 2 – *INTERESTING*: the manual analysis didn't reveal an actual vulnerability, but the developer should still visualize the clue, as it may provide important insights such as insecure coding practices;
- 1 – *FALSE*: the clue is a False Positive and should not be displayed.

The rating was performed by multiple experts, each clue type being analyzed by a single person, based on their expertise with the specific programming language and vulnerability type. While we needed binary labels (True Positive or False Positive) for the clues, we used the three labels above, including the *INTERESTING* category to allow the analyst a middle ground between “definitely vulnerable” and “definitely secure”. This label would be used by an analyst in a

situation where the code might not be vulnerable, but they would still tag the insecure coding construct in a manual code review. There was also a fourth option for the analysts, to skip clues where they weren't confident about the verdict, since the number of collected clues exceeded our manual labeling capacity. While we acknowledge that this approach is still prone to biases and verdict variations, the team worked together to calibrate the scoring system before starting the work.

Clues rated 2 or 3 will be considered True Positives, while clues rated 1 will be considered False Positives. We will denote by $R = R_T \cup R_F$ the set of rated clues, R_T being the set of clues rated as True Positive and R_F the set of clues rated as False Positive. We will approximate the True Positive probabilities for tools (t_i) and for clues (c_{ij}), with $1 \leq i \leq N$ and $1 \leq j \leq n_i$ as in Eq. 4 and Eq. 5. If no clues for tool i were manually analyzed, then $t_i = 1$, otherwise t_i is the ratio between the number of True Positives and the total number of analyzed clues. c_{ij} is computed in a similar manner, except that $c_{ij} = t_i$, if no clues of type j for tool i were analyzed.

$$t_i = \begin{cases} 1, & \text{if } \nexists r \in R \text{ s.t. } r.\text{tool} = i \\ \frac{|\{r \in R_T \mid r.\text{tool} = i\}|}{|\{r \in R \mid r.\text{tool} = i\}|}, & \text{otherwise} \end{cases} \quad (4)$$

$$c_{ij} = \begin{cases} t_i, & \text{if } \nexists r \in R \text{ s.t. } r.\text{tool} = i \wedge r.\text{clue} = j \\ \frac{|\{r \in R_T \mid r.\text{tool} = i \wedge r.\text{clue} = j\}|}{|\{r \in R \mid r.\text{tool} = i \wedge r.\text{clue} = j\}|}, & \text{otherwise} \end{cases} \quad (5)$$

5.3 The impact of projects popularity on security

The relationship between project popularity and software security presents a complex research question that has significant implications for open-source development practices. While popular projects benefit from increased community scrutiny and development resources, they also tend to be larger and more architecturally complex, potentially introducing additional security vulnerabilities. Establishing meaningful comparisons between projects of varying popularity levels requires addressing fundamental methodological challenges related to project heterogeneity, scale differences, and technological diversity. The methodology that is described in this section aims to establish a systematic approach for objectively comparing the security quality of popular versus unpopular projects, accounting for factors such as project size and complexity that could otherwise compromise the comparison.

The methodology that was applied addresses the inherent complexity of comparing projects with substantially different characteristics through systematic normalization and control strategies. The dataset of 10,855 repositories was

partitioned into two distinct categories based on GitHub star count: high-popularity projects with 1000 or more stars and low-popularity projects with fewer than 1000 stars (detailed statistics provided in Section 6.1). This threshold represents a statistically significant boundary point that distinguishes individual or small-scale development efforts from established, community-maintained repositories with broader adoption patterns.

The critical methodological innovation involves the selection of appropriate comparative metrics for security vulnerability assessment. Average security clues per MB of code was adopted as the primary dependent variable rather than absolute clue counts per project. This normalization approach addresses several fundamental challenges that would otherwise compromise analytical validity.

In particular, large repositories inherently contain larger volumes of source code, resulting in proportionally higher absolute vulnerability counts, independent of the quality of security practices. Absolute clue enumeration would introduce systematic distortion, as preliminary analysis demonstrates substantial disparities in absolute clue counts that reflect codebase magnitude rather than security posture. This necessitates the density-based normalization strategy employed.

Furthermore, high-popularity repositories frequently employ diverse programming languages and frameworks, each exhibiting distinct vulnerability patterns and detection signatures across CONSOLE's integrated security tools. The per-MB normalization methodology accounts for this technological diversity by standardizing vulnerability measurements across different language ecosystems while preserving meaningful security quality indicators.

The methodology incorporates additional control variables to ensure comprehensive comparison validity. Analysis processing time serves as an indicator for project complexity, while the distribution of repositories producing zero security clues provides insight into potential analytical limitations versus genuine security excellence. Projects with elevated community engagement typically demonstrate increased architectural sophistication, expanded feature matrices, higher file counts, and coordinated development team activities. The density-based approach isolates security practice efficacy from these confounding project scale and complexity factors.

6 Experimental results

6.1 Dataset description

The dataset consists of 10,855 public projects crawled from GitHub. We have tried to build a diverse dataset, covering:

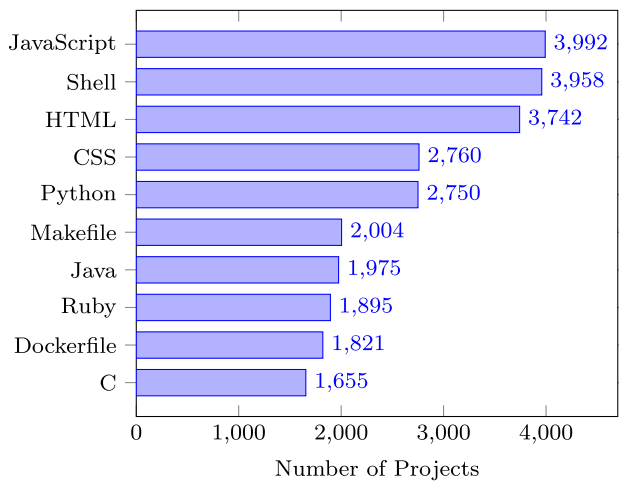


Fig. 14 Number of projects per programming language.

- various programming languages and technologies;
- recent and old projects;
- various sizes, from very small to very large;
- projects that are actively maintained and project with very few commits;
- highly rated projects (with many stars on GitHub) and unrated projects.

We have used the GitHub API [6] for crawling, benefiting from the rich information provided for the public repositories.

For crawling random projects, we generated random IDs and requested from GitHub a page with projects starting from the generated id. At the time of writing this paper, there are approximately 1 billion projects on GitHub, which are conveniently numbered from 1 to 1 billion. Most projects on GitHub tend to be small, both in terms of code size and development activities. Also, some programming languages tend to be underrepresented, so random sampling is not sufficient. Fortunately, the GitHub API allows querying for projects with given characteristics, such as the minimum number of stars or some programming language being used. Overall, 35% of the dataset was generated purely random (by generating random GitHub IDs), while the remaining 65% was generated by specifically querying GitHub for popular projects using the languages and technologies covered by the CONSOLE tools.

Figure 14 shows the top 10 programming languages and technologies, by the number of projects they are used in. Note that most projects contain multiple programming languages and technologies. Similarly, Figure 15 shows the top 10 languages based on the code size (in GB).

Most of the analyzed projects are recent and currently maintained, more than 3000 having the last commit in 2025,

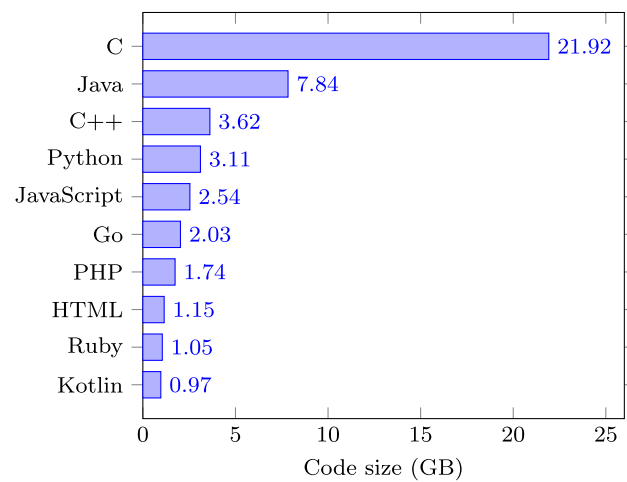


Fig. 15 Code size per programming language.

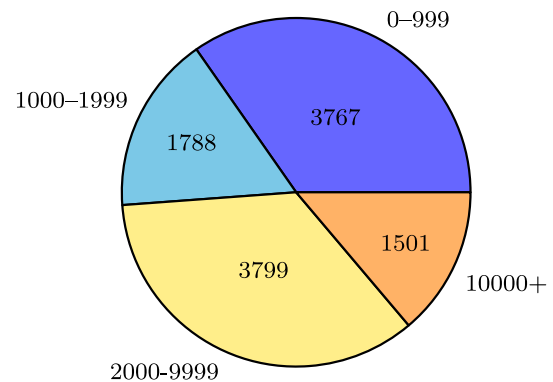


Fig. 16 Distribution of projects by number of GitHub stars.

about 2500 in 2024, while almost 1800 projects have the last commit between 2009 and 2019.

The number of stars for a GitHub repository indicates the number of users that marked the repository as favorite. Figure 16 shows the distribution of the projects based on the number of stars.

6.2 Code analysis performance

All the projects crawled from GitHub were analyzed by CONSOLE on a MacBook Pro with 2.6Ghz 6-Core Intel Core i7 CPU and 16 GB of RAM.

Since the analysis time can be very high for some outliers, we have capped the running time of the CONSOLE tools to 3 minutes. After this time, if the tool is still running, the process is stopped and the partial results (if any) are used. Note that the total analysis time can be higher than 3 minutes, as there may be extra time for parsing the partial results and waiting for the Docker containers to close.

The average running time per project was 42.79 seconds. Intuitively, the running time should increase with the code

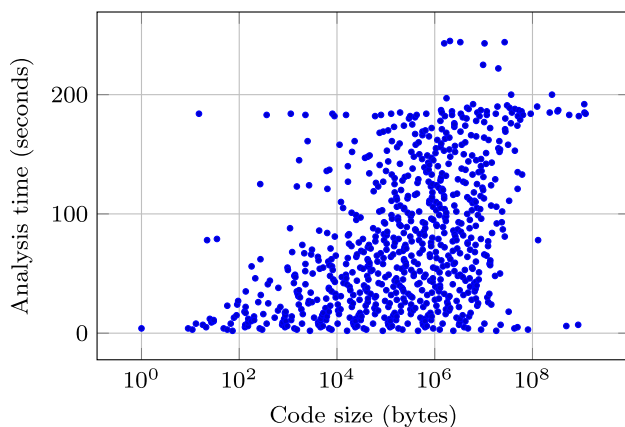


Fig. 17 Analysis time.

size, so we computed an average running time per KB of analyzed code to 4.67 second (macro average). By examining the running times more deeply, we have found that although there is a correlation between the code size and the analysis time, there are still some large projects with very small running times and small projects with large running times. Figure 17 shows a scatter plot with the project sizes on the x axis (logarithmic scale) and the total analysis time on the y axis. Note that the code size in this plot is not the total size of the code in the GitHub repository, but the size of the code belonging to languages and technologies supported by CONSOLE tools.

Considering the CONSOLE integration in CI/CD pipelines, the average running time of 42.79 seconds should not have a significant impact. Although other security operations such as checking the Software Bill of Materials (SBOM) [7] for known security risks and vulnerabilities are much faster (usually less than a second), the CONSOLE approach is proactive and able to spot even zero-day vulnerabilities in user code, not just in third-party software components. Also, the code analysis should not be seen as an overhead to the build process, but rather a second opinion on the code review, that is traditionally performed manually and takes more time than the automated analysis performed by CONSOLE.

The code analysis module can run multiple analysis tools in parallel if they are applicable to the same project. However, each analysis tool ran single-threaded, even if some of them could be run concurrently on multiple source files from the same project. A future solution to speed-up the running time is to explore the internal parallelism of these tools. Another viable solution for large projects is to scan only the modified sources between consecutive versions, significantly reducing the amount of code to be scanned in CI/CD pipelines where every version of the code gets scanned.

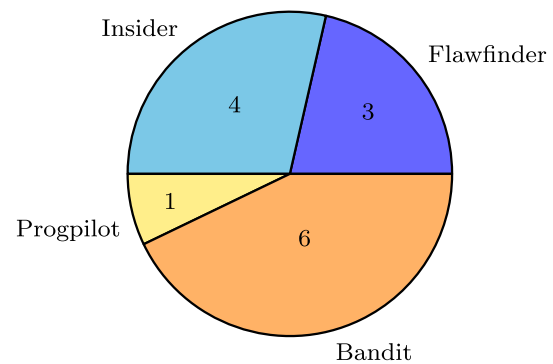


Fig. 18 Code analysis tools contributions to detections.

6.3 Evaluation metrics

The evaluation of the code analysis tools on the dataset from [34] yielded 14 detections out of 26 applicable cases, achieving the True Positives Rate / Recall $R = 53.85\%$.

Fig. 18 presents the contribution of each tool to the detections on the test cases. The pie chart shows the complementarity of the tools, as no single tool is able to handle multiple programming languages and technologies. The Recall score is promising, yet it highlights the need of using more tools to cover the blind spots of the existing toolset.

The large scale analysis of the public projects dataset (10,855 projects) revealed an average of raw clues per project of 142.56, which a single developer or a small team can manage without a significant overhead. Since projects vary in size, we also computed the number of clues per MB (Megabyte) of code, which is 655.2. Only 2642 of the analyzed projects had no clues.

The number of distinct clue types that were identified for the 11 analyzed tools is 688, from which 117 appeared in at least 100 projects and were considered for manual analysis. We analyzed 274 clues, out of which 156 were rated as True Positive, obtaining the Precision $P = 56.93\%$. Table 3 shows the 11 code analysis tools, along with the number of applicable projects (note that CONSOLE may use multiple tools for the same project) and the total number of clues emitted by that tool. The last column contains the manual rating that was computed for the tool, according to the methodology from Section 5.2.

The clue types were also rated using the methodology from Section 5.2 and we recomputed the metrics based on the adjusted scores in Table 4. The table shows that more than half (according to the average number of clues per project) or even more than two thirds (according to the average number of clues per MB of code) are False Positives. The remaining 33–50% are not necessarily security vulnerabilities, but issues that the developers should be aware of. The table also contains the 95th percentile (denoted as P95) for both the

Table 3 Manual ratings of the analysis tools

Tool name	Nr. projects	Nr. clues	Rating
Insider	6715	268471	0.31
RATS	4414	9995	0.75
NjScan	3992	14128	1.00
Bandit	2750	682063	0.65
Cppcheck	1897	13308	0.88
Flawfinder	1897	521018	0.31
Dawnscanner	1895	3359	1.00
Progpilot	1093	3792	1.00
Gosec	1075	30695	0.67
Clj-holmes	149	29	1.00
Sobelow	149	3	1.00

Table 4 Raw vs adjusted results

Metric	Raw	Adjusted
Avg. clues per project	142.56 (P95: 569)	70.1 (P95: 224)
Avg. clues per MB	655.2 (P95: 1034)	207.1 (P95: 502)
Projects with zero clues	2642	3999

raw and adjusted measurements. Since we have a significant number of projects with zero clues, the 5th percentile is 0.

6.4 Measuring the impact of projects popularity on security

The methodology from Section 5.3 was validated through quantitative analysis of the complete dataset, demonstrating the effectiveness of the density-based normalization approach. Table 5 presents the comparative results that validate the methodological framework across multiple dimensions. We present both the raw results, considering every clue, and the adjusted results, based on the False Positives mitigation technique. The final metrics that should be considered are the adjusted ones.

The primary validation finding confirms the effectiveness of density-based normalization in revealing meaningful security differences. Popular projects (≥ 1000 stars) demonstrate significantly lower vulnerability density at 144.93 clues per MB compared to 324.09 clues per megabyte for less popular projects. Popular projects have 55.3% fewer vulnerabilities per MB, which validates the methodological approach by demonstrating that popular repositories implement superior security practices per unit of code once confounding size factors are controlled.

Table 5 Validation Results: Popular vs Less Popular Projects

Metric	≥ 1000 stars	< 1000 stars
Projects analyzed (n)	7,088	3,767
Avg. analysis time (s)	52.23	25.03
Clues per project (raw)	195.59 (P95: 846)	42.79 (P95: 124)
Clues per project (adjusted)	96.79 (P95: 346)	19.87 (P95: 51)
Avg. clues per MB (raw)	598.36 (P95: 701)	762.14 (P95: 1970)
Avg. clues per MB (adjusted)	144.93 (P95: 370)	324.09 (P95: 831)
Projects with zero clues (raw)	1,097 (15.5%)	1,545 (41.0%)
Projects with zero clues (adjusted)	2,124 (29.9%)	1,875 (49.8%)

The complexity validation metric supports the methodological framework through analysis of processing time differences. Popular projects require substantially longer processing duration (52.23 seconds average) compared to less popular projects (25.03 seconds average), confirming that the methodology appropriately accounts for increased structural complexity in established repositories. This processing time differential validates that the observed security improvements occur despite, rather than because of, reduced analytical complexity.

The clues per project comparison illustrates why normalization is essential for valid comparison. Without density adjustment, popular projects appear to have substantially higher vulnerability counts (96.79 versus 19.87 clues per project), which would lead to the erroneous conclusion that popular projects are less secure. This demonstrates that the methodology successfully eliminates the size effect that would otherwise invalidate comparative analysis.

Coverage assessment through zero-clue project distribution provides additional methodological validation. Popular projects show lower rates of zero-clue results (29.9% or 2,124 projects) compared to less popular projects (49.8% or 1,875 projects). This pattern suggests that the methodology achieves more comprehensive security coverage for popular projects rather than systematic analytical blind spots, supporting the validity of the comparative approach.

These validation results collectively demonstrate that the standardized normalization methodology successfully controls for confounding variables while preserving actual security practice differences. The consistent patterns across multiple validation dimensions confirm that observed differ-

Table 6 Language-Specific Security Analysis by Project Popularity

Language	Popularity	<i>n</i>	Avg Clues/MB
Python	≥ 1000★	752	415.59 (P95: 1158.27)
	< 1000★	556	192.61 (P95: 855.24)
JavaScript	≥ 1000★	658	565.79 (P95: 125.78)
	< 1000★	818	76.73 (P95: 424.7)
PHP	≥ 1000★	660	19.23 (P95: 64.80)
	< 1000★	124	414.15 (P95: 1209.32)
Go	≥ 1000★	779	72.42 (P95: 291.56)
	< 1000★	77	156.69 (P95: 926.71)
C	≥ 1000★	744	139.88 (P95: 512.62)
	< 1000★	104	338.57 (P95: 1570.85)
Kotlin	≥ 1000★	490	14.72 (P95: 39.35)
	< 1000★	54	7.05 (P95: 37.50)
Java	≥ 1000★	754	13.80 (P95: 46.50)
	< 1000★	489	24.09 (P95: 76.90)
Ruby	≥ 1000★	736	123.50 (P95: 247.31)
	< 1000★	78	651.44 (P95: 1896.54)
Elixir	≥ 1000★	118	20.65 (P95: 93.63)
	< 1000★	6	23.17 (P95: 75.29)

ences reflect authentic variations in security implementation quality rather than methodological inaccuracies.

To address potential confounding factors related to language-specific coding patterns and tool sensitivities, a grouped analysis by dominant programming language was performed. For each project, the dominant language was identified as the one comprising at least 50% of the codebase by size. Projects without a clear dominant language were excluded from this analysis, resulting in 7,997 projects distributed across 9 languages (from the original dataset of 10,855 projects). Table 6 presents the security metrics stratified by language and popularity level, revealing substantial heterogeneity in the relationship between popularity and security quality across programming language ecosystems. For each language, the table reports the sample size (*n*), average vulnerability density (Avg Clues/MB) along with the 95th percentile (P95) for both popular (≥ 1000★) and unpopular (< 1000★) projects. The inclusion of both mean and P95 metrics is essential due to the nature of vulnerability distributions, where standard deviations substantially exceeding the mean indicate the presence of extreme outliers that disproportionately influence central tendency measures.

The stratified analysis demonstrates that six languages (PHP, Go, C, Java, Ruby, and Elixir) exhibit substantial reductions in vulnerability density for popular projects, confirming the hypothesis that popularity correlates with enhanced security practices within these ecosystems. PHP demonstrates the most pronounced effect, with popular projects exhibiting dramatically lower mean vulnerability

density compared to unpopular projects, a pattern corroborated by P95 values. Ruby, C, Go, and Java demonstrate consistent improvements across both mean and P95 metrics, suggesting that mature tooling ecosystems and codified security practices confer systematic advantages to popular projects. Elixir demonstrates a modest improvement, though the extremely small sample size for unpopular Elixir projects (*n* = 6) limits statistical confidence in this comparison and precludes definitive conclusions.

Conversely, three languages (Python, JavaScript, Kotlin) demonstrate elevated mean vulnerability density in popular projects, presenting an apparent contradiction to the general trend observed in other language ecosystems. However, rigorous examination of P95 and standard deviation values reveals this paradox derives from outlier effects rather than systematic security degradation. Specifically, JavaScript exhibits the most pronounced manifestation of this phenomenon, whereby mean values suggest popular projects demonstrate substantially inferior security posture while the P95 values demonstrate an inverse relationship.

6.5 Discussion

Using CONSOLE for large-scale analysis reveals several insights into automated vulnerability scanning and how it should be integrated within a CI/CD pipeline. The results highlight that manual analysis is no longer effective for the large datasets available today, making automation essential for achieving proper security assessments. In addition, the following observations should also be discussed:

- There is no single SAST tool capable of achieving comprehensive coverage across multiple programming languages and vulnerability types. Therefore, combining several tools (particularly tools that complement each other) is essential to increase the probability of identifying potential vulnerabilities. Within the CONSOLE project, the selected tools were chosen to provide a balanced outcome between sensitivity and specificity. This balance is reflected in the precision of 56.9% obtained after the manual validation of the results.
- The main challenge remains the False Positives, with more than 50% of the identified clues not directly linked to a vulnerability that can be exploited. It is also important to highlight that False Positives are responsible for the alert fatigue phenomena, where developers are overwhelmed by excessive alerts and must manually identify the relevant ones. To address this issue, we designed a system that calibrates the severity and probability of each clue (similar to a prioritization or triage mechanism), that identifies the clues that are most relevant from a security perspective. This approach also helps maintain

a balance between usability, which is heavily impacted by alert fatigue, and security effectiveness within CI/CD pipelines. Further research is necessary in order to bring the False Positives Rate as close to 0 as possible. One possible direction is to use powerful AI such as Large Language Models to investigate the detected clues and filter out the FPs. This approach would benefit from the speed of SAST tools, using slower but precise models only for the scenarios with high uncertainty. A second direction is to incorporate the user feed-back, provided in the CONSOLE dashboard. A clue marked as FP will be “remembered” and will not be displayed when new versions of the same code are analyzed.

- The results show that over 60% of analyzed projects contained security flaws. This emphasize two important concept: (1) mature repositories are not safe just because they exist for a longer period of time and because they are well maintained, and (2) including an automated scan in every CI/CD pipeline (no matter how fewer external dependencies are being used) is mandatory for proper security. There is a clear relation between larger projects (with increased visibility) and the security associated with that project. Our research showed that there are 55% less vulnerabilities per MB in larger (popular) repositories than in less popular ones.
- CONSOLE analysis time is on average 42.7 seconds / project and can be easily scaled because it relies on a docker-based architecture. This highlights that we can use CONSOLE for a large variety of repositories with a minimum impact in terms of delivery, but with an improved quality in terms of vulnerability assessment.

We should also highlight that while static analysis can identify various potential risky uses-cases, this endeavour should be completed with dynamic analysis as well and in the future a mix-up that will also include generative AI and Large Language Models (LLM) based technologies.

7 Conclusions

This paper explored CONSOLE, an all-in-one cybersecurity platform that meets the needs of modern software development. CONSOLE is a complete framework that goes beyond detection to support prevention, response, and resilience. It does this by combining multiple static and dynamic analysis tools, endpoint monitoring, CI/CD integration, and contextual developer training.

We evaluated 10,855 open-source projects and observed that very few of them are free of security issues, even ones that could be easily fixed with tools that are already available. These results indicate that automated code evaluation should

be a standard practice for making software protected. The CONSOLE platform not only depicts these types of weaknesses, but it also helps teams deal with them by linking data, demonstrating risks, and offering them targeted training. This all-encompassing method provides developers and project managers greater autonomy.

CONSOLE enhances organisational readiness for incident response and crisis management by minimising detection latency, setting up actionable information, and offering real-time visibility into security postures. This is particularly crucial for businesses managing vital infrastructure, since even minor errors can cause major disruptions. CONSOLE facilitates the transition from reactive to proactive resilience by integrating security into development pipelines and daily processes. Ultimately, CONSOLE demonstrates that with the right integration of automation, insight, and usability, it is possible to significantly enhance the security of software systems without overburdening developers or compromising agility.

Future work will involve decreasing false positives through improved clue sorting processes, integrating with vulnerability databases for improved context, and expanding language support. We also intend to verify the platform in further real-world pilots and expand its applications to support regulatory compliance and cyber insurance evaluations.

Author Contributions Design and implementation of the CONSOLE platform: C.O., M.T., H.M., D.O.G., A.M., O.M., C.K., E.L., G.D., N.F.; testing and validation of the CONSOLE platform: A.G., G.D., N.F.; literature review: A.G., D.G., requirements analysis: H.M., architecture design: M.T., C.O.; code analysis module: C.O., D.O.G., N.F.; research methodology: C.O., E.L., N.F., H.M., S.I.; experimental analysis: C.O., E.L., D.G.; draft manuscript preparation and review: all authors. All authors have read and agreed to the published version of the manuscript.

Funding This project has received funding from the European Union’s Digital Europe Programme under grant agreement no. 101128070.

Data Availability The dataset containing the security clues emitted for the analyzed projects and the projects metadata is available at <https://github.com/CONSOLE-Project-EU/lisa-dataset>.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regula-

tion or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

References

- Abraham, A.: libsast. <https://github.com/ajinabraham/libsast>. Accessed: 2025-07-30
- Abraham, A.: njscan. <https://github.com/ajinabraham/njscan>. Accessed: 2025-07-30
- Bodipudi, A.: Integrating vulnerability scanning with continuous integration/continuous deployment (ci/cd) pipelines. *European Journal of Advances in Engineering and Technology* (2022). <https://doi.org/10.5281/zenodo.13622978>
- Casola, V., De Benedictis, A., Mazzocca, C., Orbinato, V.: Secure software development and testing: A model-based methodology. *Computers & Security* **137**, 103639 (2024). <https://doi.org/10.1016/j.cose.2023.103639>. (<https://www.sciencedirect.com/science/article/pii/S0167404823005497>)
- Chalishhafshejani, S., Pham, B.K., Jaatun, M.G.: Automating security in a continuous integration pipeline. In: *IoTBDs*, pp. 231–238 (2022)
- Cosentino, V., Luis, J., Cabot, J.: Findings from github: methods, datasets and limitations. In: *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 137–141 (2016)
- Dalia, G., Visaggio, C.A., Di Sorbo, A., Canfora, G.: Sbom ouverture: What we need and what we have. In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pp. 1–9 (2024)
- Dennis, K., Dehaan, B., Momeni, P., Laverghetta, G., Ligatti, J.: Large-scale analysis of github and cves to determine prevalence of sql concatenations. In: *International Conference on Security and Cryptography (SECRYPT)* (2024)
- D’Onofrio, D.S., Fusco, M.L., Zhong, H.: Ci/cd pipeline and devsecops integration for security and load testing. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2023)
- Dunham, A.: Rats – rough auditing tool for security. <https://github.com/andrew-d/rough-auditing-tool-for-security>. Accessed: 2025-07-30
- Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15. IEEE (2019)
- Fischer, F., Höbenreich, J., Grossklags, J.: The effectiveness of security interventions on github. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2426–2440 (2023)
- Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pp. 557–560 (2020)
- Groenendaal, J., Helsloot, I., Reuter, C.: Towards more insight into cyber incident response decision making and its implications for cyber crisis management. In: *19th International Conference on Information Systems for Crisis Response and Management. ISCRAM* (2022)
- Górski, T.: Adapt: A reusable package for implementing smart contracts that process transactions of congruous types. *Software Impacts* **21**, 100694 (2024). <https://doi.org/10.1016/j.simpa.2024.100694>. <https://www.sciencedirect.com/science/article/pii/S2665963824000824>
- Hajdu, Á., Jovanović, D.: solc-verify: A modular verifier for solidity smart contracts. In: *Working conference on verified software: theories, tools, and experiments*, pp. 161–179. Springer (2019)
- clj holmes: clj-holmes. <https://github.com/clj-holmes/clj-holmes>. Accessed: 2025-06-15
- clj holmes: shape-shifter. <https://github.com/clj-holmes/shape-shifter>. Accessed: 2025-07-30
- Humayun, M., Niazi, M., Jhanjhi, N.Z., Alshayeb, M., Mahmood, S.: Cyber security threats and vulnerabilities: a systematic mapping study. *Arab. J. Sci. Eng.* **45**(4), 3171–3189 (2020)
- Iosif, A.C., Gasiba, T.E., Zhao, T., Lechner, U., Pinto-Albuquerque, M.: A large-scale study on the security vulnerabilities of cloud deployments. In: *International Conference on Ubiquitous Security*, pp. 171–188. Springer (2021)
- Iqbal, Y., Sindhu, M.A., Arif, M.H., Javed, M.A.: Enhancement in buffer overflow (bof) detection capability of cppcheck static analysis tool. In: *2021 International Conference on Cyber Warfare and Security (ICWS)*, pp. 112–117. IEEE (2021)
- Koishybayev, I., Nahapetyan, A., Zachariah, R., Muralee, S., Reaves, B., Kapravelos, A., Machiry, A.: Characterizing the security of github {CI} workflows. In: *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2747–2763 (2022)
- Koneru, N.: Integrating security into ci/cd pipelines: A devsecops approach with sast, dast, and sca tools. *International Journal of Science and Research Archive* **3**, 250–265 (2021). <https://doi.org/10.30574/ijrsra.2021.3.1.0080>
- Lehto, M.: Cyber-attacks against critical infrastructure. In: *Cyber security: Critical infrastructure protection*, pp. 3–42. Springer (2022)
- Li, F., Paxson, V.: A large-scale empirical study of security patches. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215 (2017)
- Marjamäki, D.: Cppcheck – a tool for static c/c++ code analysis. <https://cppcheck.sourceforge.io/>. Accessed: 2025-06-15
- Martin, B.: Common vulnerabilities enumeration (cve), common weakness enumeration (cwe), and common quality enumeration (cqe) attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems. *ACM SIGAda Ada Letters* **38**(2), 9–42 (2019)
- Meli, M., McNiece, M.R., Reaves, B.: How bad can it git? characterizing secret leakage in public github repositories. In: *NDSS* (2019)
- MITRE: Cwe-327: Use of a broken or risky cryptographic algorithm. <https://cwe.mitre.org/data/definitions/327.html>. Accessed: 2025-06-20
- Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1186–1189. IEEE (2019)
- Perego, P.: Dawnsanner – the raising security scanner for ruby web applications. <https://github.com/thesp0nge/dawnsanner/>. Accessed: 2025-07-30
- Pereira, J.D., Vieira, M.: On the use of open-source c/c++ static analysis tools in large projects. In: *2020 16th European Dependable Computing Conference (EDCC)*, pp. 97–102. IEEE (2020)
- PyCQA: Welcome to bandit. <https://bandit.readthedocs.io/en/latest/index.html>. Accessed: 2025-06-15
- dehVCurtis: vulnerable-code-examples. <https://github.com/dehVCurtis/vulnerable-code-examples>. Accessed: 2025-07-30
- Design security: progipilot – a static application security testing (sast) for php. <https://github.com/designsecurity/progipilot>. Accessed: 2025-07-30
- Insider Application Security Team: Insider. <https://github.com/insidersec/insider>. Accessed: 2025-07-30

37. NCC Group Plc: Sobelow. <https://github.com/nccgroup/sobelow>. Accessed: 2025-07-30
38. OWASP Foundation: Owasp top ten. <https://owasp.org/www-project-top-ten/>. Accessed: 2025-07-30
39. Semgrep: Semgrep – code scanning at ludicrous speed. <https://github.com/semgrep/semgrep>. Accessed: 2025-07-30
40. Sharma, N., Sharma, S., et al.: A survey of mythril, a smart contract security analysis tool for evm bytecode. *Indian Journal of Natural Sciences* **13**(75), 51003–51010 (2022)
41. Sun, W., Fang, C., Miao, Y., You, Y., Yuan, M., Chen, Y., Zhang, Q., Guo, A., Chen, X., Liu, Y., et al.: Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv preprint arXiv:2312.00413* (2023)
42. Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: GPTScan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3597503.3639117>
43. Thompson, C., Wagner, D.: A large-scale study of modern code review and security in open source projects. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 83–92 (2017)
44. Tolaram, N.: Gosec and ast. In: *Software Development with Go: Cloud-Native Programming using Golang with Linux and Docker*, pp. 111–130. Springer (2022)
45. Wadhams, Z.D., Izurieta, C., Reinhold, A.M.: Barriers to using static application security testing (sast) tools: A literature review. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW '24*, p. 161–166. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3691621.3694947>
46. Wadhams, Z.D., Izurieta, C., Reinhold, A.M.: Barriers to using static application security testing (sast) tools: A literature review. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pp. 161–166 (2024)
47. Wheeler, D.: Flawfinder. <https://dwheeler.com/flawfinder/>. Accessed: 2025-07-30
48. Woo, S., Choi, E., Lee, H.: A large-scale analysis of the effectiveness of publicly reported security patches. *Computers & Security* **148**, 104181 (2025)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.