D3.3: Dynamic Testing Analysis Components (first version)

PROJECT	
Project Number	101120962
Project Acronym	RESCALE
Project Title	Revolutionised Enhanced Supply Chain Automation with Limited Threats Exposure
Start Date	01.10.2023
Programme	HORIZON-CL3-2022-CS-01-02
DELIVERABLE	
Deliverable Type	R - Document Report
Workpackage	WP3
Deliverable Lead	BNR
Editors	Narges Yousefnezhad, Andrei Costin
Contributors	BNR, UPRC, UNSPMF, ISI, AEGIS
Dissemination Level	PU - Public

Abstract

Deliverable D3.3 introduces the initial Dynamic Testing Analysis Components for the RESCALE project, combining insights from prior deliverables (D3.1 and D2.3). This module integrates hardware and software testing components to conduct dynamic assessments and report results to the management module. It also outlines features, functionality, and a roadmap for further development, with a final version to follow in D3.4 (M30).

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101120962

Document Revision & Quality Assurance

Internal Reviewers

1. Alan Barnett - (EISI)
2. Anastasios Gogos - (INT)

Revisions

Version	Date	Partner	Overview
0.1	09/07/2024	BNR	ToC
0.2	07/11/2024	BNR	First draft
0.3	17/11/2024	BNR, UPRC, UNSPMF, ISI, AEGIS	Finished draft
0.4	20/11/2024	EISI, INT	Reviewer comments
0.5	27/11/2024	BNR, UPRC, UNSPMF	Comments addressed
0.8	28/11/2024	BNR	Final version
0.9	06/12/2024	ISI, AEGIS	Final comments
1.0	13/12/2024	BNR	Final version

Table of Contents

List of Figures

List of Abbreviations

AI Artificial Intelligence. 21, 47

API Application Programming Interface. 9, 13–15, 21, 22, 24–27, 30, 31, 47

BFS Breadth-First Search. 23

BOM Bill of Materials. 32

CD Continuous Development. 10, 27, 46

CDX CycloneDX. 9, 18, 26, 32, 34–36, 48

CI Continuous Integration. 10, 18, 27, 46

CSV Comma Separated Values. 46

DAST Dynamic Application Security Testing. 11, 12, 15, 21

DL Deep Learning. 9, 13–16, 21, 33, 47, 48

DSCG Dynamic Supply Chain component Guarantee. 8, 9, 13–19, 23, 26, 27, 32, 33, 36, 45, 47, 48

FPGA Field-Programmable Gate Array. 15

HTTP HyperText Transfer Protocol. 22, 23, 25

IAST Interactive Application Security Testing. 11

JSON JavaScript Object Notation. 22–24, 26, 33, 34, 46

ML Machine Learning. 8, 9, 13–16, 21, 24, 33, 47, 48

OTR Open Test Reporting. 45, 46

RAISE REST API Intelligent Security Explorer. 13

RASP Runtime Application Self-Protection. 12

REST Representational State Transfer. 13, 22

SARIF Static Analysis Results Interchange Format. 46

SBOM Software Bill of Materials. 17, 34

SotA State of the Art. 10, 12, 14, 47, 48

SQL Structured Query Language. 12

SSCG Static Supply Chain component Guarantee. 8, 9, 17, 32, 33

SUT System Under Test. 25, 27, 29, 30

TAP Test Anything Protocol. 45, 46

TBOM Trusted Bill of Materials. 8, 9, 13, 32, 45, 47, 48

URL Uniform Resource Locator. 19, 26

XML Extensible Markup Language. 33, 34, 45, 46

XSS Cross-Site Scripting. 12

1 Introduction

This deliverable presents the initial version of the Dynamic Testing Analysis Module, a core component of the RESCALE project's efforts to enhance cybersecurity and vulnerability detection across software and hardware components. Designed to address both low-level and high-level security analysis needs, the Dynamic Testing Analysis Module integrates multiple testing techniques, supported by Machine Learning (ML), to provide comprehensive assessment capabilities. By identifying vulnerabilities in dynamic environments, this module strengthens the overall security framework of the software supply chain. This introduction outlines the scope and contributions of this deliverable, its connections to other project work packages and objectives, and the structure of the document, providing a clear pathway for understanding the module's role and projected impact.

1.1 Scope & Contribution

Complementing the Static Code Analysis Module, the RESCALE Dynamic Testing Module equips the platform with tools and methods to identify security vulnerabilities in both hardware and software components across the supply chain. This module supports multiple dynamic testing approaches tailored to various component types, covering three main areas: Low-Level Hardware Assessment, Dynamic Hardware Analysis and Dynamic Software Testing. Together, these tasks enable comprehensive runtime security analysis and contribute to the generation of the Dynamic Supply Chain Component Guarantee (DSCG), ensuring a robust security evaluation for the RESCALE platform.

1.2 Relation to Work Packages, Deliverables, and Activities

Deliverable D3.3 is the result of Task 3.2 within Work Package 3 (WP3) and is also pertinent to Work Packages 4 (WP4) and 5 (WP5). This deliverable is closely tied to the integration of tools associated with the Dynamic Testing Module, with specific integration aspects linked to WP5. Additionally, Task 3.2 contributes to the establishment of a DSCG report, which along with SSCG will contribute to the generation of a Trusted Bill of Materials (TBOM) file, adhering to the relative specification defined in WP4.

1.3 Contribution to WP3 and Project Objectives

Deliverable D3.3, a primary result of WP3 T3.2, aligns with Objective O3.3 to develop dynamic analysis tools that employ ML techniques for enhanced input selection. This deliverable will also lay foundational work for D3.5 and D3.7, particularly in incorporating Low-Level System Security Testing, and Hardware Vulnerability & Information Leakage Detection within the DSCG.

Task 3.2, in tandem with Task 3.1, will focus on creating dynamic testing analysis modules aimed at software and firmware components. Central to this task is the design development

of techniques using Machine Learning (ML) and Deep Learning (DL), techniques designed to produce targeted test vectors for black box and grey box dynamic testing at high-/low-level software, firmware, and hardware. These advanced techniques, paired with mutation fuzzers, should enable strategic input generation that extends beyond random uncontrolled inputs, thus enhancing testing coverage and eventually faster and wider vulnerability discovery. Specifically tailored for cybersecurity testing, this approach will include automatic fuzzing rule generation from REST Application Programming Interface (API) schemas of web services and mutation-driven test cases to detect vulnerabilities by analyzing the survival of specific mutants. In line with Task 3.1, the Task 3.2 will generate a trusted dynamic analysis report (DSCG), which will in the end be incorporated into the final TBOM structure and TBOM-associated immutable accountability chains.

1.4 Document Structure

This section outlines the structure of the deliverable as follows:

- **Section 2 - State of the Art Analysis**: Provides background and context, summarizing D2.3 and presenting additional relevant state-of-the-art research that informed the creation of the Dynamic Testing Module.
- **Section 3 - Dynamic Testing Module**: Offers an overview of the Dynamic Testing Module, detailing its components and specifying which tools apply to each component.
- **Section 4 - Dynamic Software Testing Analysis**: Focuses on software testing, explaining the internal workings of the dynamic testing module.
- **Section 5 - DSCG Design and Generation**: Discusses the DSCG design, covering its requirements, the initial CycloneDX (CDX) based draft, its integration and interplay with the Static Supply Chain component Guarantee (SSCG), and their combined role in creating the final TBOM.
- **Section 6 - Main Innovations & Conclusion**: Summarizes key innovations of the work and outlines next steps to conclude the deliverable.

2 State of the Art Analysis

This section presents a summary of the State of the Art (SotA) findings from Deliverable D2.3, providing essential context for the subsequent parts of this work. By examining the current landscape of relevant research, technologies, and methodologies, we establish a robust foundation for the following sections. This context is crucial for comprehending the advancements and innovations we intend to introduce.

Additionally, in the section that extends beyond traditional SotA, we will discuss several techniques that have been utilized more frequently than those typically found in standard SotA analyses. These techniques not only prove to be practically applicable but also significantly improve existing solutions. By emphasizing these advanced methodologies, we aim to showcase their contributions to enhancing performance and efficiency within our field. This investigation will help identify shortcomings in current approaches and pave the way for our proposed enhancements, which draw from both established literature and our own innovative insights.

2.1 D2.3 SOTA Overview

In today's interconnected software landscape, securing the software supply chain has become paramount. With increasing reliance on third-party components and open-source dependencies, vulnerabilities within these interconnected systems expose organizations to significant risk. This overview examines the SotA tools and methodologies developed to address these challenges, evaluating their strengths and limitations in protecting against software supply chain vulnerabilities.

Traditional security measures struggle to keep pace with the complexity and scale of modern software ecosystems, often leaving critical gaps in visibility and control over supply chain elements. By understanding current capabilities and pinpointing areas in need of enhancement, organizations can create robust, adaptive security solutions that safeguard the entire software lifecycle.

- **Advancements and Gaps in Software Supply Chain Security**: Dynamic analysis and emulation tools, such as Anubis [10] and Joebox [12], provide insights into vulnerabilities within software dependencies by simulating code behavior. However, challenges related to scalability, cross-platform compatibility, and integration in CI/CD pipelines limit their usability for continuous monitoring across diverse environments.
- **Binary Analysis and Static Code Review**: Tools like angr [14] and KARONTE [3], are essential for securing third-party binaries that lack source code. These tools help in identifying critical vulnerabilities, but their high computational demands and limited capabilities against obfuscation techniques constrain their deployment in real-time CI/CD workflows. Increased efficiency and adaptability would broaden their application across various architectures and code types.
- **Fuzzing and Automated Testing**: Frameworks such as FirmFuzz [15] and LABRADOR [18] aid in identifying hidden vulnerabilities in software dependencies. Despite their strengths, these tools often struggle to emulate realistic input patterns for certain dependencies, and

their resource intensity complicates integration into automated pipelines. Improved resource efficiency and adaptive fuzzing would enhance their effectiveness.

- **In-Situ and Non-Intrusive Testing**: In-Situ and Non-Intrusive Testing as exemplified by IPEA [16], enables testing within production environments to ensure continuous security, particularly for new dependency updates. While efficient, these tools often lack depth in detecting complex, multi-component vulnerabilities. Expanding their application range and increasing testing depth would improve their versatility for real-world deployment.

2.2 Dynamic Testing Approaches

Dynamic testing analysis involves the active assessment and monitoring of software while it is running to identify vulnerabilities, performance issues, or unexpected behaviors. Here are several examples of such techniques:

- **Fuzz Testing (Fuzzing) [22]:** This approach involves inputting random, malformed, or unexpected data into an application to uncover hidden bugs, security flaws, and crash points. For instance, fuzzing may reveal a buffer overflow vulnerability in a network protocol by sending improperly formatted packets.
- **Penetration Testing [11]**: This method simulates real-world attacks to identify security weaknesses from an attacker's perspective. For example, a dynamic penetration test on a web application could involve searching for injection vulnerabilities, improper session management, or insecure authentication processes.
- **Load and Stress Testing [7]:** These tests evaluate how an application performs under heavy load, helping to identify bottlenecks and resource limitations. A stress test on a server application, for instance, might reveal that it begins to reject requests when memory or CPU usage reaches critical levels.
- **Memory Leak Detection [5]:** Dynamic analysis tools like Valgrind [9] monitor an application's memory usage to detect leaks where memory is not properly released, particularly in C/C++ applications. This helps to prevent long-term performance degradation and stability issues.
- **Runtime Behavior Monitoring:** Tools in this category observe application behavior during execution to detect deviations, such as unauthorized file access or unexpected system calls. This method is effective in identifying anomalies or potential threats, including unauthorized access attempts or malicious activities.
- **Interactive Application Security Testing (IAST):** IAST merges aspects of DAST and static analysis by observing the application in real-time. For instance, IAST can detect vulnerabilities like SQL injection during typical usage of a web application by analyzing input flows and how data is processed.
- **Code Coverage Analysis:** This technique tracks which sections of the code are executed during testing. For instance, in a fuzz test, code coverage analysis ensures that as many paths as possible are tested, revealing untested segments that could conceal vulnerabilities.

- **Real-time Vulnerability Detection with RASP [21]:** Runtime Application Self-Protection (RASP) continuously monitors applications in production to detect and block attacks in real-time. This could include preventing SQL injection attempts by analyzing database queries as they are executed.
- **Dynamic Application Security Testing (DAST) [6]:** DAST tools assess the security of a running application by simulating attacks from the perspective of an external user. For instance, a DAST tool might simulate SQL injection or Cross-Site Scripting (XSS) attacks to test how the application handles potentially malicious inputs.

These dynamic testing analysis methods provide a proactive strategy for identifying and addressing security and performance issues in software during its actual operation.

2.3 Beyond State of Art

The "Beyond State of the Art" section in Deliverable D3.3 highlights how RESCALE advances the field of dynamic security testing, particularly in the context of supply chains. It identifies key limitations in existing methodologies and demonstrates how RESCALE's innovations address these gaps. This section sets the stage for detailing the project's unique contributions and their impact on improving supply chain security.

2.3.1 Current Limitations in SotA Approaches

In the realm of dynamic security testing, particularly concerning supply chain security, current SotA approaches exhibit several notable limitations:

- **Limited Coverage of Supply Chain Components**: Existing dynamic testing tools often focus primarily on the application layer, neglecting the intricate dependencies and integrations inherent in modern supply chains. This oversight can leave vulnerabilities in third-party components undetected, posing significant security risks [4].
- **Inadequate Real-Time Analysis**: Many current approaches lack the capability to perform real-time security assessments. This deficiency hinders the timely detection and mitigation of emerging threats within the supply chain, allowing vulnerabilities to persist unaddressed.
- **Insufficient Handling of Dynamic Code Injection**: Modern supply chains often involve dynamic code loading and execution. Traditional dynamic testing tools may not effectively detect or prevent malicious code injections that occur at runtime, leaving systems vulnerable to attacks [13].
- **High False Positive Rates**: The complexity of supply chain environments can lead to a high incidence of false positives in dynamic testing results. This issue can overwhelm security teams, diverting attention from genuine threats and reducing the overall efficiency of security operations.

2.3.2 Innovative Approaches in RESCALE

RESCALE introduces advanced methodologies to elevate dynamic security testing, particularly within supply chain contexts. The project integrates ML and DL to optimize test vector generation, shifting from random input methods to more targeted and efficient strategies. This innovation enhances the precision of vulnerability detection, addressing complex supply chain scenarios.

Furthermore, RESCALE combines fuzzing and mutation analysis within its dynamic testing framework. This dual approach allows for automated and continuously refined test cases, improving adaptability across both software and hardware components.

At the core of RESCALE's framework is the DSCG, which aggregates outputs from various testing modules to deliver comprehensive risk assessments. This real-time evaluation not only identifies vulnerabilities but also maintains ongoing security assurances. The DSCG is integrated with the TBOM, enhancing traceability and integrity throughout the supply chain lifecycle.

RESCALE also implements automated fuzzing rule generation from REST API schemas, enabling dynamic and context-aware API testing. This ensures robust and realistic vulnerability detection.

Lastly, the system's modular and scalable architecture supports the integration of diverse dynamic testing tools, allowing for comprehensive detection of runtime and logic vulnerabilities across various environments.

2.3.3 ML/DL Related Dynamic Testing

RESTler [2] is considered a state-of-the-art tool for fuzz testing REST APIs, primarily due to its ability to perform stateful, intelligent, and efficient fuzzing on complex API interactions. Unlike traditional fuzzers, which treat each request independently, RESTler builds upon the sequence of API requests to explore potential dependencies among endpoints. RESTler uses the API's OpenAPI specification to create a grammar that defines dependencies and generates test sequences aligned with the API's structure and usage. By prioritizing sequences based on endpoint dependencies, RESTler can detect vulnerabilities that are not accessible to traditional fuzzers.

However, RESTler has room for improvement – it's request generation is based on predefined heuristics and exhaustive search methods, which can result in redundant requests and consume significant computational resources.

Here we introduce our upgrade to RESTler – RAISE (**R**EST **A**PI **I**telligent **S**ecurity **E**xplorer). RAISE aims to introduce a more targeted and adaptive approach to RESTler by adding machine learning-based algorithms into the fuzzing process. In the initial stage, genetic algorithms will be applied to REST API fuzzing through RESTler, prioritizing test cases that are most likely to expose security vulnerabilities. RAISE will evaluate the "fitness" of each generated test request sequence based on its potential to reveal security issues. Sequences with higher fitness scores—those more likely to trigger unexpected responses or error states—will be prioritized in subsequent rounds of fuzzing. This way, RAISE will minimize false positives and save

computational resources. As the project progresses, other types of machine-learning models (e.g., based on reinforcement learning) will be added to RAISE.

Our second contribution is related to the EvoMaster [1], a tool that takes a related, yet different approach than RESTler. EvoMaster represents a novel integration between three tools:

- EvoMaster – an open-source tool that can automatically generate system level test cases for web and enterprise applications, supporting both black box and white box testing using Evolutionary Algorithms enhanced with Adaptive Hypermutation. Its primary purpose is to detect errors, security vulnerabilities, and performance issues in web services and REST APIs.
- FuzzDB – a dictionary of attack patterns and primitives for black-box application fault injection and resource discovery, and
- EvoMaster is a code coverage tool that generates test cases for REST APIs using evolutionary algorithms.

The solution provides two innovative contributions:

- The integration of EvoMaster with the FuzzDB repository via a custom-built converter, and
- The use of machine learning for fuzzing vector evolution in black box mode.

2.3.4 Enhancement over Existing Solutions

RESCALE builds on existing dynamic testing methodologies by introducing several enhancements that directly address limitations in current SotA solutions. Traditional approaches often face challenges such as limited real-time analysis, high false positive rates and poor scalability. RESCALE's innovations, including the DSCG, the integration of ML and DL for test optimization and automated fuzzing rule generation, represent significant improvements over these existing methods.

In contrast to conventional dynamic testing frameworks, RESCALE's modular architecture supports seamless integration of diverse tools, allowing for comprehensive analysis across software and hardware components. This modularity ensures adaptability to various operational environments, an area where traditional systems frequently fall short.

Moreover, RESCALE introduces advanced techniques for reducing false positives, such as leveraging ML to refine test vectors and employing mutation-based fuzzing to prioritize high-risk vulnerabilities. These methods enhance accuracy, thereby reducing the overhead associated with manual result validation, a notable drawback in existing solutions.

Finally, by incorporating continuous validation mechanisms and real-time assessments through the DSCG, RESCALE addresses the dynamic nature of supply chain security risks more effectively than static or less adaptive systems. This continuous monitoring framework ensures that security assurances remain robust as new threats emerge.

3 Dynamic Testing Module

This section begins by revisiting the information provided in D2.5, offering an overview of the Dynamic Testing module, including a breakdown of its components and the technologies integrated within each. It outlines how each component is distributed across various deliverables, with a particular focus on dynamic software testing analysis in the current deliverable. Additionally, this section delves into the detailed implementation of the module and provides insights about the input and output requirements for each tool involved.

3.1 Architecture

The RESCALE Dynamic Testing module identifies vulnerabilities across hardware, firmware, and software in the supply chain, employing various testing techniques for each component type. Figure 1 illustrates the dynamic testing module along with its submodules, as part of the high-level architecture outlined in Deliverable D2.5. Key elements include a Low-level System Components Assessment to detect microarchitectural and OS vulnerabilities, a Dynamic Hardware Analyzer to identify software and hardware vulnerabilities on Hardware-type of attacks (eg. Side Channel Attacks), and a Dynamic Software Testing component that uses ML/DL enhanced fuzzing tools for black-box and gray-box testing. The DSCG Generator consolidates outputs from these subcomponents to ensure a comprehensive risk evaluation.

The **Low-level System Components Assessment** addresses system security by targeting microarchitectural issues like Spectre-type attacks (a speculative execution CPU vulnerabilities) and OS vulnerabilities such as double fetches, which could compromise kernel security. The **Dynamic Hardware Analyzer** focuses on side-channel attack risks, particularly for cryptographic IP cores, leveraging a Trace Collector to gather leakage data through specialized sensors and FPGA-based hardware. This data is processed by a Trace Analyzer to detect hardware-based leaks from timing or power fluctuations.

The **Dynamic Software Testing** component applies various methods, such as DAST, fuzzing, and symbolic execution to identify software vulnerabilities, with flexibility for black-box or gray-box testing environments. ML/DL-driven tools enhance testing efficiency by analyzing artefacts like logs and crash dumps. The component supports RESCALE's microservices framework through modular, containerized tools and APIs.

The **DSCG Generator** secures the supply chain by continuously assessing the integrity, compliance, and performance of all components. It aggregates metrics from all testing modules to provide dynamic security assurances, monitors for new risks, and updates the RESCALE repository with detailed reports, ensuring an ongoing assessment of supply chain security.

3.2 Mapping Deliverables to Dynamic Testing Module Components

Each component of the Dynamic Testing module is documented across multiple deliverables, providing a detailed exploration of their roles, functions, and implementations within the RESCALE

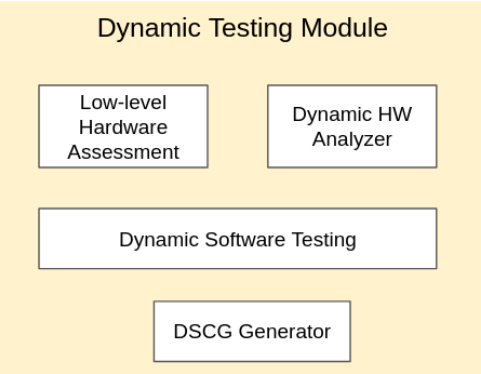

Figure 1: High-level Architecture of Dynamic Testing Module

architecture. The **Low-level System Components Assessment** component, responsible for testing security vulnerabilities in low-level system elements like microarchitectures and privileged software, is covered in detail in deliverable **D3.7**, titled "Solutions for Security Testing of Low-level System Components". This deliverable discusses the methods and tools specifically designed to identify microarchitectural vulnerabilities (such as Spectre-type attacks) and OS-level issues that could compromise system security. One such tool, *InSpectre Gadget*, is discussed for its ability to identify potential Spectre-related vulnerabilities and assess exploitability.

The **Dynamic Hardware Analyzer**, which focuses on detecting hardware or software IP Core vulnerabilities to hardware-type -f attacks through side channel leakage, particularly in cryptographic IP cores, is presented in **D3.5**, "Hardware Vulnerability and Leakage Detection". This deliverable covers side-channel attack detection methods, including those targeting post-quantum cryptography schemes, and describes tools such as trace collection and analysis for gathering and processing leakage data that may reveal hardware-type of attack vulnerabilities.

The remaining sections of current deliverable, **D3.3**, provides comprehensive coverage of the remaining components within the Dynamic Testing module: the **Dynamic Software Testing** component and the **DSCG Generator**. Accordingly, in **Section 3.3**, the focus shifts to the internal interactions within the Dynamic Software Testing component, detailing how its subtools and configurations work together to effectively test and analyze security aspects of the software under examination. More details regarding the tools working on dynamic software testing analysis comes in section 4. This section also explores how ML/DL models and testing artefacts (such as logs, crash dumps, and stack traces) contribute to an adaptive, efficient testing process.

3.3 Dynamic Software Testing Analysis

This subsection centers on dynamic software analysis, the primary contribution of the current deliverable. It begins by introducing and detailing the proposed architecture for dynamic software testing analysis. Following this, implementation specifics are outlined, including how inputs and outputs for each analyzer tool are organized and managed within the dynamic software analysis component.

Typically, the process leading to the dynamic testing module begins indirectly with the *pro-*

ducer performing a static code analysis on the software component. This step is critical for identifying potential security vulnerabilities at the code level before deployment. During this analysis, the *producer* generates a Software Bill of Materials (SBOM), which catalogs the components, dependencies, and libraries involved in the software. Alongside the SBOM, the *producer* compiles the static analysis results, detailing any detected issues and security risks. This information is sent to the SSCG Generator to create a report, integrating the analyzer results for the software. The result report is then stored in relatives repositories run by TrustOR as a record of the static analysis phase, ensuring that all upstream security information is available and traceable.

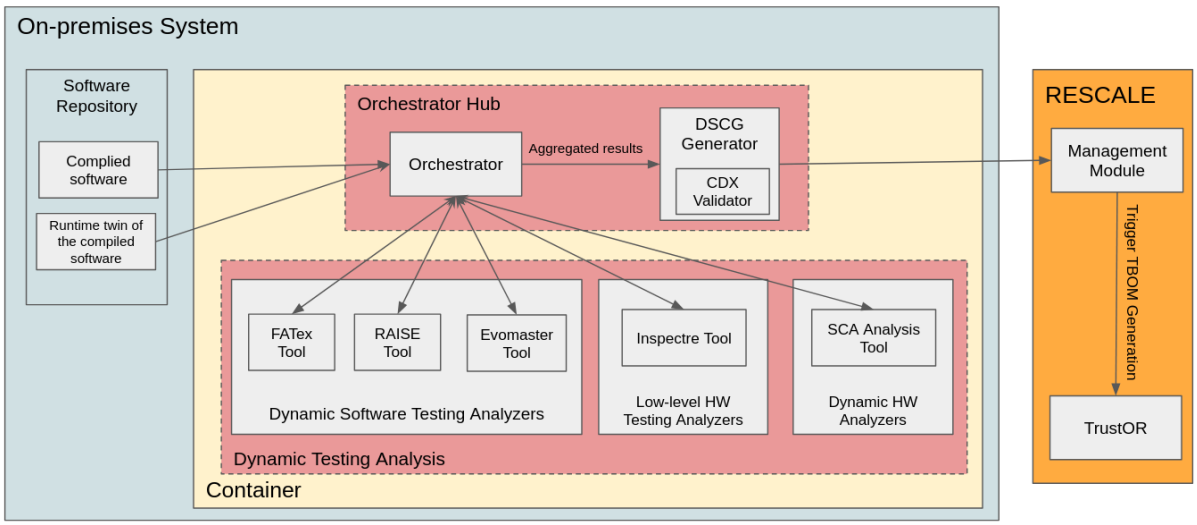

Figure 2: Preliminary Internal Architecture of Dynamic Testing Module

Once the static code analysis stage is complete, the consumer (often a downstream developer or integrator) accesses the dashboard for the next phase, which involves dynamic testing. Through the dashboard, the consumer can download the relevant docker container for the Dynamic Testing module, which includes all necessary tools and configurations for comprehensive dynamic security testing. After downloading, the consumer builds the dynamic module container locally to prepare for the testing process. As seen in Figure 2, the dynamic testing container requires two key inputs: the compiled/binary software/firmware to be tested and a runtime twin, which represents a live or running instance of the system where the software will eventually operate. It specifically refers to the active instance of the software undergoing testing or analysis. The ”runtime twin” is created and managed by the producer (or software owner) to facilitate dynamic analysis. It is typically hosted on the producer’s servers, premises, or infrastructure.

With the container prepared, the consumer then selects specific tool(s) to perform the dynamic testing based on descriptions available on the dashboard. These descriptions provide insights into each tool’s capabilities, helping the consumer choose the most suitable option for their security requirements. The chosen tool(s) are then executed within the dynamic testing container, applying techniques such as fuzzing, SCA Analysis, vulnerability and leakage detection to identify runtime vulnerabilities that may not have been detectable through static analysis.

Once the dynamic testing analysis is complete, the results are forwarded to the DSCG Generator, which compiles these findings into DSCG. This DSCG serves as a security validation for the runtime aspects of the software, ensuring that vulnerabilities detected during dynamic

testing are properly documented and addressed. The DSCG Generator validates the report, confirming that it adheres to the CycloneDX (CDX) format, a standardized structure for reporting security information. After verification, the DSCG report is finalized and ready for distribution.

The consumer then uploads the completed DSCG report to the management module to trigger the TBOM generation phase.

3.3.1 Implementation Approach

We are implementing a **single-container orchestrator-based** approach to streamline the process of dynamic software testing analysis. In this architecture, all analyzer tools—responsible for identifying vulnerabilities and potential security issues within the software—are managed within a single Docker container. This approach simplifies deployment and integration, making it especially well-suited for CI pipelines.

However, to ensure flexibility and adaptability, we are also considering a multiple-container architecture as complementary option. In a multiple-container solution, each analyzer tool would run within its own dedicated container, allowing for more granular control over resource allocation, isolation, and scaling. This design can be advantageous in scenarios where specific tools require distinct configurations, dependencies, or runtime environments that might conflict if run in a single container.

While our primary focus remains on the single-container approach due to its simplicity and ease of deployment, the multiple-container alternative provides a contingency plan, ensuring that our architecture can adapt to evolving requirements or constraints in more complex testing environments.

At the core of the single-container setup is an Orchestrator module within the container. This orchestrator functions as the central management layer, dynamically selecting and managing the execution of the desired analyzer tools based on user inputs or CI pipeline commands. Users interact with the system via a command-line interface, specifying the tools to run and configuring their parameters. The orchestrator then coordinates the execution of the selected tools, ensuring efficient resource utilization and minimizing overhead by consolidating all operations in a single environment.

In addition to managing tool selection and execution, the orchestrator plays a pivotal role in results aggregation and processing. Once the analysis is complete, the orchestrator gathers and formats the outputs from all activated tools, ensuring they are properly organized and passed to the DSCG Generator module. This generator, also part of the container, compiles the results into a unified DSCG report. The orchestrator oversees this process, verifying the completeness and consistency of inputs and ensuring the final output aligns with the required CDX standards.

By consolidating all tools and processes within a single container, this architecture significantly reduces complexity, improves maintainability, and facilitates seamless integration into automated workflows. The single-container orchestrator approach not only simplifies coordination between different testing tools and the DSCG Generator but also ensures a cohesive and streamlined dynamic testing process.

3.4 Input/Output Coordination

The Dynamic Analysis process in general is highly-specific to the corresponding application types and inputs, therefore the input and the actual flow of the dynamic analysis will vary depending on the input. Therefore, the description of each tool/sub-module operations as well as its inputs and outputs are detailed in this section (see Table 1). The dashboard interface plays an essential role in guiding users on the specific input formats expected for each tool, ensuring that users are well-informed and can select and configure tools accurately based on their needs. This pre-configured guidance allows for seamless integration, even though the tools may have varying input formats. Similarly, while each tool may produce different output formats, this does not affect the process as the DSCG Generator is designed to handle diverse output types. It simply takes the relevant content from each tool's output, processes it, and transforms it into a unified DSCG format, ensuring consistency in the final security report.

Tool	**Input**	**Output**
RAISE	Swagger JSON	JSON
InSpectre (D3.7)	Kernel files (ELF)	CSV
FATex	Firmware binary package (bin)	Text
Evomaster	Swagger JSON	JSON
SCA Analysis (D3.5)	JSON	JSON

Table 1: Preliminary expected Inputs and Outputs of Dynamic Testing sub-modules/tools

As a general principle, each tool or sub-module will first validate the input it receives. If the input is incomplete or not as expected by the tool or sub-module, a graceful exit will occur, and a meaningful, consistent error will be returned to the RESCALE platform flow. If the input is valid and expected, the tool or sub-module will proceed with any additional required validation of the inputs, followed by executing its business logic to process the input appropriately and identify vulnerabilities within its scope. For example, the scope of vulnerabilities identified by the SCA hardware leakage tool differs significantly from that of the EvoMaster API fuzzing tool, leading to variations in their inputs, internal processes, and business logic.

For example, the FATex tool takes a binary firmware as input, which it attempts to emulate before executing a series of tests and assessments to identify vulnerabilities in the firmware, such as command injection, privilege escalation, XSS, etc., either through the web interface of the firmware or other interfaces (e.g., SSH). In another example, the EvoMaster tool primarily takes a URL as input and performs a series of fuzzing tests to make the API core under test misbehave, thereby exposing specific vulnerabilities such as information disclosure, access token disclosure, privilege escalation, etc. Moreover, DAISE tool executes specific kernel tests, creating and managing a runtime environment for kernel analysis. This runtime twin is exclusive to the Inspectre tool and typically not accessible by other tools.

3.4.1 The Purpose of Runtime Twin

As shown in Figure 2, there is an additional input to the dynamic testing module called the runtime twin, which is provided by the producer. A runtime twin refers to a live or running instance of the system under analysis, providing a realistic and interactive testing environment.

The runtime twin enables dynamic analysis and validation, facilitating the identification of vulnerabilities in diverse system contexts. Depending on the use case, the runtime twin may vary in implementation and management. Below we provide some guiding examples on the usage and management of the runtime twin for some major scenarios (nevertheless this is not an exhaustive list of all possibilities):

- IoT Firmware: The runtime twin is created and managed by the FATex tool, hosted within an emulator Docker instance (e.g., powered by QEMU, FIRMADYNE and similar). The twin is generally accessed for assessment via IP or URL, and other dynamic analysis tools may leverage this twin for testing and validation of CVEs, exploits, or generic supply-chain and conformance claims.
- Inspectre Tool: The runtime twin represents the specific execution instance of the kernel under test. It is created and managed solely by the Inspectre tool and is typically not shared with other tools in the dynamic analysis module.
- Hardware Devices and Boards: The runtime twin is the physical hardware or board running the software/firmware under analysis. Hardware owner or certification evaluators may perform both static and dynamic analysis, using IP-based protocols to provide access to dynamic analysis tools. The runtime twin is managed by hardware owner or certification evaluators, ensuring accessibility for testing through connected devices.

The runtime twin can be specified using URLs (e.g., for web interfaces or RESTful APIs), IP addresses (e.g., for SSH or other network-level services and interfaces), or other standardized access identifiers. These inputs allow dynamic analysis tools to interface with the runtime twin seamlessly, regardless of whether the environment is a physical device, emulated firmware, or a proprietary application hosted on the producer's infrastructure. In cases where software is proprietary and non-distributable, the runtime twin may be limited to an API endpoint specified via URL. This approach provides access to a realistic testing environment without sharing source code or binaries. Eventually, the purpose of the runtime twin across all scenarios remains consistent: to provide an environment that facilitates dynamic analysis, validates potential exploits, finds vulnerabilities within the system under test, and confirms generic supply-chain and conformance claims.

4 Dynamic Software Testing Analysis Components

The Dynamic Software Testing component integrates a suite of tools, many enhanced with ML/DL capabilities, to provide comprehensive security insights across multiple programming languages. This module efficiently identifies and mitigates vulnerabilities, bugs, and exploit risks across both software and hardware components. Leveraging techniques such as DAST, fuzzing, symbolic execution, and AI-driven analysis, it aims to uncover potential vulnerabilities in software systems. The following subsections provide further details on the specific technologies used within each tool, as well as the configuration methods that adapt these tools to meet diverse testing requirements.

4.1 RAISE

Here we introduce an advanced, dynamic REST API fuzzing component **RAISE** - **R**est **A**pi **I**ntelligent **S**ecurity **E**xplorer. RAISE is based on RESTler [2], a comprehensive API fuzzing tool developed by Microsoft. Our approach extends the mentioned tool's capabilities by integrating a ML model to optimize fuzzing efficiency and precision. This enhancement uses machine learning approaches (currently genetic algorithms, which are under development) to prioritize high-risk and complex API paths, improving RESTler's ability to uncover potential bugs, security vulnerabilities, and performance issues within API workflows. Figure 3 shows a diagram illustrating the interaction and integration of RAISE and RESTler.

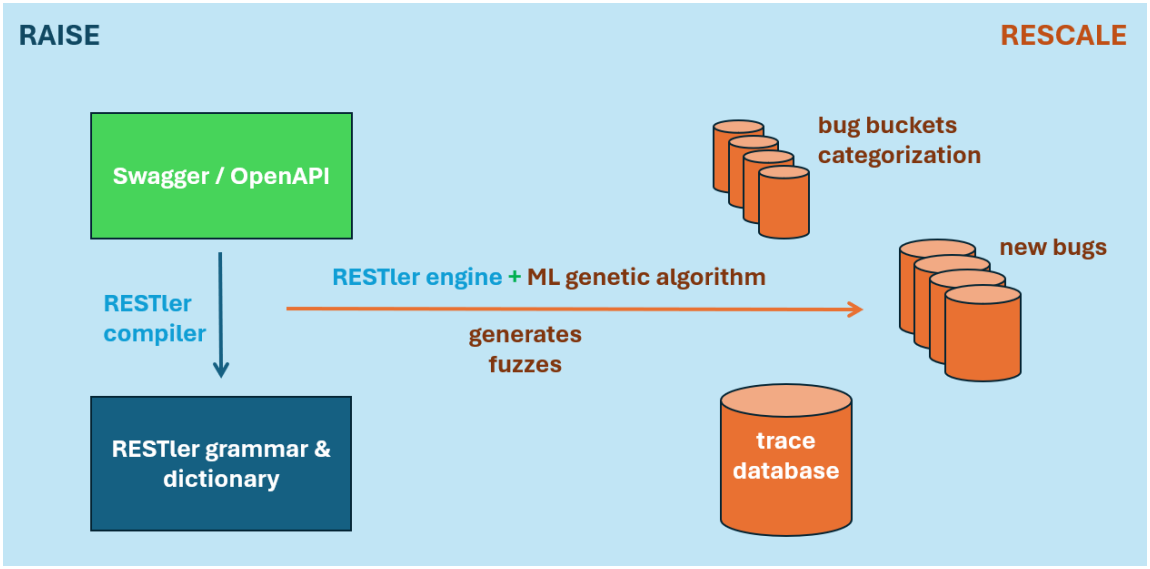

Figure 3: RAISE integration with RESTler

We utilize RESTler's compiler to generate a grammar from the provided Swagger specification for the target service's REST API. Following this, we enhance the RESTler engine with our genetic algorithm component, which optimizes the fuzzing process and enables security-tuned bug categorization.

4.1.1 Architecture

The tool is built to perform intelligent REST API fuzzing enhanced with machine-learning genetic algorithms. The tool's architecture could be divided into the following parts:

- **Input & Grammar**
  To fuzz a targeted REST API service, we first need to supply an OpenAPI JSON specification (also known as a Swagger specification) as input. Using this specification, the tool constructs a unique grammar that defines a set of rules for generating sequences of HTTP requests. These request sequences are then directed to the REST API, allowing the tool to systematically explore the API's functionality and interactions between the requests and responses.
- **Generating request sequences**
  Instead of sending isolated requests, RAISE sends a sequence of requests based on a genetic algorithm. The algorithm helps to generate sequences that have relevance in discovering new security issues and bugs. It leverages a genetic algorithm by applying evolutionary principles of selection, crossover, and mutation to iteratively optimize the request sequences. The algorithm uses a set of rules obtained from the generated grammar and selectively refines these sequences based on their effectiveness, thus enhancing its precision and coverage over time. RAISE will keep a history of sequences and their responses which are not only useful for genetic algorithm development, but also for manual reruns.
- **Genetic algorithm**
  Here is the main part of the RAISE algorithm that is being incorporated into RESTler:
  - **fitness function**
    is based on response code, error type, and length of request sequence, while evaluating request sequences;
  - **selection process**
    once each sequence's fitness score is calculated, the algorithm selects the most effective sequences to use as "parents" in generating the next generation of sequences. The selection process is designed to favor sequences with higher fitness scores, but also introduces some randomness to maintain genetic diversity;
  - **crossover & mutation**
    crossover combines segments from two parent sequences to create new offspring sequences, blending successful elements from both parents. Mutation introduces random alterations within sequences, such as modifying HTTP methods (e.g., changing POST to PUT), altering parameter values, and reordering requests.
- **Garbage collector**
  After responses are logged, the garbage collector deletes objects that are created during fuzzing, e.g., when the POST statement is executed and creates a new object, the object is afterwards deleted, so the fuzzing service remains unchanged after the fuzzing is finished.
- **Error detection & logging**
  The tool monitors API response in order to detect unusual status codes and behavior.

It logs errors and abnormalities, and sorts them into the bug buckets depending on the HTTP response code. Here is an example:


```
"500": {
    "225bf738-db26-41f6-88db-795b209f6c3a": [
        {
            "request": {
                "RequestData": {
                    "method": "GET",
                    "path": "/api/blog/posts/448521",
                    "query": "",
                    "body": ""
                }
            },
            "response": {
                "ResponseData": {
                    "code": 500,
                    "codeDescription": "Internal Server Error",
                    "content": "Internal Server Error",
                    "isFailure": true,
                    "isBug": true
                }
            }
        }
    ]
}
```


4.1.2 Implementation

The implementation of RAISE can be viewed from two perspectives: its integration with RESTler and its connection to the DSCG Generator component. RAISE is incorporated into RESTler as a new module, redirecting the fuzzing process to select the genetic algorithm module instead of the traditional random walk or Breadth-First Search (BFS) options. For details on the genetic algorithm implementation, see Section 4.1.1. Interaction with the DSCG Generator is described in Section 4.1.4.

4.1.3 Evaluation Results

RAISE results are divided into bug buckets that are listed in the JSON file, see the example in Section 4.1.1 (Error detection & logging). With the RAISE additional algorithm, the outcome file is expected to have a smaller number of false positive errors. The model is still under evaluation.

4.1.4 Interconnection/Interfaces

Connecting with RAISE is very simple since it is a dockerized component:

```
docker run -e SWAGGER_FILE_NAME=your-swagger-file-name.json \
  -v $(pwd)/data/Swagger-Spec:/home/raise-repo/working-dir/Import \
  -v $(pwd)/data/Raise-Results:/home/raise-repo/working-dir/Export \
  --name unspmf-raise-container \
  unspmf-raise
```

Three parameters have to be provided in order to run the Docker container successfully:

- name of the Swagger specification file;
- directory where the Swagger specification JSON file is located;
- name of the directory where RAISE will export the error bucket results.

Another way to start the RASIE is to download the folder that contains all of its components and to run it in CLI, with the following command:

```
$python unspmf-raise.py --swagger-file path_to/swagger.json
```

Here, the unspmf-raise.py script serves as a wrapper for executing shell commands.

A subsequent version of RAISE will have the option to fuzz a secured service via a token.

4.2 EvoMaster

The API fuzzing tool which is designed and implemented in the context of Task 3.2 for security testing includes a novel integration of three tools and ML models. The tools are EvoMaster, FuzzDB and coverage.py in case of python application to be tested. EvoMaster is the core tool in this part of the solution. It is an open-source tool that can automatically generate system level test cases for web and enterprise applications. It supports both black box and white box testing using Evolutionary Algorithms enhanced with Adaptive Hypermutation. Its primary purpose is to detect errors, security vulnerabilities, and performance issues in web services and REST APIs. FuzzDB is a dictionary of attack patterns and primitives for black-box application fault injection and resource discovery. A code coverage tool is used to feed the algorithm with usage info for the next test generation through the iterations. In the context of RESCALE, EvoMaster is used for black box testing on REST APIs, where Random Search is the default algorithm. The innovation in the solution consists of two factors. The integration of EvoMaster with FuzzDB payload repository and the use of ML for fuzzing vector evolution in black box mode. Which actually is GreyBox Testing. A converter is built in the module to implement the integration of the core tool with the FuzzDB repository. The solution will also be integrated with a code coverage tool. The integration with this tool is established by the Parser which communicates with coverage plugin. The final version of this tool will include test generation for database injection vulnerabilities in black box mode and automatically mocking external web services to avoid the noise by them. Machine Learning Model will be used on top of the feedback by the tools integrated to improve the efficiency of the whole process of testing the API.

Our tool uses EvoMaster to generate test cases by simulating an evolutionary process:

- It starts with an initial set of randomly generated test cases.
- Each test case is evaluated against the application using a fitness function, which scores how effective the test is at exploring code paths and finding bugs.
- Our tool then ”mutates” and ”recombines” these test cases in a way similar to genetic evolution, creating a new generation of tests.
- This process repeats over multiple generations, gradually evolving tests that cover more code paths and expose more potential issues.

The tool is particularly effective for RESTful web services and APIs. It automatically interacts with API endpoints, sending various inputs to simulate different use cases and edge cases. This approach helps uncover

- Unexpected responses or errors from the server.
- Potential security vulnerabilities like SQL injection or path traversal.
- Performance issues, as EvoMaster can observe response times and identify bottlenecks.

In summary, our tool is a sophisticated tool that applies evolutionary fuzzing to generate effective test cases for web APIs, aiding in the identification of bugs, security issues, and performance problems. Its evolutionary approach makes it adaptive and capable of generating nuanced tests that would be difficult to create manually.

4.2.1 Architecture

The figure 4 below outlines the architecture of the API Security Fuzzing tool (based on EvoMaster), highlighting its self-contained design and integration with key components. The container consists of EvoMaster core tool, Automation API, FuzzDB repository, Converter and the Parser. The code coverage tool is not included, since it has to run on the runtime of the SUT. EvoMaster core tool interacts with an Automation API which triggers the execution of the test. This service becomes available as soon as the tool container is built and run. The API schema of the SUT is provided through the Automation API by the process Initiator which is the Orchestrator of Dynamic Testing Module.

The module uses an initial set of configuration parameters (given through the Init API) to manage the analysis process:

USER_ACCESS_TOKEN : A user access token for secure communication with the RESCALE Management Module

ACCESS_AUTH_HEADER : In black-box testing, we still need to deal with authentication of the HTTP requests. With this parameter it is possible to specify a HTTP header that is going to be added to most requests.

MAX_TIME : (e.g. 60s) The maximum time(in seconds) that EvoMaster will run. The more the best. The time is expressed with a string where hours (`h`), minutes (`m`) and seconds (`s`)

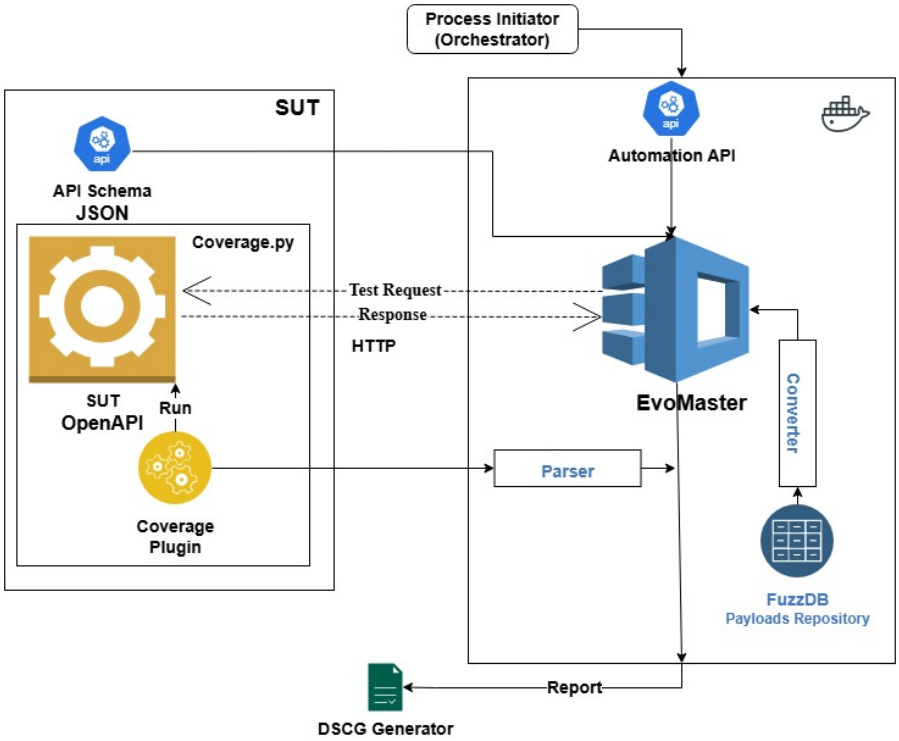

Figure 4: API Fuzzing Module Architecture

can be specified, e.g., `1h10m120s` and `72m` are both valid and equivalent. If not provided, the default value will be the 1 hour.

API_SCHEMA : The API Schema of the SUT in OpenAPI Specification 3.1.0 in JSON format. The base URL of the instance of the running SUT is included in the schema as well. The schema is used by EvoMaster to create syntactically correct test cases.

In parallel with this process, the code coverage tool is set up and running on the SUT instance using a plug-in to establish communication with the EvoMaster tool. The results of code coverage tool are sent to the Parser which handles the instrumentation with the generated tests. Parser is an instrumentation wrapper, added to the code to collect runtime information such as code coverage, exceptions, and performance data.

After executing the tests on the SUT, the results are sent to the Orchestrator. Orchestrator forwards the results to the DSCG Generator, where the DSCG, is compiled and formatted as a CDX document. The results include a summary report and tests generated so that reproducibility of requests on faulty or vulnerable endpoints is possible.

4.2.2 Implementation

The container of the tool uses Ubuntu as base and installs Java and Python with the relevant libraries. EvoMaster runs on any major LTS version (from JDK 8 on) while the other tools are implemented in Python. The code coverage tool shown in the architecture picture (Figure 1) is also for Python applications.

Overall Workflow

1. **Module Initialization**: In case of multiple containers in the Dynamic Testing Module, the container gets built and run. In case of a single container, this stage has already taken place.
2. **Code Coverage**: The code Coverage tool is set up running on the SUT runtime.
3. **Automation API**: The API is used by the Orchestrator to send to the EvoMaster module the parameters and the API Schema.
4. **EvoMaster run**: EvoMaster core tool runs integrated with FuzzDB.
5. **Test/Coverage Instrumentation**: The reports by EvoMaster and Code Coverage tool are collected and combined in the final report sent to the DSCG generator through the Orchestrator.

EvoMaster Workflow

1. **Initialization**: The EvoMaster Core loads a module based on the type of application (e.g., REST API).
2. **Test Generation**: EvoMaster uses evolutionary algorithms to generate initial test cases.
3. **Test Execution**: The execution engine runs these test cases against the SUT.
4. **Evaluation**: The fitness function evaluates the effectiveness of each test.
5. **Evolution**: EvoMaster refines the tests, evolving them to cover more paths or reach specific goals. In security testing mode, this function checks for unauthorized access, injection vulnerabilities, and exposure of sensitive data.
6. **Result Reporting**: Detailed reports are generated and sent to the CI/CD pipeline.

4.2.3 Evaluation Results

The tool is under evaluation on several types of SUTs and interfaces. Custom endpoints and existing vulnerable-by-design systems are used for this purpose. The integration between EvoMaster and FuzzDB and the use of ML model on the test generation through the iterations is expected to give even better results compared to the out of the box use of EvoMaster. An EvoMaster output report typically contains information on test execution results, including coverage statistics, detected bugs, and specific test cases generated during the fuzzing process. Here's an example of a simplified EvoMaster output report and a sample test case it might generate.

Apart from the summary report, EvoMaster creates files with the test cases generated. The tests are written in separate file depending on the success, failure or vulnerability detected. Here are some examples in Python.

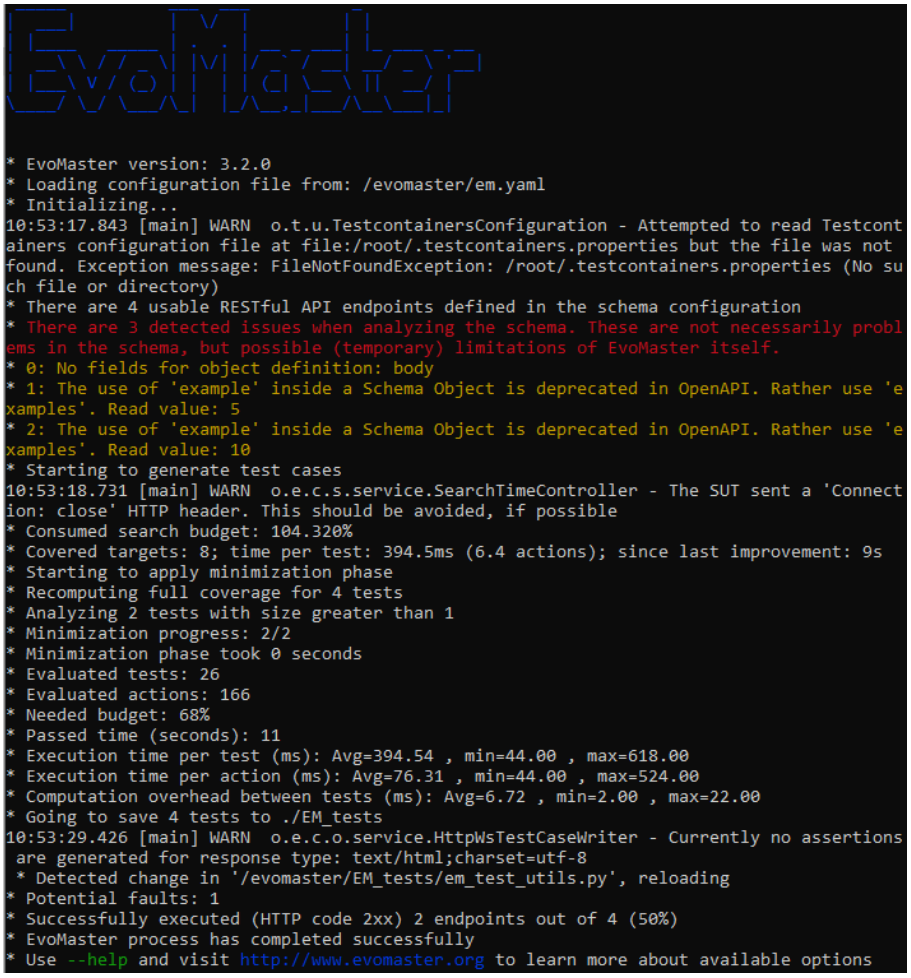

Figure 5: EvoMaster Summary Report

```python
#!/usr/bin/env python

import json
import unittest
import requests
import os
if os.name == 'nt':
    class timeout_decorator:
        @staticmethod
        def timeout(*args, **kwargs):
            return lambda f: f # return a no-op decorator
else:
    import timeout_decorator
from em_test_utils import *


#  This file was automatically generated by EvoMaster
#  The generated test suite contains 1 test
#  Covered targets: 4
#  Used time: 0h 0m 11s
#  Needed budget for current results: 68%
#  This file contains test cases that represent successful calls.
class testWS_successes_Test(unittest.TestCase):


    baseUrlOfSut = "http://host.docker.internal:5000"


    @timeout_decorator.timeout(60)
    def test_1(self):

        headers = {}
        headers['Accept'] = "application/json"
        res_0 = requests \
                .get(self.baseUrlOfSut + "/api/q9VjJdkxVJ",
                    headers=headers)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert res_0.json()["message"] == "Hello, q9VjJdkxVJ!"


if __name__ == '__main__':
    unittest.main()
```

Figure 6: EvoMaster Successful Test

The figure 6 shows a successful GET request to the /api end point of the SUT with a random string as parameter. This request had a response with status code 200.

```python
#!/usr/bin/env python

import json
import unittest
import requests
import os
if os.name == 'nt':
    class timeout_decorator:
        @staticmethod
        def timeout(*args, **kwargs):
            return lambda f: f # return a no-op decorator
else:
    import timeout_decorator
from em_test_utils import *

#  This file was automatically generated by EvoMaster
#  The generated test suite contains 1 tests
#  Covered targets: 3
#  Used time: 0h 0m 11s
#  Needed budget for current results: 68%
#  This file contains test cases that are likely to indicate faults.
class testWS_faults_Test(unittest.TestCase):

    baseUrlOfSut = "http://host.docker.internal:5000"

    @timeout_decorator.timeout(60)
    def test_0_with500(self):

        # Fault100. HTTP Status 500. POST:/api/addint
        headers = {}
        headers["content-type"] = "application/json"
        body = {}
        body = " { " + \
            " \"firstint\": 5, " + \
            " \"secondint\": 942 " + \
            " } "
        headers['Accept'] = "*/*"
        res_0 = requests \
                .post(self.baseUrlOfSut + "/api/addint",
                    headers=headers, data=body)

        assert res_0.status_code == 500
        assert "text/html" in res_0.headers["content-type"]


if __name__ == '__main__':
    unittest.main()
```

Figure 7: EvoMaster Faulty Test

The figure 7 shows a POST request to the /api/addint end point of the SUT. The response had a 500(internal server error) status code which means that there is a potential bug. The code can be used to reproduce the error.

4.2.4 Interconnection/Interfaces

EvoMaster tool is designed to be easily integrated with other tools and workflows. The first interaction which has to take place is the *build* and *run* commands of the container. Currently a batch file is used to do this. This part will be easily adapted in the environment of the Orchestrator. The EvoMaster tool has three basic interfaces implemented in web technologies (GET/POST requests). These services are used to run the tool, get the code coverage report and send the final report back to Orchestrator.

As soon as the container is up and running, a web service is up and running too. This service, which is called Automation API, expects to receive the parameters described in the architecture. The basic parameter is the API schema of the SUT. Then the web service runs the EvoMaster

and waits for it to finish. The same service collects the report which was exported by EvoMaster and sends it back to the Orchestrator. The other interface of the tool is the one with the code coverage tool. This is also a web service which gets up and running as soon as the EvoMaster runs on an API. This service is called Parser.

5 DSCG Design and Generation

DSCG is designed to capture the outcomes of prior dynamic software and hardware testing analyzers, along with relevant metadata, to ensure machine readability. Once the DSCG is generated, the Consumer sends it to the RESCALE Management Module to prepare for the next stage of the RESCALE project. In this next step, the Trust Orchestrator will be activated to create the TBOM. This section provides an overview of the DSCG generation process and details the expected output format of the DSCG for each component.

5.1 DSCG Generation

The generation process of the DSCG relies on the DSCG Generator, which is a crucial component of the Dynamic Testing Module. This generator is dedicated to ensuring the integrity and security of the supply chain for both hardware and software development. Its primary function is to dynamically assess and validate the security, compliance, and performance of various components and processes throughout the supply chain lifecycle. By producing assurance metrics and conducting rigorous checks, the DSCG Generator seeks to identify and mitigate potential risks.

The generator utilizes inputs from various components, including the Low-level Hardware Assessment, Dynamic Software Testing, and Dynamic Hardware Analyzer. By evaluating the results from these components, it generates dynamic reports that affirm the security and integrity of supply chain elements. Additionally, the DSCG Generator forwards the generated report (DSCG) to the State Manager and the TrustOR, with the report ultimately being securely stored in the RESCALE repositories.

As highlighted in D4.1 and D3.1, the CycloneDX (CDX) standard presents itself as a potentially suitable Bill of Materials (BOM) framework within the RESCALE initiative. CDX adheres to a high-level abstract specification that encompasses a broad array of metadata, dependencies, and workflows that affect the processing of the artefact, including its compilation. Initially designed as a simple BOM standard for capturing vulnerabilities in software artefacts, CDX has since evolved to address complex scenarios that involve entire service architectures, hardware components, and detailed environmental descriptions. To enhance integration with the SSCG and facilitate the subsequent creation of the TBOM, a CDX validator will be incorporated within the DSCG Generator to ensure that the generated DSCG report conforms to the CDX format.

In conclusion, the DSCG Generator is integral to assuring supply chain integrity and aligns with established standards such as CDX to enable thorough security assessments across a variety of artefacts and configurations.

5.2 Requirements

The DSCG Generator component has several key requirements that must be met to ensure its effectiveness in maintaining security and compliance within the supply chain.

1. **Security and Compliance Criteria:** The component requires clearly defined security and compliance standards for evaluating supply chain components. Its functionality relies on the preceding components' ability to accurately process binary files to identify patterns and potential risks.
2. **Integration with Other Components:** The DSCG Generator must interact effectively with other components within the Dynamic Testing Module:
    - **Collaboration with Low-level Hardware Assessment:** It will exchange hardware assessment data to ensure that the guarantees reflect the integrity of low-level hardware.
    - **Utilization of Results from Dynamic Hardware Analyzer and Dynamic Software Testing:** The DSCG Generator will correlate supply chain risks with hardware and software vulnerabilities, verifying that all components meet established security and compliance standards.
    - **Repository Interactions:** It will interact with DSCG DB and other repositories to store generated dynamic guarantees in appropriate databases and ledgers.
3. **Well-Defined Structure:** The report created by DSCG Generator must have a clearly structured format, including:
    - **Embedded Materials Information:** Detailed and clear information about embedded materials presented in a standardized exchange format (e.g., XML, JSON).
    - **Dynamic Testing Results:** These results should be formatted similarly to those of the SSCG, including certificates, relevant metadata, dependencies, conformance to standards, and additional requirements.
4. **Flexible Dynamic Analysis:** The DSCG Generator should support dynamic analysis of software and firmware through multiple optimization techniques, including:
    - **Classic and Fuzzing Techniques:** The capability to optimize the fuzzing process and mutation corpus by employing a combination of different dynamic testing techniques, such as classic, fuzzing, and ML/DL-guided approaches (e.g., classic + ML fuzzing).
    - **Side-Channel Analysis Techniques:** Leveraging side-channel data such as power consumption, electromagnetic emissions, or execution timing to identify hardware and firmware vulnerabilities. These techniques are particularly relevant to tools like the SCA Trace Analyzer, which assess vulnerabilities arising from unintended information leaks during execution.
5. **Results Reporting:** The component must generate a comprehensive report of dynamic analysis results, prioritizing identified issues based on their severity and potential impact on security.
6. **Common Data Formats:** All Dynamic Testing Modules and tools must provide data to the DSCG in a common, machine-readable format/schema. The RESCALE platform should support modules/tools that can ingest DSCG data from different analysis modules and generate a final, unified DSCG according to the specified format.

These requirements collectively ensure that the DSCG Generator can effectively monitor and guarantee the security and integrity of the supply chain throughout its lifecycle.

5.3 Initial DSCG Samples

The CDX SBOM format can be represented in JSON, XML, or protocol buffers. For simplicity, we'll use a streamlined CDX JSON format. The CDX JSON can contain various properties, such as metadata, components, services, and more. However, for the initial design, we opted for a simplified approach that aligns with the requirements outlined in Section 5.2, such as including metadata, dependencies, and ensuring conformance to standards. Therefore, only the essential specifications, as illustrated in Figure 8, have been included.

Figure 8: Highight of CycloneDX Specifications

The JSON file begins with general metadata about the file, followed by metadata on the software being analyzed.

```
"bomFormat": "CycloneDX",
"specVersion": "1.6",
"serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
"version": 1,
"metadata": {
  "timestamp": "2024-09-03T05:48:47.302Z",
  "authors": [
    {
      "name": "Some Company/person that wants to generate a DSCG for
         their binary",
      "email": "some@company.com"
```

```
      }
    ]
```

Additionally, Metadata specifies the target component that the CDX describes and their dependencies. It also includes information about the tools used to generate the CDX.

```
    "tools": {
      "components": [
        {
          "type": "application",
          "name": "ReSCALE DSCG Generator",
          "version": "1.4.17",
          "description": "A ReSCALE certified tool to generate DSCGs",
          "purl": "pkg:hex/dscg-generator@1.4.17",
          "data": [
            {
              "name": "CLI configuration flags",
              "type": "configuration",
              "contents": {
                "attachment": {
                  "content": "--test-report=report.json
                     --output=dscg.json"
                }
              }
            }
          ],
          "hashes": [
            {
              "alg": "SHA-1",
              "content": "3942447fac867ae5cdb3229b658f4d48"
            }
          ]
        },
        {
          "type": "container",
          "name": "ReSCALE dynamic Code Analysis Module",
          "version": "2.1.7",
          "description": "A ReSCALE certified container to execute
             dynamic testing",
          "purl": "pkg:docker/dynamic-code-analysis-module@2.1.7",
          "data": [
            {
              "name": "Docker Environment",
              "type": "configuration",
              "contents": {
                "attachment": {
                  "content": "RESCALE_DYNAMIC_ANALYSIS_LANG=erlang\
                  nRESCALE_DRY_RUN=false"
                }
              }
            }
          ],
          "hashes": [
            {
              "alg": "SHA-1",
              "content": "35d1c8f259129dc800ec8e073bb68f995424619c"
            }
          ]
```

```
        }
      ]
    }
```

The definitions will then be outlined, including standards, requirements, levels, and supporting documentation.

```
  "definitions": {
    "standards": [
      {
        "bom-ref": "https://rescale-project.eu/standard/1.0.0",
        "name": "The ReSCALE Standard",
        "description": "The ReSCALE Standard describes a workflow to
           create a Trusted BOM (TBOM)",
        "version": "1.0.0",
        "requirements": [
          {
            "bom-ref":
               "https://rescale-project.eu/standard/1.0.0/conformance/
            complete",
            "identifier": "rescale/1.0.0/conformance/complete",
            "title": "Full conformance with ReSCALEs 'complete'
               profile, e.g. complete absence of findings"
          }
        ]
      }
    ]
  }
```

Finally, there is a declaration detailing conformance to standards. Each declaration may include attestations, claims, counterclaims, evidence, and counter-evidence, along with conformance and confidence indicators. Declarations can also specify signatories, with support for both digital and analog signatures. These declarations establish the foundation for "compliance-as-code".

Declarations contain evidence confirming compliance or identifying gaps, with claims as hypotheses about specific requirements. Each claim supported by evidence, often presented in CDX format, same as test report data from static code analyzers. For instance, a claim might highlight a vulnerability with output evidence from a static analysis tool. In cases where no standards are set, claims are based on tool-specific outputs. CDX further allows metadata detailing, covering assessors, environment configurations, and tool usage specifics, including DSCG Generator information.

The previous fields presented here apply uniformly to all tools, but the DSCG reports we created for the three tools (FAT, Inspectre, and Raise) vary in the declaration field. As a result, there will be a unified version of the DSCG that integrates the results from all the analyzer tools. Additionally, we consider the scenario where each tool presents its own DSCG, with each having its own unique declarations, and the DSCG generator will then consolidate these results. To this end, subsections 5.3.1, 5.3.2, and 5.3.3 outline the declaration fields for each tool, which include assessors, attestations, claims, and evidence.

- Assessors: These are entities or individuals responsible for evaluating the claims made about the analyzed artifact. Assessors determine whether these claims conform to the

defined security and compliance requirements and assess the confidence level of their evaluation.

- Attestations: An attestation is an assertion made by an assessor, which links specific requirements to corresponding claims. These attestations serve as formal statements confirming that certain conditions or standards have been met in the analysis. Attestations provide the legal and technical backing for the claims presented in the DSCG.
- Claims: Claims represent statements about the analyzed components, such as their security status, compliance with regulations, or any identified vulnerabilities. Claims are evaluated by the assessors and are critical in determining whether an artifact meets the required security and compliance standards.
- Evidence: Evidence refers to the data, findings, or observations collected during the analysis that support the claims and attestations. It provides the factual basis for the claims made and serves as a reference for future validation and audits.

5.3.1 DSCG Declarations for FATex


```
"declarations": {
  "targets": {
    "components": [
      {
        "type": "firmware",
        "name": "A firmware binary which was tested by the ReSCALE
           Dynamic Code Analysis Module through FATex tool",
        "bom-ref": "pkg:hex/a-very-nice-binary@1.0.0",
        "version": "1.0.0",
        "description": "Such a good piece of firmware, but it has
           vulnerability issues ...",
        "purl": "pkg:hex/a-very-nice-binary@1.0.0",
        "hashes": [
          {
            "alg": "SHA-1",
            "content": "53ab2f0f92e87ea4874c8c6997335c211d81e636"
          }
        ]
      }
    ]
  },
  "assessors": [
    {
      "bom-ref": "Consumer Reference",
      "thirdParty": false,
      "organization": {
        "bom-ref": "Consumer Entity",
        "name": "Some Company that wants to generate a DSCG for a
           binary",
        "contact": [
              {
                      "email": "some@company.com"
              }
        ]
      }
```

```
      }
    ],
    "attestations": [
      {
        "assessor": "Consumer Reference",
        "summary": "Mapping of Requirements to Claims",
        "map": [
          {
            "requirement":
               "https://rescale-project.eu/standard/1.0.0/conformance/
            complete",
            "counterClaims": [
              "Claim: ReSCALE Test Suite XY found something!"
            ]
          }
        ]
      }
    ],
    "claims": [
      {
        "bom-ref": "Claim: ReSCALE Test Suite XY found something!",
        "target": "pkg:hex/a-firmware-version@1.0.0",
        "evidence": [
          "Evidence: browser screenshot",
          "Evidence: log info"
        ]
      }
    ],
    "evidence": [
{
        "bom-ref": "Evidence: browser screenshot",
        "description": "base64 representation of an raw image file,
           representing the screenshot. Accessing the firmware on
           browser shows whether it works fine",
        "data": [
          {
            "name": "ReSCALE Test Suite XY Output",
            "contents": {
              "attachment": {
                "content": "Test Suite XY found: \n\tproblem of file
                   firmware.bin at defualt tab"
              }
            }
          }
        ]
      },
      {
        "bom-ref": "Evidence: log info",
        "description": "Log files show whether firmware faces errors",
        "data": [
          {
            "name": "ReSCALE Test Suite XY Output",
            "contents": {
              "attachment": {
                "content": "Test Suite XY found: \n\tproblem of file
                   firmware.bin at line 23"
              }
            }
```

```
        }
      ]
    }
  ]
}
```

5.3.2 DSCG Declarations for inSpectre

```
"declarations": {
  "targets": {
    "components": [
      {
        "type": "operating-system",
        "name": "software version/revision (SBOM), A Very Nice linux
           kernel which was tested by the ReSCALE Dynamic Code
           Analysis Module through inspectre tool",
        "bom-ref": "pkg:hex/a-very-nice-library@1.0.0",
        "version": "1.0.0",
        "description": "Such a good piece of software, but it has
           security issues ...",
        "purl": "pkg:hex/a-very-nice-library@1.0.0",
        "hashes": [
          {
            "alg": "SHA-1",
            "content": "53ab2f0f92e87ea4874c8c6997335c211d81e636"
          }
        ]
      }
    ]
  },
  "assessors": [
    {
      "bom-ref": "Consumer Reference",
      "thirdParty": false,
      "organization": {
        "bom-ref": "Consumer Entity",
        "name": "Some Company that wants to generate a DSCG for a
           library",
        "contact": [
              {
                      "email": "some@company.com"
              }
        ]
      }
    }
  ],
  "attestations": [
    {
      "assessor": "Consumer Reference",
      "summary": "Mapping of Requirements to Claims",
      "map": [
        {
          "requirement":
             "https://rescale-project.eu/standard/1.0.0/conformance/
          complete",
          "counterClaims": [
```

```
            "Claim: ReSCALE Test Suite XY found something!"
          ]
        }
      ]
    }
  ],
  "claims": [
    {
      "bom-ref": "Claim: ReSCALE Test Suite XY found something!",
      "target": "pkg:hex/a-software-version@1.0.0",
      "evidence": [
        "Evidence: leakage rate"
      ]
    }
  ],
  "evidence": [
    {
      "bom-ref": "Evidence: leakage rate",
      "description": "Exploitable leakage rate found",
      "data": [
        {
          "name": "ReSCALE Test Suite XY Output",
          "contents": {
            "attachment": {
              "content": "Test Suite XY found: \n\tleakage rate of
                 file main.c at line 23"
            }
          }
        }
      ]
    },
    {
      "bom-ref": "Evidence: Shadow leak",
      "description": "Exploitable shadow leak found",
      "data": [
        {
          "name": "ReSCALE Test Suite XY Output",
          "contents": {
            "attachment": {
              "content": "Test Suite XY found: \n\tshadow leak of
                 file main.c at line 30"
            }
          }
        }
      ]
    },
    {
      "bom-ref": "Evidence: gadget in linux kernel",
      "description": "Exploitable gadget found",
      "data": [
        {
          "name": "ReSCALE Test Suite XY Output",
          "contents": {
            "attachment": {
              "content": "Test Suite XY found: \n\texploitable
                 gadget of file main.c at line 30"
            }
          }
```

```
          }
        ]
      }
    ]
  }
```

5.3.3 DSCG Declarations for RAISE

```
  "declarations": {
    "targets": {
      "components": [
        {
          "type": "application",
          "name": "A Very Nice REST API which was tested by the
             ReSCALE Dynamic Code Analysis Module through RESTler
             tool",
          "bom-ref": "pkg:hex/a-very-nice-library@1.0.0",
          "version": "1.0.0",
          "description": "Such a good piece of service6, but it has
             security issues ...",
          "purl": "pkg:hex/a-very-nice-library@1.0.0",
          "hashes": [
            {
              "alg": "SHA-1",
              "content": "53ab2f0f92e87ea4874c8c6997335c211d81e636"
            }
          ]
        }

      ]
    },
    "assessors": [
      {
        "bom-ref": "Consumer Reference",
        "thirdParty": false,
        "organization": {
          "bom-ref": "Consumer Entity",
          "name": "Some Company that wants to generate a DSCG for
             their library",
          "contact": [
                {
                        "email": "some@company.com"
                }
          ]
        }
      }
    ],
    "attestations": [
      {
        "assessor": "Consumer Reference",
        "summary": "Mapping of Requirements to Claims",
        "map": [
          {
            "requirement":
               "https://rescale-project.eu/standard/1.0.0/conformance/
            complete",
```

```
          "counterClaims": [
            "Claim: ReSCALE Test Suite XY found something!"
          ]
        }
      ]
    }
  ],
  "claims": [
    {
      "bom-ref": "Claim: ReSCALE Test Suite XY found something!",
      "target": "pkg:hex/a-software-version@1.0.0",
      "evidence": [
        "Evidence: Internal Server Error"
      ]
    }
  ],
  "evidence": [
    {
      "bom-ref": "Evidence: Internal Server Error",
      "description": "500 HTTP status code found",
      "data": [
        {
          "name": "ReSCALE Test Suite XY Output",
          "contents": {
            "attachment": {
              "content": "Test Suite XY found: \n\tHTTP status code
                 of request for API endpoints"
            }
          }
        }
      ]
    },
    {
      "bom-ref": "Evidence: Unprocessible Entity",
      "description": "422 unprocessible entity found",
      "data": [
        {
          "name": "ReSCALE Test Suite XY Output",
          "contents": {
            "attachment": {
              "content": "Test Suite XY found: \n\tunprocessible
                 entity of request for API endpoints"
            }
          }
        }
      ]
    }
  ]
}
```

5.3.4 DSCG Declarations for SCA Dynamic Hardware Analyzer

```
"declarations": {
  "targets": {
    "components": [
      {
```

```
        "type": "IP_Core",
        "name": "A Dynamic Hardware Analyzer that analyzes Side
           Channel Attack Traces collected from a software or
           hardware IP Cores in order to discover SCA leakage
           vulnerabilities",
        "bom-ref": "pkg:hex/a-very-nice-library@1.0.0",
        "version": "1.0.0",
        "description": "This IP Core has a series of SCA security
           issues ...",
        "purl": "pkg:hex/a-very-nice-library@1.0.0",
        "hashes": [
          {
            "alg": "SHA-1",
            "content": "53ab2f0f92e87ea4874c8c6997335c211d81e636"
          }
        ]
      }

    ]
  },
  "assessors": [
    {
      "bom-ref": "Consumer Reference",
      "thirdParty": false,
      "organization": {
        "bom-ref": "Consumer Entity",
        "name": "Some Company that wants to generate a DSCG for a
           library",
        "contact": [
              {
                      "email": "some@company.com"
              }
        ]
      }
    }
  ],
  "attestations": [
    {
      "assessor": "Consumer Reference",
      "summary": "Mapping of Requirements to Claims",
      "map": [
        {
          "requirement":
             "https://rescale-project.eu/standard/1.0.0/conformance/
          complete",
          "counterClaims": [
            "Claim: ReSCALE SCA Dynamic Hardware Analyzer found
               something!"
          ]
        }
      ]
    }
  ],
  "claims": [
    {
      "bom-ref": "Claim: ReSCALE SCA Dynamic Hardware Analyzer found
         something!",
      "target": "pkg:hex/a-software-version@1.0.0",
```

```
        "evidence": [
          "Evidence: Non Specific Leakage Assessment threshold
             violation"
        ]
      }
    ],
    "evidence": [
      {
        "bom-ref": "Evidence:Non Specific Leakage Assessment threshold
           violation",
        "description": "Performed non-specific Leakage Assessment and
           the leakage graph exceeds the TVLA thresholds",
        "data": [
          {
            "name": "ReSCALE SCA Dynamic Hardware Analyzer Output",
            "contents": {
              "attachment": {
                "content": "ReSCALE SCA Dynamic Hardware Analyzer
                   found: \n\tNon Specific Leakage Assessment
                   threshold violation on DuT IP Core"
              }
            }
          }
        ]
      },
      {
        "bom-ref": "Evidence:CPA assessment succesful",
        "description": "Performed CPA has succesfully recovered part
           of sensitive information",
        "data": [
          {
            "name": "ReSCALE SCA Dynamic Hardware Analyzer Output",
            "contents": {
              "attachment": {
                "content": "ReSCALE SCA Dynamic Hardware Analyzer
                   found: \n\tunprocessible software or hardware IP
                   Core execution flow against CPAs"
              }
            }
          }
        ]
      }
    ]
  }
```


5.4 Unification of Test Reports

In dynamic security testing, various tools can be used to assess the behavior and vulnerabilities of both software and hardware components. These tools generate results in different formats, each with its own strengths and limitations. To streamline analysis and reporting, the RESCALE project requires a unified format that consolidates these diverse outputs into a common structure. This unification ensures consistency and facilitates integration with other components.

The need for a unified reporting format arises from the varying data structures produced by the various dynamic testing tools. These include fuzzing results, mutation analysis and runtime security findings, which are critical for understanding system vulnerabilities under real-world conditions. These results must be aggregated in a way that is both comprehensive and adaptable for integration into higher-level frameworks, such as the DSCG and TBOM.

To achieve this, RESCALE evaluates three existing reporting formats, Open Test Reporting (OTR), JUnit XML and Test Anything Protocol (TAP). These formats are assessed based on their ability to capture the necessary information from dynamic tests and provide a foundation for a unified reporting structure that meets the needs of the project.

5.4.1 Open Test Reporting (OTR)

Open Test Reporting (OTR) [19] is a flexible reporting format designed to facilitate the exchange and integration of test results across diverse tools and environments. Unlike more rigid formats, OTR allows for a greater degree of customization, making it well-suited for dynamic analysis scenarios where test outcomes may vary in both structure and detail.

OTR can be used as a potential format for unifying dynamic test results. Its flexibility makes it capable of handling various types of testing data, such as fuzzing results and other dynamic security findings. However, like other test formats, it requires some level of adaptation to fully accommodate the specific needs of dynamic security analysis, such as reporting vulnerabilities, their severity, and runtime-specific data (e.g., system states or configurations at the time of testing).

While OTR is highly adaptable, its use within RESCALE may require extensions to incorporate additional metadata specific to dynamic security testing, such as the exploitability of vulnerabilities or detailed runtime information. These extensions would enable OTR to meet the project's requirements for continuous validation and real-time risk assessment, allowing for better integration with other project components like the DSCG and TBOM.

5.4.2 Test Anything Protocol (TAP)

The Test Anything Protocol (TAP) [20] is an open format designed to unify output across multiple testing tools, allowing for the seamless integration of test results into various workflows. Originally developed for Perl [8], TAP has expanded beyond its initial scope and is now widely adopted across programming languages and testing frameworks. TAP addresses challenges associated with diverse, often incompatible multiple testing outputs that can complicate result aggregation and interpretation.

TAP employs a simple, line-based format that is both human-readable and easily processed by machines. This simplicity enhances interoperability between tools and platforms, enabling efficient aggregation and analysis of results from multiple sources. TAP's standard format simplifies the tracking, reporting, and processing of test results, improving consistency and efficiency in handling test outputs across diverse testing environments.

A major benefit of TAP is its capability to encapsulate basic information about test outcomes, including test success or failure, specific details regarding the test cases executed, and any ac-

companying diagnostic data. TAP's structure is well-suited for scenarios such as automated bug filing, integration into CI/CD workflows, and tracking test trends. By unifying the representation of testing results, TAP supports the development of universal tools and platforms that can process output from various analyzers, promoting a more integrated and effective development and testing environment.

5.4.3 JUnit XML

JUnit XML [17] is a widely adopted reporting format, primarily used in unit testing frameworks to capture the results of individual test cases and test suites. It organizes results in a structured, hierarchical format, detailing test outcomes such as pass/fail status, error messages and stack traces. This structure is well-suited for reporting simple pass/fail results and can be extended to include additional metadata.

JUnit XML can provide a convenient format for integrating dynamic analysis results. Although it was initially designed for functional testing, its structured format and widespread adoption in continuous integration (CI) environments make it a suitable choice for capturing dynamic testing results, including fuzzing outcomes, runtime behavior analysis, etc.

However, while the format can store basic test outcomes, it would need to be adapted to include additional metadata, such as vulnerability severity, etc. These extensions would enable JUnit XML to report dynamic security findings more effectively.

5.4.4 Choosing the Reporting Standard

OTR supports XML output exclusively, meaning that while you can feed it data in various formats like JSON or CSV, the resulting report will always be generated in XML. This is similar to JUnit XML. While it's theoretically possible to extend OTR to produce results in JSON, the feasibility, required effort and time commitment are uncertain. Another option would be to develop a custom standard inspired by SARIF; however, this would also be a resource-intensive process. After further research, it appears there is no widely accepted standard that outputs final reports in JSON specifically for dynamic analysis. Even academic papers have not identified such a format. Existing formats support only a limited range of tools and are insufficient for our diverse requirements since each tool operates differently.

We are exploring suitable formats to meet our requirements, with potential options including OTR, TAP and JUnit XML. Our aim is to either adopt one of these existing standards or develop a tailored solution. Initial evaluations indicate that one format may be more promising, but it remains under review and is outside the scope of the current project. While the project does not involve delivering a final product, adopting a standardized format would improve credibility and ensure alignment with best practices.

6 Main Innovations & Conclusion

This section will wrap up the document by summarizing the key contributions of this deliverable and outlining future work related to dynamic software testing analysis and DSCG generation that will be undertaken as the RESCALE project advances.

6.1 Main Innovations

The main innovation of Deliverable D3.3 lies in its advanced approach to dynamic testing for security within the supply chain, integrating ML and DL to create more intelligent and targeted test cases. Here are a few key points that highlight the innovation:

1. **Enhanced Input Selection via ML/DL**: D3.3 leverages ML/DL techniques to generate optimized test vectors, moving beyond random input generation to create targeted test cases that improve the coverage and effectiveness of dynamic testing. This approach helps in detecting vulnerabilities in a more efficient and precise manner.
2. **Integration of Dynamic Testing with Fuzzing and Mutation Analysis**: By combining SotA fuzzers with mutation analysis, D3.3 enables automated creation and continuous refinement of test cases based on evolving vulnerabilities. This method supports a more adaptive testing process tailored to different component types (software and hardware).
3. **Comprehensive Security for Diverse Components**: The module is designed to address a range of components across the RESCALE supply chain, covering both hardware and software vulnerabilities. This scope, along with the focus on runtime analysis, provides a robust security framework adaptable to various supply chain elements.
4. **Generation of the Dynamic Supply Chain Component Guarantee (DSCG)**: D3.3 introduces the DSCG, a trusted report on dynamic analysis for each component. The DSCG not only certifies security at the component level but is structured to be integrated into the TBOM, ensuring traceability and trust throughout the lifecycle.
5. **Tailored Cybersecurity Testing for Supply Chain**: With specific adaptations for cybersecurity testing needs, such as automated fuzzing rule generation from REST API schemas and mutation-based vulnerability analysis, D3.3 offers a specialized testing approach that addresses unique supply chain security challenges.

In sum, the main innovation of D3.3 is its multi-faceted dynamic testing framework, which uses AI-enhanced testing techniques to provide comprehensive, component-specific security insights, supporting the establishment of a trusted supply chain.

6.2 Conclusion

Deliverable D3.3 details the initial implementation of the Dynamic Testing Analysis Module, a cornerstone of RESCALE's objective to enhance supply chain security. This module introduces a dynamic, multi-layered testing framework, crucial for identifying vulnerabilities across

software and hardware components. By leveraging advanced dynamic testing methodologies, including ML/DL based fuzzing and symbolic execution, the module significantly improves the depth and breadth of vulnerability detection.

Key to this deliverable is the Dynamic Supply Chain Component Guarantee (DSCG). The DSCG provides a real-time, comprehensive risk assessment by aggregating outputs from various testing components. This dynamic assurance mechanism not only enhances security but also ensures ongoing trustworthiness throughout the supply chain. Its integration within the Trusted Bill of Materials (TBOM) further strengthens its role in maintaining supply chain integrity.

The modular and scalable architecture of the Dynamic Testing Module facilitates seamless integration with RESCALE's micro-services environment. This flexibility ensures that the module can adapt to diverse use cases and evolving security needs. The module's capability to detect a wide array of vulnerabilities, including runtime and logic errors, underscores its comprehensive coverage. As the project progresses, the insights and advancements outlined in D3.3 will inform future iterations, culminating in a more robust and effective security framework in subsequent deliverables.

Ultimately, the innovations presented in D3.3 contribute significantly to RESCALE's goals by advancing state-of-the-art (SotA) dynamic testing and reinforcing the security of critical supply chains.

6.3 Future Work

Building on the advancements outlined in Deliverable D3.3, the next stages of development for the Dynamic Testing Analysis Module will focus on enhancing both technical capabilities and operational integration within the RESCALE framework.

- **Refinement and Expansion of Testing Techniques**: The dynamic testing methodologies will be further refined to improve accuracy and efficiency. Future iterations will integrate additional testing approaches to handle increasingly complex and diverse supply chain components, ensuring comprehensive vulnerability detection.
- **Enhanced DSCG Generation**: The DSCG will be optimized to deliver more comprehensive assessments of security risks, ensuring its continued effectiveness. Additionally, the results from the dynamic module require additional fields to provide detailed information about both the software and the dynamic module. For instance, since the dynamic testing module also runs the hardware analyzer, the output should specify how the component or service was manufactured or deployed. This can be defined using the workflow field (which includes a list of workflows that can be declared to achieve specific orchestrated goals and can be independently triggered) under the formulation section. These specifications will be incorporated into the CDX at a later stage.
- **Scalability and Performance Optimization**: A key focus will be on improving the scalability of the Dynamic Testing Module. Efforts will concentrate on enhancing performance to ensure efficient resource utilization across diverse environments.

- **Continuous Validation and Testing**: Ongoing validation efforts will focus on ensuring the robustness and reliability of the Dynamic Testing Module in operational environments. This includes regular assessments to confirm the effectiveness of existing tools against evolving threats, ensuring their continued alignment with RESCALE's security objectives.

These steps will consolidate the Dynamic Testing Module's as a cornerstone of RESCALE's role to establish a trusted and resilient supply chain ecosystem.

References

[1] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. Evomaster: A search-based system test generation tool. *Journal of Open Source Software*, 6(57):2153, 2021.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.

[3] A. Costin and A. Francillon. Karonte: Detecting insecure code in iot firmware. *USENIX Security*, 2020.

[4] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages, 2020.

[5] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 168–181. ACM, 2003.

[6] Pavan Kumar Joshi. Dynamic application security testing for payment applications: A comprehensive guide. *International Journal of Science and Research (IJSR)*, pages 1567–1573, 2016.

[7] Afef Jmal Maâlej, Moez Krichen, and Mohamed Jmaïel. A comparative evaluation of state-of-the-art load and stress testing approaches. *International Journal of Computer Applications in Technology*, 51(4):283–293, 2015.

[8] Michael G Schwern, Andy Lester . Documentation for tap, 2006. Accessed: 2024-11-15.

[9] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.

[10] Vienna University of Technology. Anubis: Dynamic malware analysis. https://anubis.iseclab.org/, 2023.

[11] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[12] Joe Security. Joebox: Dynamic malware analyzer. https://www.joesecurity.org/joebox/, 2023.

[13] Aman Sharma, Martin Wittlinger, Benoit Baudry, and Martin Monperrus. Sbom.exe: Countering dynamic code injection based on software bill of materials in java, 2024.

[14] Angr Team. Angr: Binary analysis framework. https://angr.io/, 2023.

[15] FirmFuzz Team. Firmfuzz: Firmware fuzzing framework. https://firmfuzz.com/, 2023.

[16] IPEA Team. Ipea: In-situ program execution analyzer. https://ipea.io/, 2023.

[17] JUnit Team. Junit xml format. https://github.com/windyroad/JUnit-Schema, 2023.

[18] Labrador Team. Labrador: Fuzzing for iot devices. https://iot-labrador.org/, 2023.

[19] OTa4J Team. Open test reporting (ota4j). https://github.com/ota4j-team/open-test-reporting, 2023.

[20] Testanything.org. Test anything protocol. Accessed: 2024-11-15.

[21] Yijun Wang. Vulnerability analysis and improvement of rasp technology. In *2022 International Symposium on Advances in Informatics, Electronics and Education (ISAIEE)*, pages 266–272. IEEE, 2022.

[22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s):230:1–230:36, 2022.