



Fraunhofer

Intelligent Application Security Testing

Contents

1. Management Summary	4
2. Introduction	6
3. Background	7
4. The IntelliSecTest Solution	9
4.1 Features and Expert Tools	9
4.2 Workflow Support	10
4.3 Integration	14
4.4 Technical Architecture and System Requirements	14
5. Case Studies	17
5.1 Testing Proprietary Code: mJS	17
5.2 Testing 3 rd Party Code: libTiff	19
5.3 Patch Validation: mJS	20
6. Summary	22
Imprint	23

1. Management Summary

Need for proactive security measures for manufacturers of digital products to meet regulatory requirements and address evolving threats, particularly in light of the Cyber Resilience Act (CRA).

Manufacturers of products that contain software and can have a data connection to another device or network need to implement proactive security measures to ensure compliance with regulatory requirements and the changing threat landscape. The necessity for more efficient and effective security testing methods and tools is underscored by recent security incidents and by regulatory acts such as the Cybersecurity Act (CSA) and the Cyber Resilience Act (CRA), which are driven by the European Union. In accordance with the CRA, manufacturers of products with digital elements intended for sale on the European market – which, in essence, encompasses all digital products – are subject to new obligations including, among others, to

- identify vulnerabilities through effective and regular tests and
- address and remediate them without delay.

This encompasses not only proprietary code created by manufacturers but also (open-source) software they integrate. While these new responsibilities may be seen as an additional burden for manufacturers, they offer a significant opportunity to enhance supply chain security. To fully leverage this opportunity, manufacturers must have effective tools offering a high degree of automation and skilled employees to operate them and process their results.

IntelliSecTest is an innovative solution that helps manufacturers to address the challenges posed by both regulations and the threat landscape. IntelliSecTest provides effective assistance in applying suitable testing methods for C programs to ensure compliance with regulatory requirements, such as the CRA.

The IntelliSecTest solution provides support to organizations and their development and testing teams in their workflows:

- security testing 3rd party code, e.g., open source software (OSS)
- security testing proprietary code, developed by the manufacturer
- development of effective security patches for identified vulnerabilities.

By automating different steps of workflows with a highly automated, configurable and extensible set of deeply integrated expert tools, IntelliSecTest relieves employees from complex tasks, thus increasing their efficiency and reducing costs. Without IntelliSecTest, these tasks often require expert knowledge, further straining this limited, valuable resource. The IntelliSecTest solution enables testers to perform these tasks efficiently and effectively from the outset, eliminating the need for lengthy training periods. Moreover, development teams get precise information about the location of vulnerabilities in C code and receive assistance in the debugging process through test cases, which significantly reduces the time required to comprehend and resolve identified issues. The IntelliSecTest solution presents all the results directly in the working environment, through integration with major integrated development environments (IDEs), such as Visual Studio Code (VS Code), using standardized interfaces.



IT software failure paralyzed airports in July 2024

Key Provisions of the Cyber Resilience Act (CRA) Pertaining to Security Testing

The CRA, enacted by the European Parliament in March 2024, introduces several new obligations for manufacturers regarding the management of vulnerabilities in products with digital elements. These obligations significantly impact security testing activities:

1. Manufacturers of products with digital elements must identify and document vulnerabilities.
2. Manufacturers are required to address and remediate vulnerabilities promptly, including the provision of security updates.
3. Manufacturers must conduct effective and regular tests and reviews of the security of products with digital elements.

To meet these new requirements, it is essential that manufacturers enhance their security testing processes and activities. Although the CRA will not take effect until 2027, it is crucial for manufacturers to begin adapting their processes and upgrading their tool landscape immediately to ensure compliance when the regulation is enforced.

2. Introduction

Security represents a dynamic challenge, influenced not only by a continuously evolving threat landscape but also by changing regulatory requirements.

The recent sophisticated supply chain attack attempts on Linux via xz utils, along with the introduction of the CSA and the CRA, underscore the necessity for more effective security testing methodologies and tools. These tools should streamline processes and deliver reliable, qualified, and quantified results that meet the demands of emerging regulations.

While previous regulations, such as NIS, have focused on critical infrastructure, more recent and upcoming regulations also apply beyond that sector and target specific technologies, such as connected digital products in the case of the CRA. The CRA imposes new responsibilities on manufacturers, requiring a focused on *effective and regular tests and addressing and remediating vulnerabilities without delay*. These requirements extend beyond proprietary code to include integrated open source and 3rd party software, thereby broadening the scope of manufacturers' security-related activities and associated costs. Given that security is often seen as a cost rather than a competitive advantage, there is an increasing need among manufacturers to improve the efficiency of their security measures.

Moreover, manufacturers are now required to demonstrate their compliance efforts, particularly in the event of a security breach, to mitigate liability for resulting damages and potential violations of privacy rights. This requirement not only reinforces the importance of robust security practices and also highlights the critical need for transparency and accountability in the face of regulatory scrutiny.

The IntelliSecTest solution effectively addresses regulatory requirements by providing comprehensive coverage of several key areas. It significantly reduces the manual effort involved in security testing through advanced automation, allowing for the swift and efficient identification of security vulnerabilities.

IntelliSecTest provides reliable and quantified results, enabling manufacturers to make informed decisions and take targeted actions to meet the necessary security standards. To meet the demands of new regulations such as the CSA and the CRA, IntelliSecTest ensures comprehensive compliance by assisting manufacturers in the identification, management, and documentation of vulnerabilities. This encompasses not only proprietary code but also integrated open-source and 3rd party software, ensuring a thorough approach to security in line with evolving regulatory expectations. Furthermore, it enables manufacturers to provide evidence of their compliance activities.

Additionally, in the event of uncovered vulnerabilities, IntelliSecTest facilitates the validation of developed security patches. By rigorously testing the patches, IntelliSecTest ensures that they effectively eliminate the identified security issues without introducing new vulnerabilities, thereby maintaining the integrity and security of the software. Therefore, IntelliSecTest not only meets the immediate need for effective security testing but also ensures long-term compliance with evolving regulations, making it an indispensable tool for manufacturers navigating the complex landscape of cybersecurity requirements.

3. Background

Static and dynamic analysis tools are currently utilized in the domain of cybersecurity quality assurance. While both categories serve distinct purposes, they possess complementary strengths and weaknesses that can limit their overall effectiveness and efficiency when integrated in a naive way.






Static program analysis is used for determining interesting properties of a given software program, such as the absence of security vulnerabilities, without executing the program. Static analysis for finding security vulnerabilities is commonly referred to as “Static Application Security Testing” (SAST).

One can distinguish different classes of static analysis approaches, ranging from formal methods like model checking, over techniques of abstract interpretation, to data-flow analysis. While static analysis can achieve full path coverage, it might report false positives, i.e., warnings about vulnerabilities that do not actually occur when running the program. Therefore, static analysis cannot prove the presence of errors (due to false positives), but only their absence. Discriminating the true positives, actual vulnerabilities that need to be fixed, from these false positives is a tedious effort that is usually done manually, which makes static analysis less efficient.

In contrast, dynamic analysis in security testing involves evaluating a software system or component during its execution. This method is used to detect security vulnerabilities that may not be evident through static analysis. Dynamic analysis can identify issues such as runtime errors, memory leaks, and other security threats that emerge during the application’s execution.

The most prevalent technique applied in security testing is fuzzing, categorized under “Dynamic Application Security Testing” (DAST). Fuzzing, or fuzz testing, is a dynamic software testing technique that involves automatically generating and inputting (semi-)random, malformed, or unexpected data into a software system to uncover coding errors and security-critical bugs. The primary objective of fuzzing is to trigger crashes, memory leaks, or unhandled exceptions that indicate potential vulnerabilities.

Figure 1: Strengths and weaknesses of static analysis and fuzzing

	Static Analysis	Fuzzing
 Approaches		
 Advantages	<ul style="list-style-type: none"> • High path coverage • Good presentation of results 	<ul style="list-style-type: none"> • Very few false warnings
 Drawbacks	<ul style="list-style-type: none"> • Requires approximation • High number of false warnings (false positives) 	<ul style="list-style-type: none"> • Random path coverage • Poor results presentation

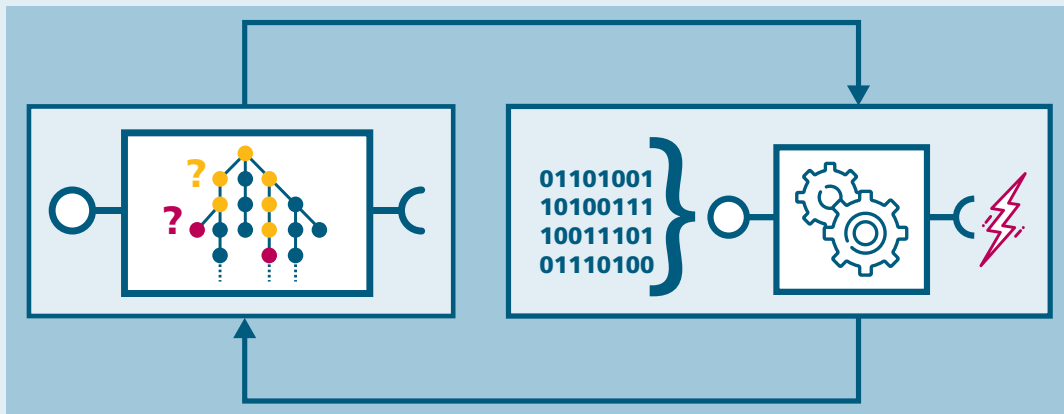


Figure 2: Hybrid analysis, also referred to as IAST, integrates SAST and DAST

However, fuzzing is constrained by limited path coverage, necessitating a substantial amount of time to adequately cover an entire program. This requirement can impede short development cycles. Fuzzing typically produces a large number of crashing inputs that have the same underlying bug as their root cause. Analyzing such duplicate crashing inputs occupies valuable resources.

Furthermore, this large number of duplicates makes it difficult to get an overall picture of the security posture of a program and increases the effort in processing these results, even more since they are often presented in a way that makes it hard to grasp them. Even further, specific test drivers require specific knowledge of the employed fuzzing tool when testing libraries.

Since using static analysis or dynamic analysis tools only is insufficient, IntelliSecTest implements a hybrid analysis – also referred to as “Interactive Application Security Testing” (IAST) – that combines and integrates SAST and DAST (see Figure 2). This approach is unique on the market, even though there are number of market players that provide static analysis tools, fuzzing tools or even both.

The market analysis categorizes tool providers based on their offerings, indicating that they typically supply either static analysis tools (represented by dots in the yellow area) or fuzzing tools (represented by dots in the blue area). Some providers have expanded their portfolios to include tools from the other category, potentially positioning them to offer hybrid analysis tools in the future (green dotted circles). However, as of the writing of this document, no provider offers a hybrid analysis tool.

Notably, one provider is known to be developing such a tool, although it is designed for Java applications rather than C applications.

4. The IntelliSecTest Solution

Advanced application security testing tool designed to help manufacturers identify and remediate security vulnerabilities in both proprietary and 3rd party code.

The CRA imposes new vulnerability handling requirements on manufacturers. These requirements encompass effective and regular testing, as well as the identification and remediation of vulnerabilities, extending beyond proprietary code to include OSS. This impacts the processes of development and testing teams.

The IntelliSecTest solution supports the development and testing teams in their efforts to meet their new responsibilities in three workflows:

- Security testing 3rd party code, e.g., OSS
- Security testing proprietary code, developed by the manufacturer
- Development of effective security patches for identified vulnerabilities.

4.1 Features and Expert Tools

The features offered by the IntelliSecTest solution are directly or indirectly supporting these workflows.

- **Detection of memory corruption vulnerabilities in C applications.** IntelliSecTest is able to identify critical vulnerabilities in C code, i.e., buffer overflow, double free, and use after free, of which buffer overflows and use after free still belong to the most critical vulnerabilities, according to CWE Top Ten 2023¹ and CWE Top 25 Most Dangerous Software Weaknesses 2023².
- **Verified vulnerabilities, including proofs.** All findings IntelliSecTest reports are confirmed vulnerabilities, false positives, as known from other tools, are excluded. A test case for analysis and debugging purposes accompanies each confirmed vulnerability.

- **Proof or estimation on the absence of vulnerabilities**
In the course of the verification, IntelliSecTest discriminates true vulnerabilities from false positives. However, it may happen that a vulnerability candidate from the static analysis cannot be proven to be a false positive. In that case, IntelliSecTest can report a statistical estimation that the vulnerability candidate is a false positive, which might be useful when justifying the testing efforts against third parties, e.g., authorities.
- **Fuzz driver generation for libraries.** Using its Fuzz Driver Generation technology, IntelliSecTest completely automates the task of writing fuzz drivers necessary to conduct library fuzzing.
- **Deduplication of fuzzing results.** With its advanced deduplication techniques, IntelliSecTest reliably identifies duplicates and thus, presents only consolidated test reports to the development and test teams.
- **Validation of security patches.** Through its advanced test generation techniques, IntelliSecTest provides a comprehensive evaluation of security patches, supporting developers through test cases that show the extent to which vulnerabilities can be triggered and potential weak points in a given security patch.
- **Integration with major interactive development environment (IDE).** IntelliSecTest seamlessly integrates with developers' most common user interface, the IDE.
- **Easily scalable, configurable, and extensible.** Through its container infrastructure using technologies like Kubernetes and Docker®, IntelliSecTest scales to nearly any infrastructure. Furthermore, it can be easily extended with new tools. Through its simple configuration, the integration of different expert tools is done in a few moments.

The IntelliSecTest solution automates the working steps of each of these workflows through a set of deeply integrated expert tools:

- **Fuzz Driver Generation** automatically generates fuzz drivers for library fuzzing.
- **Library Fuzzing** employs automatically generated fuzz drivers from Fuzz Driver Generation to perform library fuzzing, using AFL++.

¹ https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html

² https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

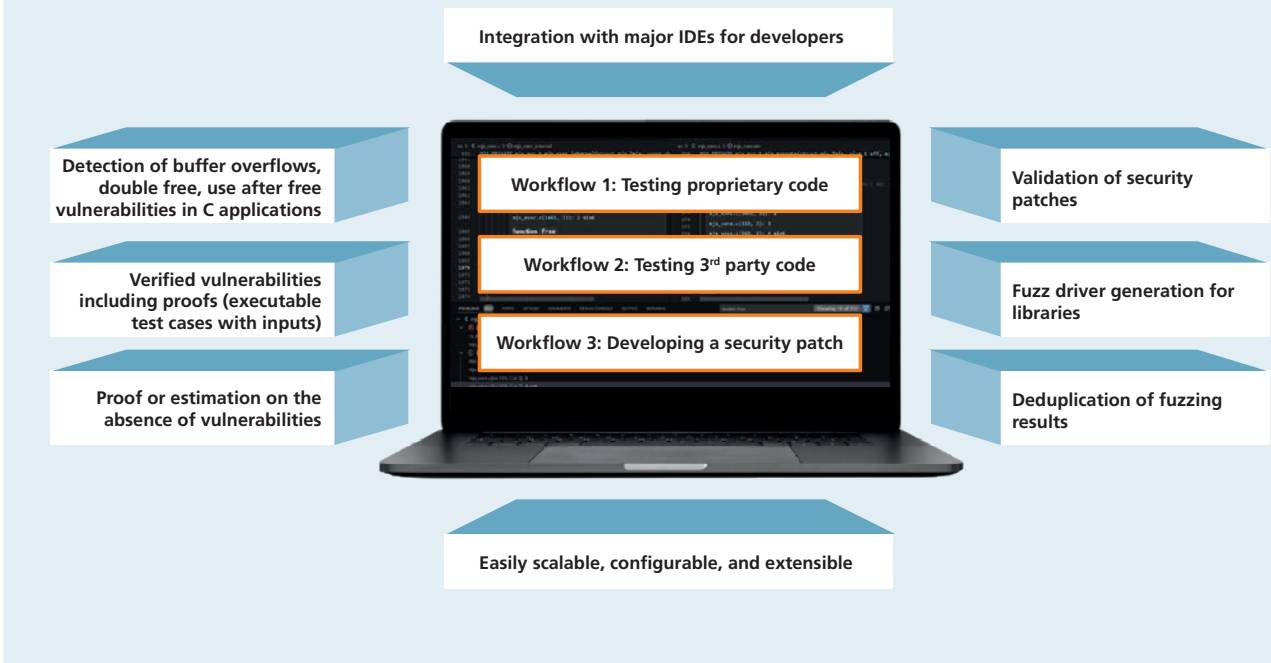


Figure 3: Overview of IntelliSecTest features

- **Static Analysis** performs data flow analyses on the source code to identify potential vulnerabilities.
- **Verification of SA Findings** employs advanced constraint solving techniques to discriminate true positives and false positives from the static analysis. It generates executable test cases as proofs for the true positives.
- **Directed Fuzzing** investigates code regions through directed fuzzing for vulnerabilities, using AFLGo.
- **Crash Deduplication** identifies duplicate crashes through stack trace analysis to report only unique bugs.
- **Patch Validation** generates a diverse test suite to trigger a vulnerability to evaluate the efficacy of a security patch.
- **IDE Reporting** presents the test results after deduplication in the IDE, within the source code and in the corresponding widgets.

4.2 Workflow Support

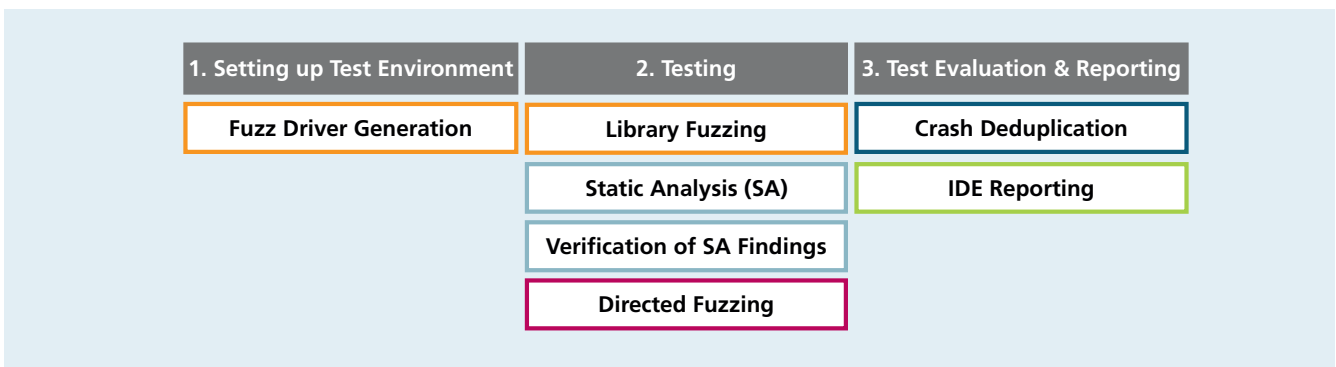
In the following, we present three workflows which would largely benefit from innovations of the IntelliSecTest solution.

These are:

- Testing proprietary code
- Testing 3rd party code, e.g., OSS
- Developing a security patch for a given vulnerability

In the following, we will present the workflows from a high level perspective and discuss how the capabilities of the IntelliSecTest solutions support them in detail. For each workflow, the IntelliSecTest solution builds upon established security testing tools, such as AFL and AddressSanitizer (ASan), and builds on the one hand a wrapper and on the other hands integrates them into a turnkey solution that developers without expert knowledge in security testing and experience in using respective tools can use out of the box.

Figure 4: Workflow and supporting IntelliSecTest expert tools for testing proprietary and 3rd party code



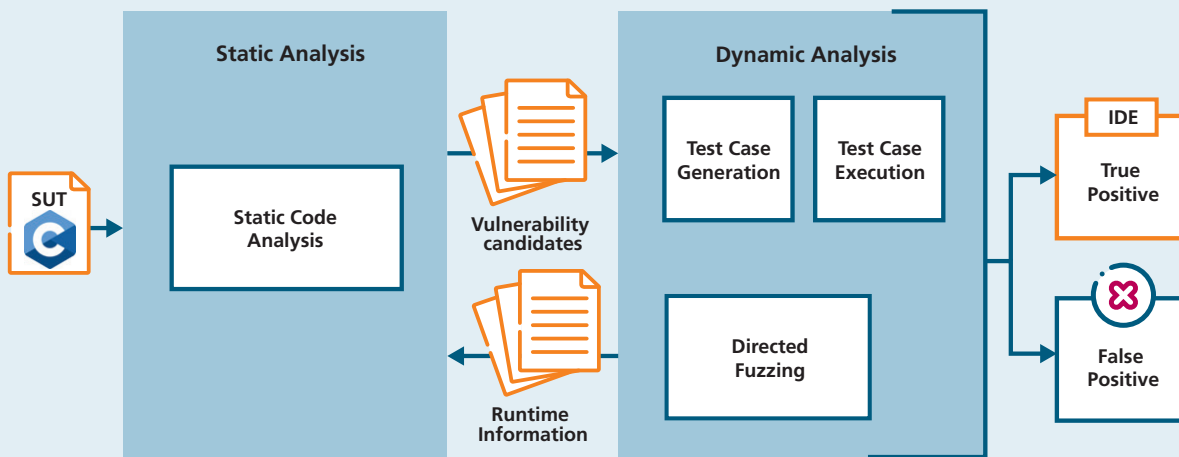


Figure 5: Overview of the vulnerability detection process for proprietary code

Testing Proprietary Code A common use case is to test proprietary code, i.e., code developed by the manufacturer, for vulnerabilities. It consists of three steps, as shown in Figure 4. First, the developers or tester must set up a test environment that includes testing tools and the system under test (SUT). This step is usually done manually, and requires a deep understanding of the fuzzing tool. This is where IntelliSecTest steps in and completely automates this step, relieving developers and testers of this task, through its expert Fuzz Driver Generation. As a result, the subsequent Library Fuzzing can be started automatically. The second step of this workflow – testing – is done by the static analysis and the verification of its results. IntelliSecTest’s Static Analysis (SA) expert tool automatically identifies vulnerability candidates in the source code and propagates this information to the Verification of SA Findings expert tool, which processes this information and discriminates between true and false positives using advanced constraint solving techniques. If this is not possible, Directed Fuzzing dynamically analyzes the source code, which is suspected of containing the vulnerability.

As shown in Figure 5, IntelliSecTest performs static analysis, such as deep data-flow analyses, to compute potential vulnerability candidates and to extract static information from the SUT regarding them. In addition, it runs directed fuzzing in the background to incrementally enhance the static analysis results through dynamic analysis. To avoid reporting false warnings, the IntelliSecTest solution verifies all vulnerability candidates by executing small, fine-grained test cases directed towards the vulnerability in question that are automatically generated. This way, IntelliSecTest filters out vulnerability candidates that are false warnings or cannot be exploited.

The test results are then subjected to Crash Deduplication, so that any vulnerability identified is only reported only once and directly presented within the developer’s working environment via IDE Reporting. A test case is attached to each reported vulnerability, completing the Test Evaluation & Reporting phase. This allows developers to understand, analyze, and debug a vulnerability.

Testing 3rd Party Code As software products become increasingly complex, they are composed of many components of varying origins, e.g., proprietary code, OSS, and contractor-supplied artifacts. To gauge the security of such software, it is crucial to test 3rd party code to eliminate supply chain vulnerabilities and to meet regulatory requirements, such as those from the CRA. The workflow of testing 3rd party code applies, for example, when a developer wants to integrate an OSS for a new functionality of the software product or if a developer is tasked to evaluate the security of already integrated components.

Building on the previous discussion of testing proprietary code for vulnerabilities, the workflow for testing 3rd party code follows a similar three-step process, as shown in Figure 4. First, the test environment must be set up, which includes configuring the necessary test tools and the SUT. Additionally, test adapters are required to connect the test tool to the SUT. This requires a lot of manual effort and requires a lot of time, since in most cases the tester is not familiar with the code to be tested. In contrast to the manual effort typically required in this phase, IntelliSecTest utilizes its Fuzz Driver Generation tool to automate the setup, significantly simplifying the process for developers and testers.

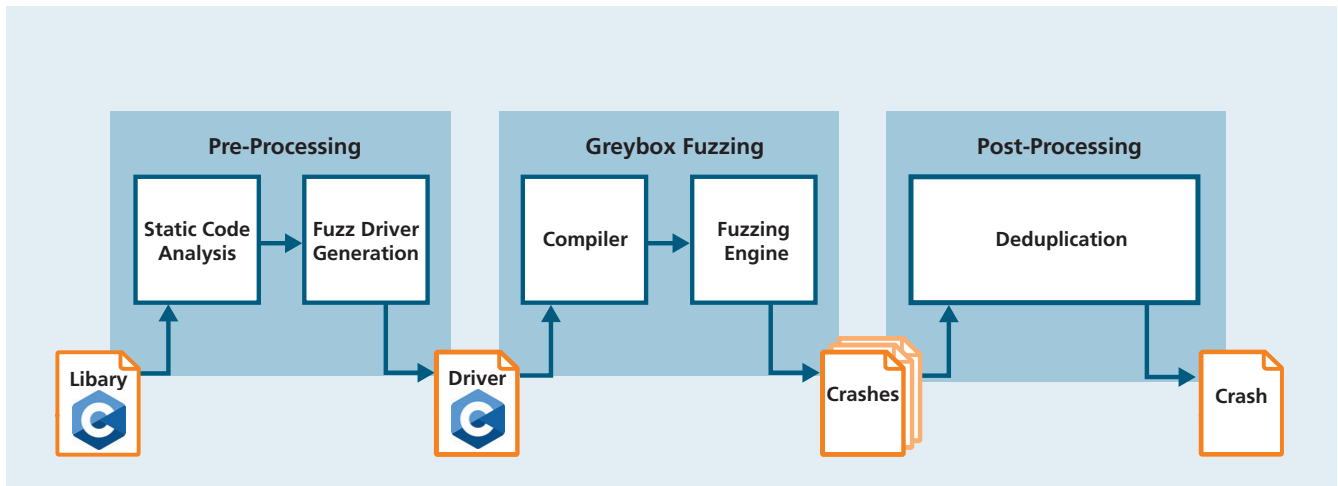


Figure 6: Overview of the 3rd party library testing process

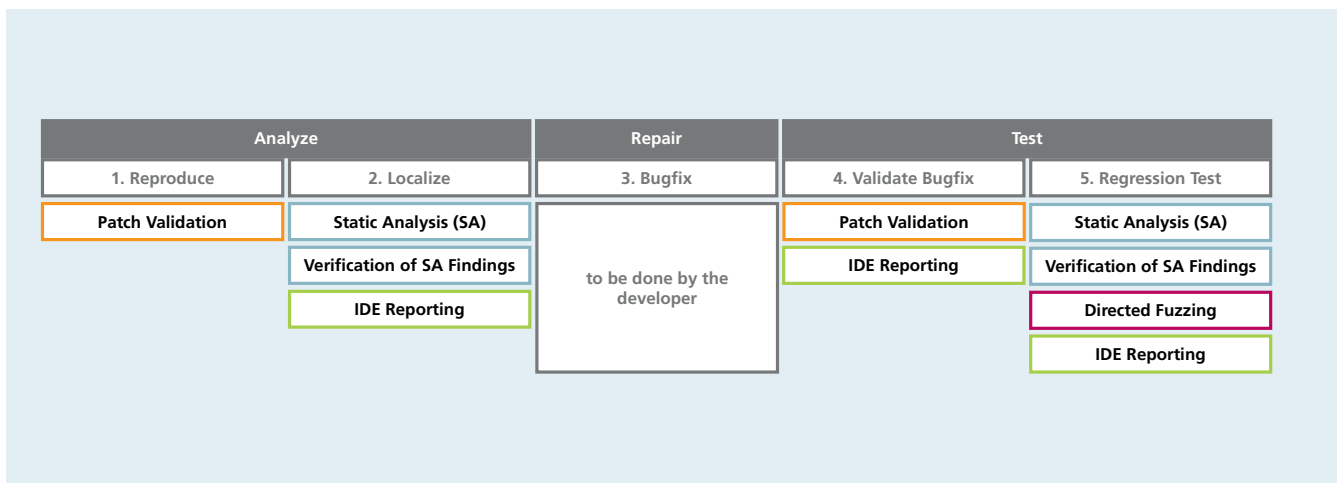
Figure 6 displays how IntelliSecTest supports and automates this workflow. First, IntelliSecTest analyzes the target library for possibly vulnerable entry points and passes them to the Fuzz Driver Generation. The Fuzz Driver Generation combines this information with the library and generates a fuzz driver, a dedicated program that exercises the library and, thus, makes it fuzzable. The next step is to compile the fuzz driver and use it with a modern greybox fuzzer, AFL++. During fuzzing, every detected crash is passed to the Crash Deduplication for post-processing. Finally, IntelliSecTest reports the deduplicated crashes back to the user. Static Analysis and the verification of its results also identify further vulnerabilities, similar to the first workflow.

While this workflow is technically identical to the first, it has one major difference for the manufacturer: the developer is not familiar with the code. This can be a major obstacle to effective and efficient security testing, and can pose a

significant risk to manufacturers as they deal with the security of integrated 3rd party products. This risk stems from the fact that open source projects are not accountable for vulnerabilities in their software, that it is often unknown whether and to what extent the security of the OSS has been verified, and whether this process conforms to standards and best practices. In fact, even popular, widely used OSS is regularly affected by critical vulnerabilities, such as Heartbleed, ShellShock, and Log4Shell.

Patch Validation The Patch Validation addresses the CRA’s requirement that manufacturers must provide patches for identified vulnerabilities in a timely manner. IntelliSecTest facilitates this process to enable developers to test code that is intended to address a vulnerability identified by the developer or reported by security researchers.

Figure 7: Workflow and supporting IntelliSecTest expert tools for patch validation



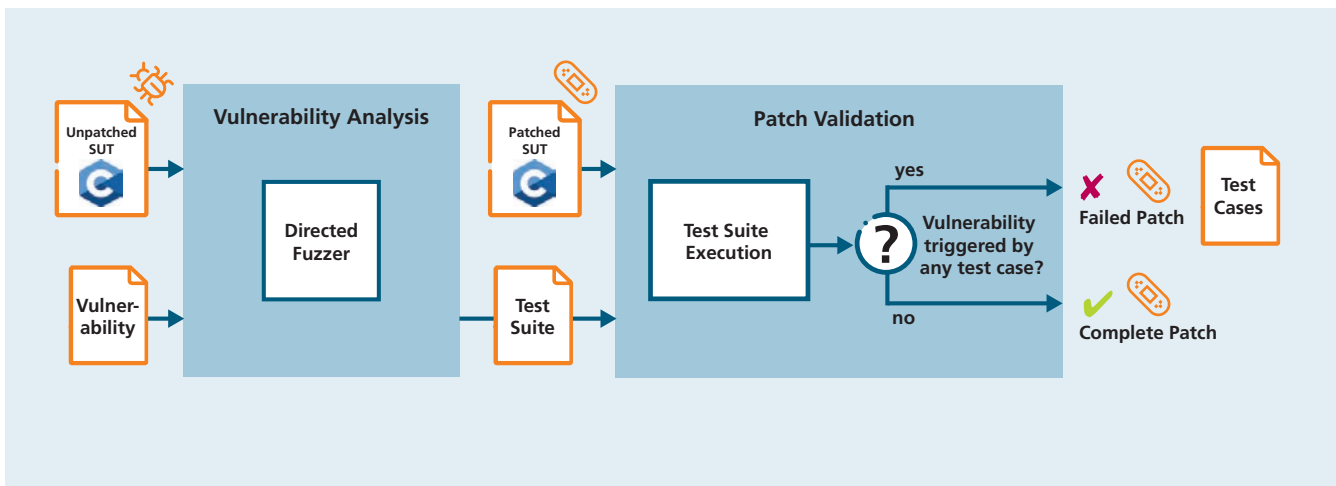


Figure 8: Overview of security patch validation process

Figure 7 illustrates a comprehensive patch validation workflow supported by IntelliSecTest, structured into three main stages: Analyze, Repair, and Test. Each stage of the workflow has specific tasks and utilizes specialized tools to ensure comprehensive patch validation and efficient resolution of security vulnerabilities. The initial step of the analysis stage is to reproduce the reported vulnerability to confirm its existence and understand its impact. This step is undertaken to accurately reproduce the conditions under which the vulnerability occurs. As with the previous workflow, various expert tools can be employed to assist with the localization of the vulnerability. In the Repair stage, the developers implement a security patch for the identified vulnerability. The results from the preceding stage provide essential input to the patch development process. This includes comprehensive data about the vulnerability, such as its location in the code and the test cases generated in the Reproduce stage that trigger the vulnerability. In the Test stage, the Patch Validation expert tool is used to confirm that the patch is effective and fully closes the previous vulnerability and that the applied security patch resolves the issue without introducing new problems. If the vulnerability has not been fully closed, the IDE reporting shows the open vulnerability. Finally, the IntelliSecTest expert tools are used to perform Regression Testing to check the codebase for any remaining or new vulnerabilities.

By providing the unpatched code of the SUT along with an input that triggers the known vulnerability, the IntelliSecTest solution generates a test suite that triggers this vulnerability in many different ways. Static analysis helps to identify the specific location of the vulnerability in the source code. The developer can use both information to efficiently develop a security patch and validate it through the generated test suite. Once the patch has been confirmed to be effective and not overfit to a specific proof of concept, the IntelliSecTest solution performs

further regression tests to prevent that further vulnerabilities are introduced by the patch. Figure 8 shows the IntelliSecTest solution for the security patch validation (Workflow 3) and the involved IntelliSecTest expert tools.

The Patch Validation uses directed grey-box fuzzing to generate more test cases that trigger the vulnerability in the unpatched version of the SUT. These new test cases differ in their code coverage profile from the one provided by the developers, meaning that they will reach other locations in the code when executed. This way, the new test cases have the potential to trigger the bug in spite of the patch, since an incomplete patch might not consider all possible paths through the code that lead to its vulnerable part. The generated test cases are then used as input for the patched SUT. Address-sanitation is used to check if the vulnerability can be triggered and thus reveals a blind spot in the patch.

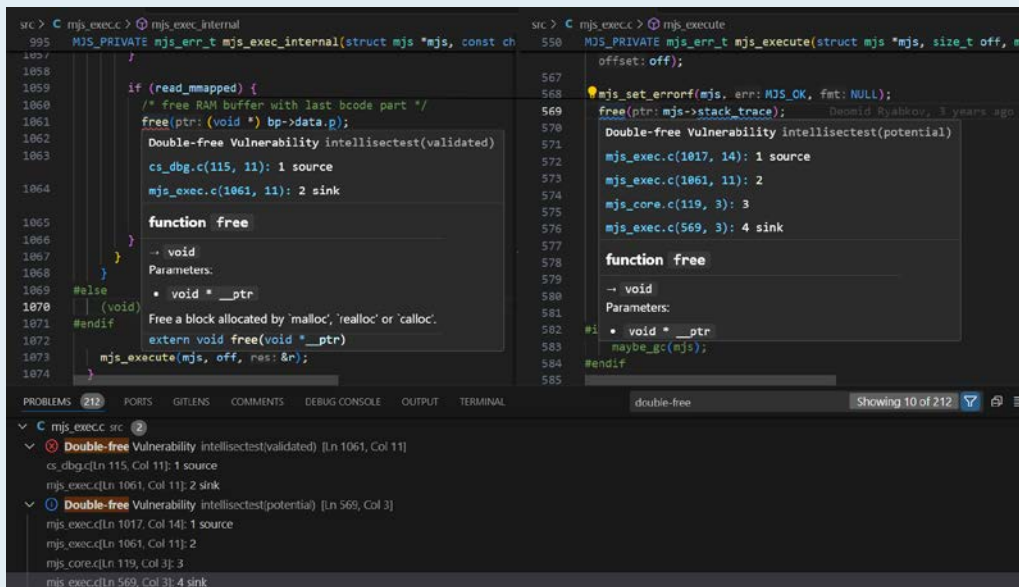


Figure 9: Example of IDE integration via LSP. Confirmed vulnerabilities have red error squiggles instead of the informational blue squiggles. The vulnerability path can be used for navigation, as shown with the highlighted source tree at the bottom.

4.3 Integration

4.3.1 IDE: VS Code

The IntelliSecTest tool is integrated into VS Code³ to inform developers about security vulnerabilities during the development. Using MagpieBridge⁴ and the Language Server Protocol (LSP)⁵, IntelliSecTest provides seamless and extensible integration with most IDEs. Therefore, the integration into development environments such as VS Code, IntelliJ IDEA products and Eclipse works out-of-the-box.

Figure 9 shows the integration of IntelliSecTest into VS Code. Static analysis results are initially reported as vulnerability candidates (indicated by a blue squiggly underline, as shown on the right side of the image and by an information symbol in the problems window) until they are confirmed. Once the analysis results are confirmed, IntelliSecTest raises the severity to an error (red squiggles, shown in the left side of the figure and a cross symbol in the problems window). Differentiating severity levels of warnings within the IDE allows the developers to hide unconfirmed vulnerabilities and thus, prioritize fixes accordingly. In addition, developers can interact with the diagnostics to navigate through the vulnerability path reported by IntelliSecTest's analysis tool (see problems window in Figure 9): The integration of IntelliSecTest with VS Code is enabled by a simple configuration file that is specified by the user. This file contains information about the compilation commands and the connectivity information (URL and port) of the IntelliSecTest cluster.

³ <https://code.visualstudio.com/>

⁴ <https://github.com/MagpieBridge/MagpieBridge>

⁵ <https://microsoft.github.io/language-server-protocol/>

4.3.2 CodeChecker

CodeChecker⁶ is a web application that runs various static code analysis tools and displays their results. It can display the results from IntelliSecTest in real time as they occur. It is convenient to get an overview of all the results of the analysis and, e.g., plan a strategy to solve the problems. For each Continuous Integration (CI) run, a link can be generated that displays the analysis results. Figures 10 and 11 show a screenshot of CodeChecker with results generated by IntelliSecTest for a simple demo example.

4.4 Technical Architecture and System Requirements

IntelliSecTest is built on the microservices architecture paradigm, which structures an application as a collection of loosely coupled services, each of which implements different functionality. IntelliSecTest's concrete implementation uses Docker[®] and Kubernetes to realize those design goals, among other technologies. Docker[®], a platform used for automating the deployment, scaling, and management of applications, is leveraged to implement the microservices. By encapsulating each service in a Docker container, the services can be tested, deployed, scaled, and updated independently. Kubernetes is an open-source platform for managing containerized workloads and services, and provides a robust framework for running distributed systems resiliently. It handles the scaling and failover of applications, and offers various deployment patterns. It is also highly configurable so that IntelliSecTest can be deployed

⁶ <https://codechecker.readthedocs.io/en/latest/>

Report hash	File	Message	Checker name	Analyzer	Severity	Bug path length	Latest review status	Latest detection status
9455b98744...	/share/code/asan/src/echo.c@_Line_3_0	attempting double-free on 0x603000000040 in thread T0:	AddressSanitizer	asan	H	4		
373428c68f...	/share/code/asan/src/echo.c@_Line_3_5	attempting free on address which was not malloc()-ed: 0x7fbc00ed8060 in thread T0	AddressSanitizer	asan	H	4		

Rows per page: 25 1-2 of 2

Figure 10: CodeChecker shows a list of crashes to get a fast overview. The crashes can be sorted and filtered.

in different scenarios with different hardware (e.g., on a single server or a cluster of servers). One of the standout features of the IntelliSecTest solution is its extensibility and flexibility. The innovative design allows users to seamlessly add new micro-services tailored to their specific needs, whether for analysis, execution, reporting, or other essential tasks. By empowering developers to integrate additional functionalities effortlessly, IntelliSecTest ensures that other security testing frameworks can evolve and adapt. This adaptability not only streamlines the testing processes but also enhances the overall efficiency and effectiveness of your security initiatives.

With the ability to run multiple analyses in parallel, the IntelliSecTest solution delivers efficiency. This powerful capability ensures that multiple analysis, execution, and reporting tasks can be performed simultaneously, dramatically reducing overall testing time and accelerating your development cycle. In IntelliSecTest there is also experimental support for CI via GitLab. CI is a development practice where developers integrate their code into a shared repository frequently, preferably several times a day.

Figure 11: CodeChecker shows a crash that has been reported by ASan within the IntelliSecTest solution. It also shows a stack trace and highlights the corresponding lines of the source code.

```

... (origin/master) | ... Found in:   job_63:echo.c:L30 4
25
26 void double_free() {
27     int *foo = (int *)malloc(32);
28     foo[0] = 1337;
29     free(foo);
30     free(foo);
31 }
32
#0 0x49bf12 in free (/share/testcases/6481222685573904611/tc_6481222685573904611_2+0x49bf12) #1 0x7f8598a815de in double_free /share/code/asan/src/echo.c:30:3 #2 0x4cdfa9 in testcase_0 /share/testcases/6481222685573904611/tc_6481222685573904611_2.c:19:2 #3 0x4ce029 in main /share/testcases/6481222685573904611/tc_6481222685573904611_2.c:25:3 #4 0x7f8598776d8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) #5 0x7f8598776e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f) #6 0x41f294 in _start (/share/testcases/6481222685573904611/tc_6481222685573904611_2+0x41f294)
3 < #1 0x7f8598a815de in double_free /share/code/asan/src/echo.c:30:3 >
4 < attempting double-free on 0x603000000040 in thread T0:
   For more information see the checker documentation.

```



Each integration can then be verified by automated builds and tests. The primary goal of CI is to catch and address bugs faster, improve software quality, and reduce the time to validate and release new software updates. This is where IntelliSecTest can be used for nightly or merge request related in-depth analysis of the code. For GitLab we provide a `.gitlab-ci.yml` file that starts the analysis with IntelliSecTest. The configuration can be adjusted to only run at certain points of time, on merge requests and also an analysis timeout can be set to limit resource usage (such as time, CPU, RAM, etc.).

Thanks to the microservice architecture, IntelliSecTest's system hardware requirements can be efficiently scaled. The minimum requirements are tailored to the system under test and the workflow used. During development we tested the communication between each microservice, representing the minimal resource requirements for the IntelliSecTest solution.

Integration tests require only two cores and 6 GB of memory to complete within 10 to 60 seconds. The hardware requirements have been obtained using real-world examples, mJS and libTiff. Each with a size of 35 K lines of code (KLOC), has specific resource requirements for testing which can be seen in Table 1.

These are just examples, but the architecture is designed to support and cater to your specific needs, whether you need to use some of the microservices or combine them.

Table 1: Minimum and Recommended System Hardware Requirements for Common Tasks. CPU requirements can be adjusted based on runtime, while memory requirements remain fixed. Current runtime values are suggested for daily tasks, but higher values increase the likelihood of achieving greater coverage. For release testing, a runtime of 24 hours is recommended.

Minimum and Recommended System Hardware Requirements for Common Tasks

	CPU Cores	Memory (GB)
Minimum hardware requirements	2	6
Recommended hardware requirements	5	11

5. Case Studies

Demonstrating the effectiveness of the IntelliSecTest solution on the open-source software projects mJS and libTiff.

To assess the effectiveness of the IntelliSecTest solution, we conducted a series of experiments on two SUTs: the open source softwares mJS⁷ and libTiff⁸. These SUTs were chosen due to their documented vulnerabilities (as referenced in the CVE databases) and their widespread usage. In this section, we illustrate how the IntelliSecTest solution enhances developer workflows through these case studies.

5.1 Testing Proprietary Code: mJS

Our first case study is conducted from the perspective of a development team which decides to systematically address the security of their project (namely, a JavaScript interpreter). Since a lot of code has been written, the team decides to rely on automated tools to avoid the manual effort of inspecting the code through reviews. Since the team has no experience in using automated security testing tools and interpreting their results, they are looking for a holistic solution that provides results in a developer-friendly way, within the IDE.

The IntelliSecTest solution is a perfect fit for the development needs, as it can be run fully automated with minimal configuration and user interaction, delivering only the actual results inside the source code in the IDE of the developer's choice.

The IntelliSecTest solution provides a pre-built configuration, called job graph, that uses various static and dynamic analysis tools, to perform the security analysis required by the developer, and only requires inputs such as the location of the source code (e.g., URL of the repository) and compilation scripts. Since the IntelliSecTest solution reports findings as soon as they are confirmed, developers do not have to wait for the analysis to

be completed, but get the results presented within their IDE as soon as they arrive. In addition, executable test cases and stack trace information are provided with each confirmed vulnerability for analysis and debugging purposes.

We illustrate how to use the IntelliSecTest solution and what results can be expected using mJS as an example. mJS is a lightweight JavaScript engine designed for microcontrollers and other constrained devices. This engine is particularly useful for Internet of Things (IoT) applications, where it allows developers to write and deploy JavaScript code directly on microcontrollers, facilitating rapid prototyping and development of connected devices.

After running the IntelliSecTest solution for 24 hours, the developers were presented with **seven** actual vulnerabilities that had previously gone undetected. This immediate and accurate detection demonstrates the IntelliSecTest solution's capabilities to identify vulnerabilities. The results of the verification of the vulnerability candidates are summarized in Table 2, highlighting the effectiveness of the IntelliSecTest solution in identifying vulnerabilities and providing developers with the information they need to maintain a secure development life cycle.

During the static code analysis phase, IntelliSecTest identifies 35 vulnerability candidates, primarily related to memory management and buffer overflows. These vulnerability types are critical because they often represent significant security risks if left unaddressed. To investigate these potential vulnerabilities, IntelliSecTest generates specific test cases to exercise 22 of the 35 vulnerability candidates. This automation of test case generation ensures that the majority of potential vulnerabilities are thoroughly tested without the need for extensive manual effort.

After executing these test cases, seven candidates are confirmed as true positives. These true positives represent real security vulnerabilities, demonstrating the accuracy and effectiveness of the IntelliSecTest tool. By discovering these issues in

⁷ <https://github.com/cesanta/mjs>

⁸ <http://www.libtiff.org>

Performing 24 h vulnerability detection process for mJS

	Static Analysis (SA)	Verification of SA Findings (Constraint Solving)	Verification of SA Findings (Directed Fuzzing)
Test Cases	n/a	3 test cases on avg. (per vulnerability candidate)	450,000 test cases on avg. (per vulnerability candidate)
Duration	around 2 min (in total)	5:30 min on avg. per vulnerability candidate	1h per vulnerability candidate
Results	35 vulnerability candidates	7 confirmed vulnerabilities (plus 12 further vulnerabilities)	0 confirmed vulnerabilities

Table 2: Performing 24 hrs vulnerability detection process for mJS

3rd party or proprietary code, developers can take immediate action to implement the necessary fixes. In recent testing, the generation process for confirming seven vulnerabilities took only a few seconds. This rapid turnaround ensures that potential security issues are quickly verified, allowing developers to address them promptly. By significantly reducing the time between reporting and confirmation, the IntelliSecTest solution enhances the efficiency of your security testing workflow.

For the remaining 13 vulnerability candidates, where specific test cases cannot be generated, the IntelliSecTest solution performs targeted fuzzing. Each vulnerability candidate is thoroughly fuzzed for up to three hours. Despite these efforts, no true positives were identified within three hours per vulnerability candidate, strongly suggesting that these candidates are likely false positives. To quantify the unconfirmed vulnerability candidates, the IntelliSecTest solution provides a residual risk estimation, assigning a probability value to each vulnerability candidate that helps to assess the likelihood of each candidate being a true positive.

The automated generation of test cases and the use of targeted fuzzing for remaining candidates demonstrate the efficiency of the IntelliSecTest solution. It minimizes the reliance on manual testing, accelerates the debugging process, and ensures comprehensive code coverage. Unlike existing security analysis tools, the IntelliSecTest solution provides a more comprehensive and accurate security assessment by reporting exactly where the vulnerabilities are located and how they manifest during their execution. By pinpointing the exact lines of code where vulnerabilities reside, developers can quickly address issues and seamlessly integrate security fixes into their workflow. This efficiency not only accelerates the development

process by reducing the time and effort required for remediation, but also significantly reduces the risk of security breaches. For an example of a result within IDE, see Figure 9. The IDE integration provides both validated (left side) and potential vulnerabilities that have not yet been classified as true or false positives (right side). Developers are not only shown the (potential) vulnerability, they also get important information about the data flows that lead to the vulnerability.

Using the IntelliSecTest solution significantly increases both the speed and effectiveness of security testing and provides a comprehensive set of benefits:

Accurate identification of vulnerabilities

IntelliSecTest employs a combination of static and dynamic analysis to accurately identify vulnerabilities. This dual approach ensures a comprehensive detection of both code-level issues and runtime anomalies. By understanding the context in which the vulnerabilities occur, IntelliSecTest can generate highly targeted test cases. Static analysis can provide the exact line of code where the vulnerability resides, as shown in Figure 9, and dynamic security testing ensures that only true positives are reported. The figure shows that the developer is informed about a double-free vulnerability in line 1061.

Reduced manual effort

Developers are not required to manually review the code or write an extensive test suite for each vulnerability candidate, significantly reducing the time required for security testing. In the case of the experiment conducted, test cases for 22 vulnerability candidates are automatically generated without the involvement of a developer.

Vulnerability-specific test case generation

Instead of relying on generic test cases, IntelliSecTest generates tests based on the specific type of the vulnerability and its context within the source code. Vulnerability specific test generation ensures that the test case is most likely to exploit the vulnerability in question, increasing the likelihood of detecting true positives and saving resources.

Early and continuous feedback

The IntelliSecTest solution integrates seamlessly into the development CI/CD pipeline, providing continuous and early feedback to developers. Vulnerabilities are identified during development, thus accelerating the overall vulnerability fixing process.

5.2 Testing 3rd Party Code: libTiff

The second case study considers a development team that wishes to incorporate TIFF file processing functionality into its application. During the research phase, the team identified libTiff as a suitable candidate to use for this purpose. The development team locates the header files that export the functionality and becomes acquainted with the build steps. Prior to integrating the library into their application, the team aims to ascertain that it does not introduce any unknown vulnerabilities from libTiff. As the team lacks experience in security analysis, they intend to rely on a fully automated tool.

IntelliSecTest offers an effective solution for automated vulnerability detection, eliminating the need for in-depth security expertise or familiarity with the 3rd party code. It offers a pre-defined configuration for testing 3rd party code based on an automated fuzzing workflow, requiring the user to input only the location of the header files and the necessary build steps. The analysis results are presented in an accessible format, accompanied by detailed information on the nature of the vulnerability, its location, and the corresponding call stack.

We evaluated the capability of IntelliSecTest to test 3rd party code (cf. Section 4.2) on libTiff, as included in the Magma⁹ framework. As previously stated, libTiff is a library for processing images in the TIFF format. The library offers its functionality through an API comprising 182 functions and a total of 25,000 lines of code. The version included in Magma contains 14 “forward-ported” real-world bugs, as documented on the GitHub page for the project¹⁰. These bugs are included to test the ability to reach deep paths within the library as well as the ability of testing tools to trigger real-world bugs.

Once the location of the library header files and the build configuration steps have been provided, IntelliSecTest performs the subsequent steps without requiring user input. In the initial phase, IntelliSecTest’s static analysis tool identified 68 API functions as potential entry points for a fuzzing campaign. The results were then used to generate a fuzz driver, which was subsequently fuzzed for 24 hours with AFL++. Within this time frame, IntelliSecTest achieved an average line coverage of 18% while producing 814 crashes. The further post-processing of crash deduplication reduced the number of crashes to 172 (21%), which is the result presented to the user. For comparison, state-of-the-art deduplication tools such as Crashwalk reduce the same crashes merely to 613. At the same time the IntelliSecTest-generated driver can reach four Magma bugs and trigger three of those.

The integration of automated fuzz driver generation and deduplication in a tool like IntelliSecTest provides substantial benefits for development teams seeking to guarantee the security of their applications without extensive security expertise.

Reduced Manual Effort

The automatic generation of fuzz drivers eliminates the need for developers to manually write fuzzing scripts, thus streamlining the process. This significantly reduces the time and effort required to set up a fuzzing campaign, particularly in the case of 3rd party code. This allows developers to focus on other critical tasks. To initiate a fuzzing campaign, developers simply need to provide the location of the header files and the build configuration steps. The tool handles the rest, making the process accessible to those with limited security analysis experience.

Comprehensive Coverage

IntelliSecTest automatically identifies API functions as entry points, ensuring comprehensive testing of the library’s functionality and eliminating the need for the manual effort. For libTiff, 68 such potential entry points have been identified for fuzzing. The manual identification of these entry points would be extremely time-consuming and error-prone. It would require developers to meticulously analyze the codebase to pinpoint relevant API functions.

Enhanced Crash Deduplication

The automated deduplication of crashes reduces the number of unique issues that developers must investigate. This approach streamlines the analysis process by focusing on distinct vulnerabilities, rather than repeatedly examining similar crashes. In the absence of deduplication, developers would be confronted with the task of investigating 814 crashes, a significant increase from the 174 crashes reported by IntelliSecTest. Each of these crashes would require individual attention to determine their root causes and potential fixes, which would be a significant time commitment for the development team.

⁹ <https://hexhive.epfl.ch/magma/>

¹⁰ <https://github.com/HexHive/magma/tree/v1.2/targets/libtiff/patches/bugs>

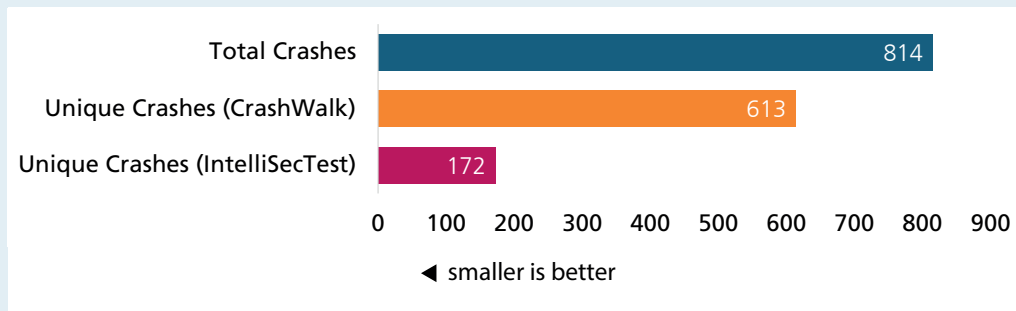


Figure 13: Crash deduplication by CrashWalk and IntelliSecTest

Superior Deduplication The tool’s deduplication capabilities exceed those of state-of-the-art tools such as Crashwalk. In the case study, IntelliSecTest reduced 814 crashes to 172 unique issues, whereas Crashwalk reduced them to 613. This highlights IntelliSecTest’s ability to minimize redundant crash reports more effectively, allowing developers to focus on addressing unique vulnerabilities.

Time Savings By reducing the number of crashes to be analyzed, deduplication saves valuable time for developers, allowing them to address critical issues faster.

Improved Resource Allocation With fewer crashes to investigate, development and security teams can allocate their resources more effectively, focusing on high-impact vulnerabilities.

Enhanced Reporting Deduplication provides more meaningful and manageable reports, making it easier for stakeholders to understand the security posture and make informed decisions.

5.3 Patch Validation: mJS

During the software testing phase, the aforementioned development team identified a potential security vulnerability. One of the automatically generated test cases confirmed that the vulnerability poses a security risk. The team was able to develop a corresponding patch thanks to the feedback provided by the IntelliSecTest solution regarding the code location responsible for the vulnerability. The effectiveness of the patch in fully eliminating the vulnerability is yet to be determined. It is possible that the patch may only address the issue on a particular input or configuration of the system.

To ensure the comprehensive elimination of the vulnerability by the security patch, IntelliSecTest offers assistance in the Patch Validation workflow. In this phase, new test cases that trigger

the vulnerability in the unpatched system are generated and then executed on the patched version. This process ensures that the patch not only addresses the vulnerability identified by the initial test case that exposed the vulnerability.

To demonstrate the process of Patch Validation, we conducted additional experiments on mJS. These experiments analyzed a segmentation fault¹¹, which was supposed to be fixed the current version [or release]. In a period of 24 hours, the IntelliSecTest solution was able to generate 64 test cases that triggered the bug in the unpatched version. Of the 64 test cases, one also uncovered a vulnerability in the patched system. This detection demonstrated that the initial patch did not fully address the underlying security issue. By revealing that the vulnerability was not fully fixed, IntelliSecTest demonstrates its power for supporting robust and thorough security measures. This proactive approach not only improves the quality of patches but also significantly enhances the security and reliability of the application.

The IntelliSecTest solution offers a comprehensive set of benefits that streamline the patch management process, enhance security, and reduce operational costs:

Reduced Number of Patch Revisions

The validation of security patches has the effect of reducing the number of patch revisions required. By thoroughly validating patches before deployment, the IntelliSecTest solution ensures that patches are effective and reliable from the first release. IntelliSecTest’s patch validation saves valuable time and resources by minimizing the need for subsequent revisions and updates. For instance, the incomplete patch for mJS’ segmentation fault could have been found directly during the original patch development [or validation]◀.

¹¹ <https://github.com/cesanta/mjs/issues/249>

System Hardware Consumption for Different Workflows.

Workflow	CPU Cores	Memory (GB)
Testing proprietary software (mJS)	5	11
Testing 3 rd party dependency (libTiff)	5	10
Patch validation	7	12

Table 3: System hardware consumption for different workflows

No Need to Design and Implement Test Cases

The automated creation and execution of 64 test cases that trigger the vulnerability significantly reduces the reliance on manually written test cases. This automation not only accelerates the testing process but also guarantees comprehensive and uniform testing, which is vital for identifying and addressing alternative inputs that could trigger the vulnerability.

Early Feedback on Patch Quality

One of the key benefits of IntelliSecTest's Patch Validation solution is the early feedback it provides on patch quality. By integrating seamlessly with development workflows, IntelliSecTest provides immediate insights into the effectiveness and stability of patches, offering valuable feedback that can be incorporated into the development process. This early feedback loop enables developers to implement necessary adjustments before full deployment, thereby ensuring higher quality and more reliable patches.

Reduced Attraction of Attackers

Effective patch validation and timely deployment reduce the window of opportunity for potential attackers. By ensuring the robustness and vulnerability-free status of patches prior to release, IntelliSecTest helps to mitigate the risk of exploits and enhance the overall security posture of your systems.

Lower Patch Deployment Costs

IntelliSecTest optimizes the entire patch management process, resulting in lower deployment costs. By reducing the number of patch revisions, manual test cases, and the time required for root cause identification, IntelliSecTest enables a more efficient and cost-effective deployment process. This not only saves direct costs induced by development but also minimizes the indirect costs associated with system downtime and security breaches.

5.4 Resource Consumption

Considering the resource consumption of security testing tools is critical for optimizing performance and ensuring the efficient integration into development environments. In Table 3, we present the relevant metrics of the IntelliSecTest solution, focusing on CPU and memory usage during the testing process of the previously described case studies. By showcasing these metrics, we provide a comprehensive view of the resource demands associated with the IntelliSecTest solution. This information helps developers and organizations to estimate the hardware consumption of the IntelliSecTest solution, ensuring a balance between thorough security testing and optimal resource utilization.

One of the key advantages of the IntelliSecTest solution is its adaptability to varying resource availability. The resource consumption can be restricted based on the system's capacity, ensuring that the testing process does not overwhelm the development environment. This flexibility allows to use the IntelliSecTest solution both on high-end servers and on more resource-constrained machines (see minimal system requirements in Table 1).

6. Summary

The IntelliSecTest solution implements a highly effective approach to security testing that helps manufacturers to meet the requirements from forthcoming legislation, such as the CRA.

The IntelliSecTest solution offers a cutting-edge and highly effective approach to application security testing. This solution helps manufacturers to meet regulatory requirements as set out in legislation such as the CRA. IntelliSecTest automates the process of identifying security vulnerabilities in both proprietary and 3rd party code and effectively supports remediating them, leading to accelerated bug fixes and comprehensive security coverage. Integration into common IDEs allows developers to be precisely alerted to potential weaknesses in the code. By precisely identifying vulnerabilities through comprehensive static and dynamic analyses, as well as targeted fuzz tests, the IntelliSecTest solution offers an efficient and thorough security assessment. The solution allows for the rapid and effective development of security patches and provides development teams with specific information on vulnerability locations in the code. The Patch Validation workflow within IntelliSecTest ensures that developed security patches are subjected to rigorous testing to eliminate identified vulnerabilities without introducing new ones, thus maintaining software integrity and security. Moreover, IntelliSecTest not only improves the speed and efficacy of security testing but also considerably reduces the risk of security breaches.

IntelliSecTest can be easily integrated into existing CI environments like GitLab. The IntelliSecTest solution helps to reduce the number of patch revisions, automate test case generation, accelerate vulnerability identification, and provide early feedback on patch quality. By integrating into CI environments, IntelliSecTest offers a flexible and extensible solution that helps manufacturers optimize their security processes and meet regulatory requirements effectively.

By adopting innovative security testing tools and methodologies, organizations can strengthen their security posture, adapt to regulatory changes and stay ahead in the ever-evolving cyber-security landscape. IntelliSecTest is the tool which helps you meet those demands.

Imprint

**Fraunhofer Institute for
Open Communication Systems FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany**

The IntelliSecTest solution was developed in a joint research project by the four Fraunhofer institutes IEM, AISEC, FKIE, and FOKUS. The work was supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231.

Publisher

Fraunhofer Institute for
Open Communication Systems FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

Authors

Eric Bodden, Matthias Meyer, Sriteja Kummita,
Fabian Schiebel, Lucas Briese (Fraunhofer IEM)

Julian Horsch, Konrad Hohentanner, Patrick Herter,
Vincent Alhrichs (Fraunhofer AISEC)

Elmar Padilla, Martin Clauß (Fraunhofer FKIE)

Martin Schneider, Ramon Barakat, Roman Kraus,
Fabian Jezuita (Fraunhofer FOKUS)

Design

Ivy Kunze

Illustration

Simone Geppert-Dahlhorst

Project Partners

Fraunhofer Institute for Mechatronic Systems Design (IEM)

Fraunhofer Institute for Applied and Integrated Security
(AISEC)

Fraunhofer Institute for Communication, Information
Processing and Ergonomics (FKIE)

Fraunhofer Institute for Open Communication Systems (FOKUS)

Picture Credits

title: istock / Laurence Dutton

page 5: istock / narvikk

page 16: istock / gorodenkoff

all other graphics by Fraunhofer

© Fraunhofer FOKUS,
Berlin 2024

Contact

Prof. Dr. Eric Bodden
Director Software Engineering and IT Security
Phone +49 5251 5465-150
eric.bodden@iem.fraunhofer.de

Fraunhofer IEM
Zukunftsmühle 1
33102 Paderborn, Germany
www.iem.fraunhofer.de

Dipl.-Inform. Martin Schneider
Head of Testing, Quality Engineering
Phone +49 30 3463-7383
martin.schneider@fokus.fraunhofer.de

Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
www.fokus.fraunhofer.de

