The Impact of the European Product Liability Directive on Software Engineering

Doriana Cobârzan[1][a], Richard Bubel[2][b] and Torsten Ullrich[1,3][c]
[1]*Fraunhofer Austria Research GmbH, Klagenfurt am Wörthersee, Austria*
[2]*Software Engineering, Technische Universität Darmstadt, Darmstadt, Germany*
[3]*Institute of Visual Computing, Graz University of Technology, Graz, Austria*

Keywords: Product Liability Directive, Software Quality, Software Supply Chain, Risk Management.

Abstract: The European Commission's revised Product Liability Directive was signed in October 2024 and will come into force in 2026. The revision extends the concept of a product to include software and software-based services, and significantly strengthens the legal rights of customers in the event of damage caused by software. This makes liability issues a key aspect of software development. The precise manner in which national legislation is to be drafted and interpreted remains to be clarified. However, the general direction has been sufficiently outlined to enable the implementation of preventative measures, which are discussed briefly here. This article looks at the legal implications of the directive for software producers, focusing on third-party components. It also discusses guidelines to ensure high software quality and improve the legal position of the producer. The present work is concerned exclusively with the Product Liability Directive, notwithstanding its embedding within a framework of regulations, including, for example, the AI Act and the General Data Protection Regulation.

1 INTRODUCTION

The European Single Market is an economic area comprising several national markets of the European Union Member States, which are characterised by the absence of internal borders. In addition to guaranteeing the four fundamental freedoms (free movement of goods, persons, services, and capital), the European Single Market aims to ensure growth, maintain and improve competitiveness, and promote job creation. The advantages for consumers include a greater variety of goods and services, as well as lower prices. Competition between companies also leads to improvements in the quality of goods and services. It facilitates the ability of citizens to find employment and to reside in EU Member States. As part of the European Single Market, national laws are harmonised to create a more uniform legal framework. The Product Liability Directive (85/374/EEC) has been an essential part of this harmonised framework since 1985 and remains in force today, unaltered.

In view of the recent substantial progress, especially in the field of information and communication technology (ICT), a thorough review and modernisation of the Product Liability Directive was imperative.

On the 28th of September, 2022, the European Commission published its proposal for a directive on the liability of defective products, which revises the existing Product Liability Directive that was adopted nearly 40 years ago. Following a two-year period, the directive was formally signed on the 23rd of October 2024 and subsequently published in the Official Journal on the 18th of November 2024 (2024/2853). The new directive will be implemented into national law and will come into force 24 months after the directive enters into force, i.e. in 2026. The precise manner in which national legislation is to be drafted and interpreted remains to be clarified. However, the general direction has been sufficiently outlined to enable the implementation of preventative measures, which will be discussed briefly here. This work focuses exclusively on the Product Liability Directive. While it is part of a broader European regulatory initiative, the focus remains on the Product Liability Directive only.

[a] https://orcid.org/0009-0000-4243-8084
[b] https://orcid.org/0009-0003-4847-4707
[c] https://orcid.org/0000-0002-7866-9762

Cobârzan, D., Bubel, R. and Ullrich, T.
The Impact of the European Product Liability Directive on Software Engineering.
DOI: 10.5220/0013363100003928

In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 626-634
ISBN: 978-989-758-742-9; ISSN: 2184-4895

2 SOFTWARE IS A PRODUCT

The European Union's updated Product Liability Directive (see https://eur-lex.europa.eu) expands the definition of a product to reflect the evolving landscape of software-driven goods and digital ecosystems. In detail, Article 4(1) defines that " *'product' means all movables, even if integrated into, or interconnected with, another movable or an immovable; it includes electricity, digital manufacturing files, raw materials and software* ". In other words, the product concept is explicitly extended to digital manufacturing files (e.g. 3D printing files), raw materials (e.g. gas and water) and any software, including stand-alone software and AI systems (except free and open-source software).

2.1 Liable Parties

The economic operators concerned, i.e. potentially liable parties, include "*the manufacturer of a defective product;*" (Article 8(1)a) and "*the manufacturer of a defective component, where that component was integrated into, or inter-connected with*" (Article 8(1)b). By definition, a manufacturer is "*any natural or legal person who:*

(a) *develops, manufactures or produces a product;*

(b) *has a product designed or manufactured, or who, by putting their name, trademark or other distinguishing features on that product, presents themselves as its manufacturer; or*

(c) *develops, manufactures or produces a product for their own use;*" (Article 4(10)).

In the case of a manufacturer established outside the EU, the liable parties are the importer of the product/-component, the authorised representative of the manufacturer (in terms of product safety law) and the fulfilment service provider (storage, packaging and shipping service provider). If none of the above can be identified, the liable parties are the distributor and the online platform provider.

The directive defines software as a product, regardless of how it is delivered or used. It can be embedded in a device, used via a network or cloud, or delivered through a software-as-a-service model. Strict liability also applies to related services. The directive says software producers or developers, including AI system providers, are the manufacturer. They could also be held liable for updates, improvements or machine learning algorithms. Digital technologies such as AI allow manufacturers and developers to exercise control over products even after they have been placed on the market or put into service.

2.2 Defect

According to the Product Liability Directive a "*product shall be considered defective where it does not provide the safety that a person is entitled to expect or that is required under Union or national law*" (Article 7(1))). The directive presents a non-exhaustive list of factors that may be relevant in the assessment of defectiveness:

- "*the presentation and the characteristics of the product, including its labelling, design, technical features, composition and packaging and the instructions for its assembly, installation, use and maintenance;*" (Article 7(2)a),
- "*reasonably foreseeable use of the product;*" (Article 7(2)b),
- "*the reasonably foreseeable effect on the product of other products that can be expected to be used together with the product, including by means of inter-connection;*" (Article 7(2)d),
- "*relevant product safety requirements, including safety-relevant cybersecurity requirements;*" (Article 7(2)f),
- "*in the case of a product whose very purpose is to prevent damage, any failure of the product to fulfil that purpose*" (Article 7(2)i).

A noteworthy distinction between the previous directive and the current one is the timing for determining defectiveness. In the future, the determination of defectiveness will not only consider the time of placing the product on the market, but also the period during which the product remains under the control of the manufacturer after being placed on the market. This period may be affected by factors such as the presence or absence of software updates and upgrades or a significant modification (Article 7(2)e).

2.3 Damage

With respect to potential damages, and in recognition of the increasing importance and value of intangible assets, the destruction or corruption of data, such as digital files deleted from a hard drive, will also be covered, including the cost of recovering or restoring that data.

The possibility of setting financial limits on the liability, e.g., via end-user licence agreements, etc., was removed from the revised Product Liability Directive. Furthermore, the limitation period within which an injured person may claim compensation for personal injury under the directive has been extended from 10 to 25 years if, according to medical findings, the symptoms of the personal injury appear late.

The burden of proof for the injured party remains unchanged; however, new rules of presumption will be introduced. It will henceforth be sufficient for the claimant to demonstrate the likelihood of a product defect in order to successfully pursue claims for damages based on strict liability. The duty to disclose evidence represents a significant development in the judicial enforcement of claims. Once a claimant has established their claim for damages, the defendant is obliged to disclose relevant evidence. In the event of the defendant failing to do so, rules of presumption will apply to the claimant's benefit.

2.4 Supply Chain

The intricate nature of modern supply chains, particularly those that cross national borders, renders them as complex systems. It is anticipated that companies will be required to oversee the management of diverse and geographically distributed supply chains, which may include multiple tiers of contractors and subcontractors in addition to the normal operations of the company. The term "supply chain liability" denotes the legal obligation of a corporation to compensate for damages incurred by its business partners, which may include suppliers, subcontractors and even customers. The legal theory of corporate supply chain liability postulates that a company can be held liable for damage-causing events in its supply chain if it fails to prevent such damage in contravention of a legal or moral obligation to do so.

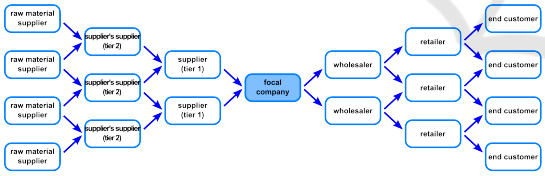

Figure 1: A focal company or product is characterised by a supply chain network structure comprising suppliers and customers in tiers.

These concepts are now being explicitly applied to software with one important exception; free and open-source: "*This Directive does not apply to free and open-source software that is developed or supplied outside the course of a commercial activity*" (Article 2(2)). In terms of product liability, it does not matter whether the company is "at fault" or not; it is liable regardless of its own culpability. This highlights the interdependence of one's own software or software-based services on third-party libraries. There is a potential risk that the chain of liability (along with the question of compensation and regress along the supply chain) may be disrupted at the European borders or by open-source exceptions.

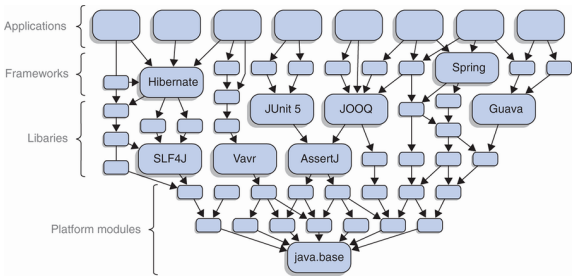

Figure 2: The fundamental premise of a software library is the efficient utilisation of resources through the reuse of existing components. Once a solution has been implemented, it need not be re-solved if it has been solved correctly and comprehensively in the first place. However, the act of reuse inherently introduces a dependency, which may have implications for the future. *image source:* (Parlog, 2019).

The aim of supply chain risk management is to reduce the vulnerability of the supply chain as a whole. This can be achieved by taking a coordinated, holistic approach, which ideally involves all stakeholders in the supply chain working together to identify, analyse and address potential failure points or modes within or affecting the supply chain (Wieland and Wallenburg, 2015). It would be remiss not to consider all the risks that could affect the supply chain. These could include, for example, quality, safety, resilience and product integrity.

3 SOFTWARE DEPENDENCY

The interdependencies of a company and/or a product are represented in a network as a double pyramid. On one side is the network of suppliers, comprising tier #1, tier #2, and so forth. On the other side is the customer network, also arranged in tiers. The company or product under consideration is situated in the middle (see Figure 1). It is imperative that these dependencies are considered in the context of software development with regard to liability issues.

The simplified representation of hypothetical Java applications in Figure 2 demonstrates that this network can rapidly expand to become complex and challenging to navigate. In particular, the use of tools for automatic package management (Pashchenko et al., 2020) may result in dependency confusion (Neupane et al., 2023) or in a lack of awareness regarding the depth and extent of the network (Islam et al., 2023).

In the fast-moving field of software development, there is a growing trend towards building applications by integrating external libraries, frameworks, tools and other software components. This approach

has the potential to significantly accelerate development and drive innovation. By allowing developers to leverage prebuilt functionality, it enables them to concentrate on creating application-specific features instead of reinventing foundational components. It is important to note that while third-party components can offer significant benefits, they may also present certain risks.

It would be advisable for organizations to take ownership of the code they integrate, even if they did not author it. Modern practices such as containerization (Watada et al., 2019) and Infrastructure-as-Code (IaC) (Bentaleb et al., 2022) have made it increasingly common for developers to package applications in containers and deploy them with IaC solutions like Kubernetes and Terraform. While these technologies offer the potential for scalable, automated deployments, they also introduce challenges such as vulnerabilities, version conflicts, and integration failures. These issues further intensify the need for meticulous dependency management to ensure the security, reliability, and stability of software.

3.1 Dependency Tracking

In the context of security, the concept of risk and the associated risk management techniques have become integral to the field of software engineering. A variety of dependency management strategies may be employed, many of which consider the entirety of the software supply chain. Among these, zero-trust software supply chains stand out as a potentially valuable approach, emphasising practices designed to mitigate the risks of compromised dependencies (do Amaral and Gondim, 2021). This strategy is predicated on the assumption that no component, whether direct or transitive, is inherently secure and that verification is required at every stage of the software life cycle.

Google's Supply Chain Levels for Software Artifacts (Enck and Williams, 2022) is a security framework designed to safeguard the software supply chain by ensuring the integrity and trustworthiness of software artifacts. It provides a structured set of guidelines and best practices that developers can adopt in order to mitigate risks such as tampering, unauthorised modifications, and compromised dependencies. A noteworthy standard in this domain is the Software Component Verification Standard (Open Worldwide Application Security Project, 2020), which establishes a benchmark for evaluating and verifying the security of software components. Collectively, these two frameworks offer complementary strategies for managing risks associated with the modern software supply chain, addressing the integrity of artifacts and the verification of component security (Tran et al., 2024). A software component is regarded as dependent on another if it is invoked or integrated with it, thereby forming essential connections that enable functionality. It is of paramount importance to implement effective dependency management strategies in order to guarantee the security, compliance and integrity of all dependencies. Notable incidents, such as the SolarWinds breach (Martínez and Durán, 2021) and the Log4Shell exploit (Everson et al., 2022), illustrate the substantial risks associated with unmanaged dependencies. In order to mitigate these risks, organisations are increasingly adopting tools such as Dependency-Track[1], Mend[2], and Sonatype Nexus Lifecycle[3]. The integration into the CI/CD pipeline (Mooduto et al., 2023) facilitate real-time tracking, vulnerability management, and compliance monitoring across the software life cycle.

The utilisation of Software Bills of Materials (SBOM) represents an emerging practice within the domain of modern dependency management strategies. This trend can be attributed to the necessity for enhanced transparency and security within the software supply chain. SBOM provide comprehensive inventories of all software dependencies, including both direct and transitive components, thus offering a clear view of the software's structure (Zahan et al., 2023). This enables teams to monitor the utilisation of every library, framework, and tool employed, along with the precise versions integrated into the application (Bi et al., 2024). Despite the advantages they offer, the implementation of SBOM is not without its difficulties: the scalability of SBOM in large and complex projects, and the necessity for expertise in their creation and maintenance, remain significant challenges (Xia et al., 2023).

Software Component Analysis (SCA) offers an additional efficient method for monitoring dependencies, assisting organisations in identifying and controlling both open-source and proprietary components within their software. Tools such as Dependency-Track (see above) integrate both SCA and SBOM to provide a unified solution, thereby enabling organisations to proactively manage dependencies, mitigate security risks and ensure long-term maintainability.

3.2 Dependant Notification

Software dependencies can be represented in a network as a double pyramid. Tracking software dependencies is one side of the double pyramid; the other

[1] see https://dependencytrack.org
[2] see https://www.mend.io
[3] see https://www.sonatype.com

side is "being tracked" by customers, or dependant notification. If a product has a defect, it should be recalled and repaired or replaced. In software engineering, this process is called a software update (Vaniea and Rashidi, 2016), (Rossel, 2017). The Product Liability Directive provides some defences to liability, but these defences do not apply in certain cases, for example, where the defect in the product is caused by a lack of software updates or upgrades necessary to maintain the safety of the product (Article 11(2)b, 11(2)c). To be on the safe side in this context, your software should have mechanisms to (1) track software in circulation, (2) reliably notify customers of updates and implement them, and (3) be able to withdraw insecure software from circulation.

1. For tracking the software and customer contacts, customer relationship management tools are available (Payne and Frow, 2016).
2. The notification of updates can be realised via regular automatic update checks. The software used by the customer checks whether an update has been provided in the meantime and actively prompts the customer to install it.
3. Deactivation can also be implemented via update mechanisms, but also via licence models (Ballhausen, 2019) and licence servers (Ferrante, 2006), in which the licence validity check also checks the software version and deactivates old/unsecure versions.

4 THIRD PARTY COMPONENT

In order to take account of the implications of the revised Product Liability Directive, a number of available options may be considered.

4.1 Elimination of Dependence

The term "dead code" is used to describe sections of a program that have been written but never executed, or sections of code that have been executed but have no impact on the program's output or functionality (Romano et al., 2020). Its presence results in an increase in the size of the codebase, which in turn makes the program more challenging to read, maintain, or debug (Malavolta et al., 2023).

There are a number of ways in which dead code can arise. This may include unreachable code, errors, obsolete features, legacy code, and so forth. Although it does not directly affect the execution of the program, it may have a significant drawback: the presence of dead code may result in inadvertent interactions with other components of the codebase, which could potentially pose a security risk if not properly isolated. It is therefore recommended that non-essential libraries be eliminated in order to minimize the total number of dependencies.

4.2 Contractual Regulation

The directive "*does not apply to free and open-source software that is developed or supplied outside the course of a commercial activity*" (Article 2(2)). This means that by integrating them into your own software product, there may be a liability risk if it is not also distributed as non-commercial, free and open-source software. One way to avoid this risk is to use software commercially. If, for example, a software library is offered both non-commercially and commercially, the exception does not apply to the commercial version. However, the legal circumstances and the then applicable regulations must be taken into account (see Section 2.1 on importers).

4.3 Complete Integration

Another option is to take ownership. In the context of third-party software libraries, taking ownership involves deeply integrating third-party libraries within one's own system (Greiler et al., 2015). This requires a comprehensive understanding of the library's functionality and structure. Consequently, it also necessitates maintaining the library's up-to-date status, ensuring its reliability, and developing a plan to address failures or replace it if necessary, in order to align its usage with the overall stability and goals of the project. In other words, the external library is treated in a manner consistent with that of one's own code, and thus maintained accordingly (see Section 5).

4.4 Risk Minimisation

In the event that the third-party component is indispensable for functional reasons and removal is not a viable option, if contractual regulation is not a possibility, or if taking code ownership does not appear to be a reasonable course of action (for example, due to the size of the library), risk minimisation may be considered as an alternative option.

In the field of cybersecurity, software isolation plays a crucial role in safeguarding computer systems from potential security threats and mitigating the risk of damage. It entails limiting the access and interaction between distinct software components to prevent unauthorised actions or malicious activities. There are two principal approaches to achiev-

ing software isolation: hardware-based isolation and software-based isolation (Pricop et al., 2020).

Hardware-based isolation employs the underlying hardware to separate software components. It utilises the capabilities of modern processors, including virtualisation and memory protection. Software-based isolation employs software techniques to separate two or more disparate software components. For example, operating systems employ a process isolation mechanism that prevents direct access to another process's memory or files. Furthermore, the user rights management system enables the restriction of access to the absolute minimum required for a process to function. Containerisation represents a lightweight form of virtualisation, whereby software components are isolated within containers. Both hardware-based and software-based isolation approaches have their respective advantages and limitations. Hardware-based isolation offers robust isolation guarantees and is frequently more secure. Nevertheless, it may necessitate particular hardware infrastructure and may prove to be less adaptable. Conversely, software-based isolation offers greater flexibility and can be implemented on a more diverse range of systems.

The prerequisite for the isolation of a software component into its own process and/or environment is the limited integration of that component into the software system (Richards and Ford, 2020). In the event that the degree of integration of a third-party component is minimal, it can be isolated through the implementation of software architectures, such as microservices. Consequently, each microservice is capable to operate in isolation. In contrast, components that are deeply integrated into the software system present a significant challenge to the implementation of effective isolation techniques (Shu et al., 2016).

5 FIRST PARTY COMPONENT

This section is concerned with code-level techniques that help to guarantee that software meets the relevant quality criteria, such as correctness or reliability.

The following sections are ordered according to the formal and mathematical training and experience required for application – from lightweight to heavyweight methods. Lightweight tools (see Section 5.1) are designed to identify generic and simpler property violations but may report a higher number of false warnings. Additional tools (see Sections 5.2ff.) allow one to determine whether violations exist or to verify more complex properties that a user may specify.

Nevertheless, the demarcation lines between different tools are not entirely distinct, but rather overlap in both directions. We want to highlight that methods and tools categorized to be of higher weight do not replace those that are more lightweight; rather, they are to be regarded as complementary. To illustrate, formal verification does not supersede the testing of software. Furthermore, testing can identify errors in hardware, operating system functions, or compilers.

All of the lightweight tools and many of the other tools support automation by integration into build systems; i.e., they can be used as components of continuous integration workflows. Consequently, software teams are able to identify and respond to issues within the context of their usual operational procedures.

5.1 Lightweight Methods

To guarantee that software meets the required quality standards, a number of methodologies can be employed, including code reviews, unit testing, and integration testing. In addition, lightweight static analysis techniques may be utilised, which are primarily concerned with the identification of code smells through the use of syntactic pattern matching and potentially simple control-flow and data-based analysis.

Code reviews are conducted by other team members with the objective of evaluating the effectiveness of code modifications in achieving the desired functionality or addressing identified issues. To enable reviewers to concentrate on potential enhancements or issues within the program logic, it is advisable that code reviews do not primarily focus on code style audits (code formatters can automatically ensure adherence to code style guidelines and formatting rules). The level of effort and formality associated with code reviews can vary considerably, spanning the spectrum from more traditional, structured approaches characterised by strict adherence to checklists and detailed scrutiny, as exemplified by the methodology proposed by (Beller et al., 2014), to more streamlined, lightweight techniques employed in numerous open-source software development projects. Code reviews are facilitated by hosting sites such as GitLab and GitHub, which provide views of the code changes and enable reviewers to place their comments and suggestions directly adjacent to the relevant changes. Code reviews have been demonstrated to be an effective method for reducing the number of bugs encountered after a release. However, potential issues must be taken into account (McIntosh et al., 2014; dos Santos and Nunes, 2018).

Once unit and integration tests have been written, their regular execution can be automated. Provided that sufficient code coverage has been achieved, they offer confidence that the software works as intended

for major use cases. Furthermore, a rigorous process of running automated tests automatically after code changes can also detect regressions. Numerous testing frameworks are available tailored to specific programming languages and deployment platform; such as JUnit[4], GoogleTest[5], and Cucumber[6].

Lightweight analysis tools ("linters") analyse source code for a range of issues, including, but not limited to, correctness, security, and concurrency issues. Subsequently, the tools generate a report that can be accessed and the identified issues can then be addressed. The majority of these tools can be configured such that reported issues are relevant beneficial warnings, while reducing the number of false positives. Such tools can be integrated into a continuous integration framework and executed automatically following, for example, each commit sequence. The option of defining a baseline (enabling the analysis of only new code or code changes) permits the gradual introduction of these tools into a legacy codebase, thus avoiding the necessity of addressing thousands of warnings. Popular tools in this category are SpotBugs[7], SonarQube[8], PMD[9], and many more.

5.2 Mediumweight Methods

Employing extended static analysis tools that are based on more sophisticated analysis techniques, such as symbolic execution or abstract interpretation, is a notable advancement. These tools offer more precise and comprehensive coverage of properties. Moreover, they are capable of checking for more complex properties, which may necessitate greater computational power and time. Some of the tools may require supplementary user annotations at code level, which consequently necessitates software engineers to understand the approach of the underlying analysis. However, this increases the accuracy of the tools and reduces the occurrence of false warnings.

The Checker Framework[10] offers pluggable type checking for Java and provides predefined type-based analyses to prevent errors associated with null pointers, resource leaks, lock checking (to avoid specific concurrency errors), and other issues. The software developer must provide additional type annotations, like `@NonNull`, for fields that cannot be null.

FBInfer[11] is a static analyser developed and utilised by Facebook. The tool is based on separation logic and bi-abduction to identify intricate issues within the source code, necessitating the examination of call chains. The following issues are among those that the tool can detect: livenesses, resource consumptions, buffer overruns, and memory safety issues. It offers support for C, C++, Objective-C, and Java. In contrast to the Checker Framework, it does not require the user to provide annotations.

Other analysis tools are capable of identifying complex errors. One such a tool is ENTROPOSCOPE (Dörre and Klebanov, 2016), which is designed to analyse pseudo-random generators for entropy loss. It has been used to identify potential security vulnerabilities in GnuPG and Libgcrypt (CVE-2016-6313) as well as in NSS/Firefox (CVE-2017-5462).

5.3 Heavyweight Methods

For software modules that require a high degree of confidence in their correctness, logic-based approaches such as model checking and deductive verification are to be considered. Model checking is applicable across the majority of software activities, from design phase to implementation. In particular, it can be applied when designing protocols (e.g., key exchange protocols) to verify specific properties.

CPAChecker[12] is a software model checker that automatically analysis programs and identifies generic issues and user specified property violations (Baier et al., 2024). The properties to be verified can be provided in the form of hand-crafted specification-automaton files. CPAChecker is also capable of checking for functional (data-related) errors by examining runs that may violate program assertions. It generates witnesses (test cases) that trigger such violations, facilitating the comprehension of the underlying issue. The witnesses can be used as supplementary regression test cases.

The preceding tools do not typically guarantee correctness; however, they are useful for identifying bugs. In particular, the latter provides a high degree of assurance regarding the quality of the software.

We conclude with an examination of a tool category that provides mathematical proof of a program's adherence to its specification. One should be aware that such guarantees are relative to often implicit assumptions like the absence of hardware or compiler errors. A comprehensive examination of these limitations can be found in (Livshits et al., 2015).

[4] see https://junit.org
[5] see https://github.com/google/googletest
[6] see https://cucumber.io/tools/cucumber-open
[7] see https://spotbugs.github.io
[8] see https://www.sonarsource.com
[9] see https://pmd.github.io
[10] see https://checkerframework.org
[11] see https://fbinfer.com
[12] see https://cpachecker.sosy-lab.org

The specification of program units (e.g., methods) can be achieved with contracts, as outlined by (Meyer, 1992). Method contracts state prerequisites that must be fulfilled by the caller at invocation time. If these requirements are satisfied, the method ensures that upon completion the resulting state satisfies the properties specified by the contract's postcondition.

Such specifications permit to express complex functional properties that can be proven correct using deductive verification systems such as Frama-C[13], GNATProve[14], KeY[15], or VerCors[16].

Although these techniques have their origins in academia, they have been successfully applied in industry and used in real-world case studies. To illustrate, Dafny (Leino, 2013), a programming language designed for the verification of software, is used by Amazon to validate components within their web services. Another tool, KeY (Ahrendt et al., 2016), has been used to verify several methods of the Java Standard Library, including the sorting algorithm for reference types, TimSort. The analysis carried out by KeY revealed the presence of a persistent bug in the underlying algorithm. A proposed solution was implemented and subsequently verified to be correct (de Gouw et al., 2019). Similarly, case studies have identified errors in the Java standard library. Furthermore, the sequential version of the ips[4]o sorting algorithm has been implemented in Java and verified to be correct (Beckert et al., 2024). It offers one of the fastest provably correct sequential algorithms.

Finally, proof assistants like Coq[17],Lean[18], or Isabelle[19], are highly expressive but require significant expertise in formal methods.

6 CONCLUSION

The modernisation of the Product Liability Directive serves to enhance the accountability of commercial software providers. We examined its implications in the context of the software producer and surveyed approaches to (i) dependency tracking to ensure that the software producer is in a position to react to defects in used third-party components; and (ii) software quality assurance tools to identify a wide variety of software defects before release. Therefore, in order to maximise readiness, the period prior to the implementation of the Directive should be used for proactive preparation, allowing software producers to adapt their processes and effectively mitigate potential risks.

[13] see https://frama-c.com

[14] see https://www.adacore.com/sparkpro

[15] see https://www.key-project.org

[16] see https://vercors.ewi.utwente.nl

[17] see https://coq.inria.fr

[18] see https://lean-lang.org

[19] see https://isabelle.in.tum.de

ACKNOWLEDGEMENTS

The work in this paper has received funding from the Austrian Research Promotion Agency (FFG) under Grant Agreement no. FO999899544; project "PREdictions for Science, Engineering N' Technology – PRESENT".

Furthermore, the work was supported by the DFG project "Forschungssoftware KeY" (BU 2924/3-1, HA 2617/9-1) and the ATHENE project "Model-centric Deductive Verification of Smart Contracts".

REFERENCES

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M., editors (2016). *Deductive Software Verification - The KeY Book – From Theory to Practice*. Springer.

Baier, D., Beyer, D., Chien, P., Jakobs, M., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M. L., Wachowitz, H., and Wendler, P. (2024). Software Verification with CPAchecker 3.0: Tutorial and User Guide. *Formal Methods FM (LNCS)*, 14934:543–570.

Ballhausen, M. (2019). Free and Open Source Software Licenses Explained. *Computer*, 52:82–86.

Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., and Witt, S. (2024). Formally Verifying an Efficient Sorter. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS (LNCS)*, 14570:268–287.

Beller, M., Bacchelli, A., Zaidman, A., and Jürgens, E. (2014). Modern code reviews in open-source projects: which problems do they fix? *Working Conference on Mining Software Repositories*, 11:202–211.

Bentaleb, O., Belloum, A. S. Z., Sebaa, A., and El-Maouhab, A. (2022). Containerization technologies: taxonomies, applications and challenges. *Journal of Supercomputing*, 78:1144–1181.

Bi, T., Xia, B., Xing, Z., Lu, Q., and Zhu, L. (2024). On the Way to SBOMs: Investigating Design Issues and Solutions in Practice. *ACM Transactions on Software Engineering and Methodology*, 33:149:1–25.

de Gouw, S., de Boer, F. S., Bubel, R., Hähnle, R., Rot, J., and Steinhöfel, D. (2019). Verifying OpenJDK's Sort Method for Generic Collections. *Journal of Automated Reasoning*, 62:93–126.

do Amaral, T. M. S. and Gondim, J. J. C. (2021). Integrating Zero Trust in the cyber supply chain security. *Workshop on Communication Networks and Power Systems (WCNPS)*, 5:1–6.

Dörre, F. and Klebanov, V. (2016). Practical Detection of Entropy Loss in Pseudo-Random Number Generators. *ACM SIGSAC Conference on Computer and Communications Security*, 23:678–689.

dos Santos, E. W. and Nunes, I. (2018). Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data. *Journal of Software Engineering Research and Development*, 6:14.

Enck, W. and Williams, L. (2022). Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security & Privacy*, 20:96–100.

Everson, D., Cheng, L., and Zhang, Z. (2022). Log4shell: Redefining the Web Attack Surface. *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 4:23010:1–8.

Ferrante, D. (2006). Software Licensing Models: What's Out There? *IT Professional*, 8:24–29.

Greiler, M., Herzig, K., and Czerwonka, J. (2015). Code Ownership and Software Quality: A Replication Study. *IEEE/ACM Working Conference on Mining Software Repositories*, 12:2–12.

Islam, S., Gaikovina Kula, R., Treude, C., Chinthanet, B., Ishio, T., and Matsumoto, K. (2023). An Empirical Study of Package Management Issues via Stack Overflow. *IEICE Transactions on Information and Systems*, E106-D:138–147.

Leino, K. R. M. (2013). Developing Verified Programs with Dafny. *International Conference on Software Engineering, ICSE*, 35:1488–1490.

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. (2015). In defense of soundiness: a manifesto. *Communications of the ACM*, 58:44–46.

Malavolta, I., Nirghin, K., Scoccia, G. L., Romano, S., Lombardi, S., Scanniello, G., and Lago, P. (2023). JavaScript Dead Code Identification, Elimination, and Empirical Assessment. *IEEE Transactions on Software Engineering*, 49:3692–3714.

Martínez, J. and Durán, J. M. (2021). Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. *International Journal of Safety and Security Engineering*, 11:537–545.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. *Working Conference on Mining Software Repositories*, 11:192–201.

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25:40–51.

Mooduto, A., Rijanto, E., and Pamuji, G. C. (2023). Optimization of Software Development Automation via CICD, Dependency Track, and AWS CodePipeline Integration. *International Conference on Informatics Engineering, Science & Technology*, 1:1–7.

Neupane, S., Holmes, G., Wyss, E., Davidson, D., and Carli, L. D. (2023). Beyond Typosquatting: An In-depth Look at Package Confusion. *USENIX Security Symposium*, 32:3439–3456.

Open Worldwide Application Security Project (2020). *Software Component Verification Standard*. OWASP Foundation, 1st edition.

Parlog, N. (2019). *The Java Module System*. Manning Publications, 1st edition.

Pashchenko, I., Vu, D.-L., and Massacci, F. (2020). A Qualitative Study of Dependency Management and Its Security Implications. *ACM SIGSAC Conference on Computer and Communications Security*, 27:1513–1531.

Payne, A. and Frow, P. (2016). *The Marketing Book*. Taylor & Francis Group, 7th edition.

Pricop, E., Fattahi, J., Dutta, N., and Ibrahim, M. (2020). *Recent Developments on Industrial Control Systems Resilience*. Springer Cham, 1st edition.

Richards, M. and Ford, N. (2020). *Fundamentals of Software Architecture – An Engineering Approach*. O'Reilly Media, Inc, 1st edition.

Romano, S., Vendome, C., Scanniello, G., and Poshyvanyk, D. (2020). A Multi-Study Investigation into Dead Code. *IEEE Transactions on Software Engineering*, 46:71–99.

Rossel, S. (2017). *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing Ltd., 1st edition.

Shu, R., Wang, P., Gorski III, S. A., Andow, B., Nadkarni, A., Deshotels, L., Gionta, J., Enck, W., and Gu, X. (2016). A Study of Security Isolation Techniques. *ACM Computing Surveys*, 49:50:1–37.

Tran, N. K., Pallewatta, S., and Babar, M. A. (2024). An Empirically Grounded Reference Architecture for Software Supply Chain Metadata Management. *International Conference on Evaluation and Assessment in Software Engineering*, 28:38–47.

Vaniea, K. and Rashidi, Y. (2016). Tales of Software Updates: The process of updating software. *Conference on Human Factors in Computing Systems*, 35:3215–3226.

Watada, J., Roy, A., Kadikar, R., Pham, H., and Xu, B. (2019). Emerging Trends, Techniques and Open Issues of Containerization: A Review. *IEEE Access*, 7:152443–152472.

Wieland, A. and Wallenburg, C. M. (2015). Dealing with supply chain risks: Linking risk management practices and strategies to performance. *International Journal of Physical Distribution & Logistics Management*, 42:887–905.

Xia, B., Bi, T., Xing, Z., Lu, Q., and Zhu, L. (2023). An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. *IEEE/ACM International Conference on Software Engineering (ICSE)*, 45:2630–2642.

Zahan, N., Lin, E., Tamanna, M., Enck, W., and Williams, L. (2023). Software Bills of Materials Are Required. Are We There Yet? *IEEE Security & Privacy*, 21:82–88.