Software Security Analysis in 2030 and Beyond: A Research Roadmap

MARCEL BÖHME, Max Planck Institute for Security and Privacy, Bochum, Germany
ERIC BODDEN, Faculty of Electrical Engineering Computer Science and Mathematics, Department of Computer Science, Software Engineering Group, Paderborn University, Paderborn, Germany
TEVFIK BULTAN, University of California at Santa Barbara, Santa Barbara, California, United States
CRISTIAN CADAR, Department of Computing, Imperial College London, United Kingdom of Great Britain and Northern Ireland
YANG LIU, Nanyang Technological University, Singapore, Singapore
GIUSEPPE SCANNIELLO, Department of Informatics, University of Salerno, Fisciano, Italy

As our lives, our businesses, and indeed our world economy become increasingly reliant on the secure operation of many interconnected software systems, the software engineering research community is faced with unprecedented research challenges, but also with exciting new opportunities. In this roadmap article, we outline our vision of software security analysis for the systems of the future. Given the recent advances in generative AI, we need new methods to assess and maximize the security of code co-written by machines. As our systems become increasingly heterogeneous, we need practical approaches that work even if some functions are automatically generated, e.g., by deep neural networks. As software systems depend evermore on the software supply chain, we need tools that scale to an entire ecosystem. What kind of vulnerabilities exist in future systems and how do we detect them? When all the shallow bugs are found, how do we discover vulnerabilities hidden deeply in the system? Assuming we cannot find all security flaws, how can we nevertheless protect our system? To answer these questions, we start our roadmap with a survey of recent advances in software security, then discuss open challenges and opportunities, and conclude with a long-term perspective for the field.

CCS Concepts: • **Software and its engineering**; • **Security and privacy**;

Additional Key Words and Phrases: SE2030, vision statement, perspective article

A part of the research presented in this article has been financially supported by the European Union NEXTGenerationEU project and by the Italian Ministry of the University and Research MUR, a Research Projects of Significant National Interest (PRIN) 2022 PNRR, project no. D53D23017310001 entitled "Mining Software Repositories for enhanced Software Bills of Materials (MSR4SBOM)." Part of the research presented in this article has also received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).
Authors' Contact Information: Marcel Böhme (corresponding author), Max Planck Institute for Security and Privacy, Bochum, Germany; e-mail: marcel.boehme@acm.org; Eric Bodden, Faculty of Electrical Engineering Computer Science and Mathematics, Department of Computer Science, Software Engineering Group, Paderborn University, Paderborn, Germany; e-mail: eric.bodden@uni-paderborn.de; Tevfik Bultan, University of California at Santa Barbara, Santa Barbara, California, United States; e-mail: bultan@ucsb.edu; Cristian Cadar, Department of Computing, Imperial College London, United Kingdom of Great Britain and Northern Ireland; e-mail: c.cadar@imperial.ac.uk; Yang Liu, Nanyang Technological University, Singapore, Singapore; e-mail: yangliu@ntu.edu.sg; Giuseppe Scanniello, Department of Informatics, University of Salerno, Fisciano, Italy; e-mail: gscanniello@unisa.it.

ACM 1557-7392/2025/5-ART144
https://doi.org/10.1145/3708533

ACM Reference format:
Marcel Böhme, Eric Bodden, Tevfik Bultan, Cristian Cadar, Yang Liu, and Giuseppe Scanniello. 2025. Software Security Analysis in 2030 and Beyond: A Research Roadmap. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 144 (May 2025), 26 pages.
https://doi.org/10.1145/3708533

1 Introduction

Over many years, the primary focus of the software engineering community has been on building larger and more reliable software systems with minimal effort. Today, we are proud to see the world's digital economy run reliably on a super-scale, ever-changing, hyper-connected network of software systems. With the power of the software supply chain, cloud computing resources at our fingertips, and a large automation ecosystem at our disposal, we can build complex systems from existing components at an unprecedented pace. Recent advances in AI promise further automation even of creative tasks (incl. auto-programming to build even larger systems). Our software systems are becoming more heterogeneous and extend further into the real world, as for instance, virtual reality, the **Internet-of-Things (IoT)**, autonomous cars, and robots.

Over the coming years, we anticipate that software security analysis will become another important focus of the software engineering community. How can we make our large, interconnected software systems robust against attacks? In this article, we draw a research roadmap toward 2030 and beyond. We identify concrete challenges and opportunities for the security analysis of the software systems of the future and provide specific directions of research for the software engineering community.

2 State of the Research and Practice

Software security analysis has seen significant progress over the last decade. Several program analysis techniques, ranging from static analysis [49, 55, 62] to fuzzing [126, 143, 202] to symbolic execution [42, 146, 166], have reached maturity, and many software systems have started to adopt them. Combined with other software development changes, such as new-generation **integrated development environments (IDEs)** [59, 186], better code hosting platforms [85], and higher adoption of code review and **continuous integration (CI)** [28, 170], this has enabled the development of larger and more complex software systems.

2.1 Software Analysis Techniques

This section briefly describes the most widely used software analysis techniques employed in practice, particularly in a security context:

Formal verification provides the highest level of security guarantees, with formal machine-checked proofs being able to establish the absence of certain classes of bugs. In the last 15 years, the field has seen tremendous progress, with researchers building several safety and security-critical software systems, such as the seL4 microkernel [113] and the CompCert compiler [120]. Yet, there are, unfortunately, two main inter-related limitations of formally verified systems: (1) they require years of PhD-level expertise, with the specification often larger than the verified code itself, and (2) the resulting systems lack many of the features and/or performance of their non-verified counterparts. In addition, evolving these systems is often costly, as the verification effort is not always modular.

Static analysis reasons about program code without executing it. Abstract interpretation [61] is a well-known type of static analysis, with sound reasoning, which can also be used for formal verification. Especially in the security domain, more lightweight vulnerability detection techniques have become very popular [122]. These frequently focus on identifying the most common kinds of vulnerabilities, e.g., the SANS 25 [65]. These techniques are usually based on dataflow analysis that—for scalability reasons—oftentimes forgoes complex reasoning about arithmetic and other aspects. To provide a "good signal" with few false positives, most such approaches these days also accept a certain level of unsoundness [149]. While recent work has shown that clever program abstractions and algorithms can improve analysis precision and speed at the same time [172], scalability remains a challenge when it comes to analyzing large code bases. Another major shortcoming of static analysis, shared with other techniques, is that analyzers need to be configured: they only report what they are configured to report, which is why they must be told, e.g., which particular API calls in which combination can lead to which kinds of vulnerabilities [157, 158]. In other words, static analysis also does not completely forego a specification, yet here one specifies vulnerability types, not program functionality.

Fuzzing [34], at a high level, involves testing software with randomly generated inputs. There are three main forms of fuzzing: blackbox, greybox, and whitebox fuzzing, with the latter also known as dynamic symbolic execution and discussed separately below. Blackbox fuzzing does not use any knowledge of the implementation nor any execution feedback other than what can be observed by running the software in a blackbox manner. At the same time, it requires domain knowledge to be effective, such as grammar for the inputs accepted by the program. The technique has been applied with a lot of success in several areas, for instance for optimizing compilers and **database management systems (DBMS)**, where it has found hundreds of critical bugs in mature compiler and DBMS implementations [118, 162, 197]. Greybox fuzzing is the most widely used form of fuzzing for general software systems, which is guided by the code coverage achieved by the generated inputs. It is typically combined with mutation-based input generation. Greybox fuzzing was pioneered by AFL [202] and now there are dozens of greybox fuzzers available for many different languages and domains. **Open source system (OSS)**-Fuzz [142] is a service provided by Google which runs several state-of-the-art greybox fuzzers on open-source software. To date, it has found over 10K vulnerabilities and over 36k bugs in over one thousand projects.

Dynamic symbolic execution [43, 48, 86, 146, 166], also referred to as *concolic execution*, *whitebox fuzzing* or just *symbolic execution*, is a program analysis technique that can systematically explore paths in a program by using an SMT constraint solver [70] to reason about path feasibility and generate concrete test inputs for each explored path. While the technique has a long history going back to the 1970s [56, 111], it is only in the last couple of decades that it has become practical, enabled by both algorithmic advances and practical tool design [29]. Nevertheless, important scalability challenges remain, both in terms of path explosion and constraint solving [48]. Dynamic symbolic execution has started to be adopted in practice [44, 45] by companies such as Fujitsu [121], Microsoft [86], and Samsung [116], and nowadays several mature open-source tools exist for different languages and platforms [42, 69, 146, 159].

Machine Learning (ML) was applied in software security analysis mostly to the problem of further automating code-level vulnerability detection, ranging from approaches that help configure traditional static dataflow analyses [157] to such that try to detect vulnerabilities solely using ML, without dataflow analysis [52]. For the latter case, while in cross-validation lab experiments, many approaches in this field at first showed extremely promising results [211], it has been unfortunately proven again and again that with current models the recognition quality quickly degrades when the ML models are tested on code on which they had not been trained [84, 115]. This result is similar to what has been observed for malware detection [19]. Another issue with many such classifiers is that they merely report that they suspect *some* vulnerability in a given method or even file—information that is hardly useful to developers and can at best help prioritize further vulnerability detection activities. While more recent approaches attempt to report findings on the line level [82] and try to report at least the given common weakness enumeration classification of the suspected vulnerability, it remains to be shown whether developers can act on such findings.

Runtime protection mechanisms are used during software deployment, targeting vulnerabilities that development- and testing-time activities have failed to detect. These techniques target specific classes of errors, typically memory vulnerabilities [178], and usually make it harder rather than impossible, for attackers to exploit those vulnerabilities. Well-know techniques include stack canaries [63], control-flow integrity [15], dataflow integrity [15], code-pointer integrity [117], and write integrity testing [16].

2.2 Detected Errors

Given that most of our security and safety-critical code continues to be developed in unsafe languages like C and C++, most attention has been given to memory safety errors [178]. On the one hand, techniques targeting such errors have been designed and adopted in practice, such as compiler sanitizers [168, 173], which can find bugs such as buffer overflows and use-after-free errors. On the other hand, techniques that can in principle find a broader class of bugs, such as static analysis and symbolic execution, have been primarily applied to find memory safety errors and injection vulnerabilities, both due to their importance and their easily available oracles [42, 49, 55, 166].

To go beyond memory safety, researchers have exploited differential and metamorphic oracles. For instance, differential testing has been extremely successful in the context of compiler testing [197], while various types of metamorphic transformations have been effective for DBMS [162].

Certain types of functional bugs may escape differential and metamorphic testing. Specifications have been for long proposed as a way to write correct software, but adoption has been limited, particularly in mainstream languages.

3 Challenges and Opportunities

As we look forward, we predict a substantial shift in the *key challenges* for security analysis. Not only will our software systems grow larger, more interconnected, more distributed, and evolve faster. We also expect to see a greater dependence on the software supply chain, a deeper integration with the physical world, and a broader range of important security properties beyond memory safety. There are also many *untapped opportunities* for us to develop the area of software security analysis. For instance, recent advances in ML hold a strong promise for non-traditional, empirical approaches for reasoning about the security of a software system [33]. Outside of the technical aspects, as the field matures, we should work on *pedagogical aspects*, such as establishing and teaching the foundations of software security, as well as *legal aspects*, such as questions of liability or accountability in the digital world.

For easy reference, each challenge or opportunity is identified uniquely using a counter and a name. For instance, *O1. This is a challenge or opportunity.*

3.1 ML and Security

Recent advances in ML promise a paradigm shift in software security analysis.

3.1.1 Security for ML. Traditional software systems are *programmed*, i.e., composed of human-written instructions for the machine to process inputs. So, most existing security analyses work by processing these instructions to reason about the security of the software system. However, we expect that future software systems, at least in part, will be *learned*. ML methods, such as **deep learning (DL)** and **large language models (LLMs)**, have proven extremely useful in the automation of certain tasks. For instance, rather than humans writing programs that provide precise instructions on how to generate a video, an image, or even source code from a text description (called prompt), it might be sufficient to train a ML model with large amounts of data.

O1. Analysis of ML Systems. How can we make statements about the properties of a program whose behavior emerges from a network of neurons with real-valued weights? From a *formal perspective*, we could investigate transpilation techniques [192] which turn a ML model into a traditional program (i) that is guaranteed to exhibit all and only those behaviors as the original model and (ii) that can be analyzed by existing software verification or static analysis tools. Skipping the intermediary traditional program, we could develop new analysis techniques that directly extract a formal model of computation representing all executions of that model and allow us to formally reason about the properties of that model. From an *empirical perspective*, we could investigate statistical methods or probably-approximately-correct learnability to quantify our uncertainty about empirical statements about a model's properties, given a sample from the operation distribution of inputs for that model [119]. We could employ experimental methods to make empirical statements about hyper-properties [57] of an ML-based software system (e.g., robustness quantifies the degree to which a small change in the input causes a change in the output of the model).

O2. Vulnerability Types in ML Systems. Which security flaws might exist in ML-based software systems that could be exploited with malicious intent, and how do we analyze and mitigate them? We already know about certain broad types of attacks on ML systems. In a data poisoning attack, the malicious actor manipulates the training data to influence the behavior of the trained software system that is eventually deployed. In a data privacy attack (e.g., membership inference, model extraction, or model inversion), a malicious actor can extract sensitive data from the deployed system. In a model evasion attack, a malicious actor learns to manipulate the deployed system's outcome, e.g., to evade detection. There are most certainly other types of vulnerabilities; some may be domain- or model-specific. How can we detect and mitigate them?

O3. Vulnerabilities in ML-Generated Code. Which security flaws are typically observed in ML-generated code and how do we detect them? How do we improve the security of ML-generated code? Can ML-generated code be more secure than human-written code? For little over a year, we have seen increasing adoption of ML-based programming assistants that can translate simple comments [134, 211] or complex Github Issues [208] into source code written in any programming language. Such ML assistants free developers from resolving mundane issues and help them focus on the high-level, creative part of the development process. However, these machine programmers are also known to "hallucinate" [150, 152]. Today, they tend to generate faulty and potentially insecure implementations. If the complicated, auto-generated code is accepted by the busy programmer without much reflection, ML-generated vulnerabilities might start to sneak into production code at an increasing velocity.

3.1.2 ML for Security. ML has always played an important role in software security analysis, e.g., for intrusion detection [72], malware analysis [182], and vulnerability detection [125]. Specifically, ML is used to identify patterns that are *correlated* with security flaws. While results on academic benchmarks seem promising, other factors, like over-fitting or spurious correlations, cannot be excluded as alternative explanations for the required code reasoning capabilities [22, 53, 163]. However, recent developments in LLMs, particularly toward systematic reasoning and planning

[189], and further developments in causal reasoning from data [153] hold exciting prospects for the application of ML to software security analysis.

O4. ML4SbD (Secure-by-Design). We anticipate that multi-model ML models, which, for instance, can work with not just text but also drawings, will reshape very soon the way in which we design and document software systems. One can easily envision systems that derive software architectures in a joint interaction with developers. But how can we make sure that ML models, if used in this way, come up with "the right" design? How can we factor in security? Will it be problematic if such models yield non-deterministic output? And can ML even help detect security flaws on the architectural level?

O5. ML4VD (Vulnerability Detection). How can we use recent advances in ML to swiftly analyze source code written in arbitrary languages, at an arbitrary scale, with a tolerable number of false positives or negatives? Existing approaches that work without executing the program require some human-provided encoding of the pertinent semantic rules of the programming language. ML can do away entirely with such semantic rules and—given enough training data—works for programs written in any language. For instance, *defect prediction* [107] has been developed over many years as an ML-based approach to identify potentially defective components via code property correlates. DL-based methods require substantial training data, and it is an open question if such data can at all be properly curated in sufficient amounts for these models to work effectively. However, recent advances in DL, LLMs, and statistical methods [33, 119] promise an entirely new form of reasoning over the properties of a program. As LLMs are known to make up facts ("hallucinate"), we should explore the limits of these approaches and identify where they work well and where they do not. Given the enduring success of existing formal/symbolic approaches in program analysis, we further envision the development of new neurosymbolic approaches.

O6. Tool Configuration or Assistance. How can we reduce the level of expert knowledge required to set up security tooling for a given software system most effectively? For interactive security tooling, how can we increase automation and further assist or even replace the engineer with ML? Most security tools are designed to be general, i.e., to work for any software system that is in scope. To adapt such a tool for a specific system, it needs to be *configured.* For instance, to attach a fuzzer to the system the user needs to write the required fuzz drivers [167] and provide the required input format or protocol [21, 154]. To enable a **static analysis security tool (SAST)**, the user needs to add it to the CI pipeline and possibly to the build process. Such a setup requires expert knowledge about both, the security tools as well as the software system. The recent success of LLMs in similar high-level, creative tasks (e.g., translating hundreds of pages of natural language protocol specification into a format that is usable by a fuzzer [132]) is a promising precedent. In the future, we envision that even the layman can set up security tooling for any specific software system.

O7. Sound Evaluation. How can we properly evaluate the true capabilities and limits of ML-based security tooling to discover or mitigate security flaws in important software systems? How can we establish a level playing ground for a sound empirical comparison to other security analysis approaches, like static analysis? In ML-based vulnerability detection, some papers report a performance on public benchmarks that greatly exceeds all expectations, given the general experience in the industry with other security tools (e.g., lt. 6% false positives; 7% false negatives [125]). Yet, they cannot distinguish between vulnerable and patched functions [163]. Going forward, we need to study the reasons for those outstanding results, identify concrete benchmarking pitfalls, and develop reliable methodologies to evaluate ML techniques for security analysis that systematically exclude alternative explanations for the observed performance.

3.2 Security of Evolving Software

Software systems have always been in constant change, but their evolution is due to further acceleration as more tools are being integrated into the development process and AI-based systems are becoming capable of repairing code and contributing new features. As a result, program analysis tools will need to become more agile in the way they are deployed, focusing on analyzing the recently introduced code changes rather than on the overall system.

O8. Program Analysis for Fast-Evolving Software. How do we design program analysis techniques that keep up with the high evolution of modern software systems? Preliminary techniques have already been proposed in the literature [35, 130], but they still lag behind their whole-program counterparts. An important challenge is that such techniques have to be fast, so as to not slow down software development; for instance, while whole-program fuzzing campaigns are often expected to take 24 hours [112], incremental ones should take on the order of minutes [114]. How can incremental updates [23] be used to speed up static analysis? Program analysis techniques should take advantage of the runs performed on earlier versions [41, 196], as well as use the behavior of the previous version as an implicit oracle [144].

3.3 Supply Chain Security

Yesterday's software systems were developed and analyzed as monoliths that were often entirely developed by a single vendor. Today's software systems are (recursively) composed of many third-party components. *This is the software supply chain.* The security posture of the projects behind these third-party components may drastically vary. Some projects may have a vulnerability disclosure policy and track unique identifiers (**common vulnerabilities and exposure (CVEs)**) for the software vulnerabilities that were present across different versions of the project. Others may not be actively maintained anymore and are riddled with known vulnerabilities that are so automatically included in the complete software system.

Tomorrow's software systems will contain new vulnerabilities that could be introduced into any part of the supply chain at any time, e.g., by reusing code snippets from Q&A forums or source code forges, that include both known and unknown vulnerabilities. Some projects, like the Linux kernel,[1] may have a liberal CVE-assignment policy while others never request CVEs for their security flaws which renders the number of CVEs in the supply chain of a software system an unreliable measure of its security posture. Moreover, with the increasing recent adoption of LLMs for the generation of code, security issues hidden in third-party artifacts could be reused and imported in much more flexible ways, which also challenges the existing infrastructure of supply chain security.

3.3.1 *Software Composition Analysis (SCA).*

A SCA [104, 141] automatically identifies the third-party components of a software system to detect known vulnerabilities, license compliance issues, or other risks or quality issues arising from the usage of those third-party components. Existing work on SCA has focused on code clone detection either in the system's source code [88, 102] or the distributed binary [123, 203].

O9. SCA in the Era of ML-Generated Code. How can we reliably identify code snippets that are copied from a code repository that is license-protected or that is potentially riddled with security flaws? If the adoption of AI-assisted programming tools continues at the current rate, an increasing proportion of code added to our code bases will be ML-generated. As these tools pull from a vast array of sources to generate or suggest code snippets, accurately tracking and analyzing the security

[1] "[..] the CVE assignment team is overly cautious and assigns CVE numbers to any bugfix that they identify"; https://docs.kernel.org/process/cve.html.

of these piecemeal components becomes daunting [24, 151, 165]. This fragmentation complicates dependency and vulnerability mapping in SCA, making it difficult to ensure comprehensive coverage in vulnerability scans. Furthermore, the fragmented code reuse also introduces significant challenges for the identification of plagiarism of existing artifacts [124, 177, 200], which further complicates the copyright and license detection for SCA. To tackle these challenges, we require new SCA techniques at the level of code snippets, like software genes [193], for improved supply chain security assurance.

O10. Risk Analysis. How can we evaluate the impact of security flaws in third-party components on the host software system with reasonable accuracy and scalability? Even if SCA tools reliably identify the third-party components with known security flaws (i.e., n-day vulnerabilities), it is left to the user to confirm whether such security flaws or their interactions yield a vulnerability of the host system. A variety of static analysis techniques have been developed to automate this process and identify specific vulnerable execution paths [137, 194, 204] or to determine the host's general security posture w.r.t. the exploitation of n-day vulnerabilities [67, 78]. However, there is a notable lack of dynamic verification techniques to generate a witness host execution to confirm a vulnerability, and existing static analysis techniques are often either not scalable or too imprecise.

O11. Tool Support across the Software Development Life Cycle (SDLC). How can we support the automation of supply chain security analysis across the different phases of the SDLC of a project and its dependencies? Similar to the supply chain in manufacturing, the software supply chain consists of people, processes, tools, third-party components, and other artifacts that play a role in the development and maintenance of a system [1]. Throughout a project SDLC, we can identify several threats to supply chain security. During *development*, developers could submit code containing security flaws [90, 94] or add vulnerable third-party dependencies [147] either accidentally or via compromised accounts or malicious insiders [89]. During *build and deployment*, tools such as the compiler or build server may be compromised and inject security flaws [46, 180]. During *distribution*, **package managers (PMs)** and other registries could distribute malicious third-party components, e.g., via typosquatting [179], compromised maintainer accounts [201], exploits of dependency resolution mechanisms [92], or the persistence of outdated dependencies [99, 204]. During *maintenance*, the security posture of a project may degrade over time when maintainers move on to other projects or simply do not have the time anymore. This leaves downstream users exposed to known security flaws until they are eventually fixed, if ever [175]. All of these challenges require new techniques to support the security of a software system across the entire SDLC.

3.3.2 Supply Chain Ecosystem. Third-party components of a software system are often distributed via PMs and reused within an ecosystem, constituting a network of dependencies. Examples of PMs and the corresponding ecosystems are NPM/JavaScript [127], Maven/Java [104], PIP/Python [187], GoLang [99], and Android [203]. Apart from third-party libraries, pre-training models, datasets, and cloud services are also critical components in new emerging systems, such as LLM-based systems, distributed systems, cloud platforms, and **cyber-physical systems (CPS)**. These new types of dependencies also become potential entry point for attacks.

O12. Longitudinal Studies of Ecosystem Security Health. How can we detect long-term or emerging security threats in a swiftly evolving network of software dependencies involving multiple component versions? How does the overall security health of an ecosystem evolve over time? Existing studies of such ecosystems have focused on the identification of security threats [212] and the propagation of vulnerabilities [127]. As these ecosystems evolve and new ecosystems emerge, in the future similar studies should be repeated in regular intervals and summarized longitudinally.

We should develop the capabilities to identify and track various types of attacks on an ecosystem and to monitor how the security posture of individual projects impacts the overall security health of the entire dependency network.

O13. Ecosystem-Wide Vulnerability Remediation. How can we improve the security of a software system that depends on third-party components with known vulnerabilities? How can we improve the overall security health of an ecosystem? Some projects in an ecosystem might reach their end-of-life, for one reason or another [100], but even if every project was very well maintained where security flaws are fixed as soon as they are found, there is still technical lag in the propagation of these fixes to its dependants [91]. Some hosts (i.e., dependants) might be reluctant to update their otherwise trustworthy dependencies and still use an older version [64]. Hence, vulnerable versions should be identified [31], vulnerability patches should be back-ported to these versions [71], and affected dependants should be identified and updated [54]. To minimize the risks of breaking updates, multi-version execution techniques can be used to roll back any failing updates [98, 156]. To minimize the reliance on (vulnerability-inducing) third-party components, developers can use debloating techniques to trim redundant dependencies from a software system [39, 171]. If the system depends on a third-party component with known vulnerabilities, existing SCA tools would suggest adopting the corresponding patch [164], to update the third-party component (where the vulnerability is fixed) [188], or to migrate to a different component that implements the required functionality [97].

However, most existing remediation strategies are vulnerability-specific with little adjustment to the host where the remediation is applied. In other words, remediation might lead to potentially breaking changes in the host itself [206] or the host's dependants [205]. Moreover, beyond remediation for individual dependants, we should develop ecosystem-wide intervention strategies to maximize the overall health of the ecosystem in the presence of known vulnerabilities or emerging security threats.

O14. The Software Supply Chain of Emerging Systems. How can we identify and properly manage new types of dependencies in emerging systems? How can we systematically identify the new attack surfaces alongside? How can we propose comprehensive detection and management solutions to mitigate these newly emerging threats? The software supply chain of emerging systems faces significant challenges due to its complex, multi-layered nature. Managing dependencies, particularly with pre-trained models, and cloud services, beyond third-party libraries, is increasingly difficult, often resulting in opaque, hard-to-track components. Security vulnerabilities from external services [145, 185] becomes a potential entry point for attacks, especially in distributed and cloud architectures. The rapid and untraceable updates [87, 199] of external services could introduce potential unreliable functionalities, which should be further included into the management of supply chain. Ensuring transparency and traceability across these diverse systems, particularly in ML models and LLMs, is another critical challenge. Pre-training models [79, 93, 106, 198, 209], training frameworks [95, 191, 210], as well as poisoned datasets [36, 60, 148], are also vital components in supply chain, proper detection and management of corresponding threats should also be concerned. Additionally, CPS, such as IoT systems, introduce heightened risks due to their integration of software with safety-critical physical components [131], their supply chains are intricate systems that involve the production and integration of not only software but also hardware components [32, 129], their compliance with safety-critical standards for industries, such as healthcare [51, 110, 128], transportation [207], and manufacturing [83], should also further managed.

3.3.3 *Vulnerability Data Quality.*

The performance of security analysis tools that are developed for the software supply chain rests on the quality of the vulnerability data that is available.

O15. Vulnerability Provenance Metadata. How can we uniquely identify and track a vulnerability across the entire history[2] of an ecosystem? How can we do so in bytecode or even machine code? How can we store, update, and access relevant metadata for each vulnerability in a standardized, machine-readable format? What are the legal and ethical considerations for the storage and availability of such potentially sensitive data? Currently, a vulnerability is assigned a unique identifier, called CVE by a CVE Numbering Authority and stored in vulnerability databases. The first vulnerability database was the Repaired Security Bugs in Multics project published on February 7, 1973. Major vulnerability databases such as the ISS X-Force database, Symantec/SecurityFocus BID database, the Open Source Vulnerability Database, and the National Vulnerability Database aggregate a broad range of publicly disclosed vulnerabilities, including CVEs. Many SCA tools use these databases to identify known vulnerabilities in third-party components. Therefore, vulnerability databases must be updated regularly to ensure the maximum effectiveness of these tools.

However, there does not exist a standardized, machine-readable format for the associated metadata, such as the vulnerable versions, the patch, or the proof-of-vulnerability. We should develop the mechanisms needed to track a vulnerability through the ecosystem, and to swiftly update the metadata as remediation proceeds and new facts emerge. The metadata should be comprehensive, current, and trustworthy. While we believe in responsible disclosure in favor of the dependants, we should investigate the legal or ethical consequences of tracking such potentially sensitive vulnerability information at the ecosystem scale.

O16. SCA Tool Benchmarking. How do we soundly evaluate the capabilities of SCA tools [68]? What about domains where closed-source components play an important role, such as automotive, IoT, or blockchain? The scope of third-party artifacts included in the SCA feature databases could also heavily influence SCA capabilities [105]. Though many experimental SCA tools are proposed to improve accuracy, they are mostly only validated on a limited feature dataset, i.e., by filtering open-source projects by metrics such as stars [66]. This is also why many experimental SCA tools reach high accuracy but seem less satisfactory in real-world scenarios. Moreover, it is also difficult to collect a high-quality feature dataset for specific domains, such as automotive [96], IoT [135], and blockchain [176], where closed-source artifacts play a much more important role and could compromise SCA detection if no corresponding datasets are well established.

3.3.4 *OSS Supply Chain Governance.*

In response to the escalating software supply chain security threats, the community and open-source ecosystem have rallied to implement various countermeasures aimed at mitigating risks. However, there are many regulatory and sociotechnical challenges. Addressing these challenges demands a collaborative effort to simplify security practices, share resources, and foster a culture of security awareness, ensuring that security enhancements do not impede the innovation and agility inherent to the open-source community.

O17. Regulatory Challenges. To what extent **software bills of material (SBOMs)** are used and support cybersecurity? Various efforts from governments have been made to enhance the cybersecurity of software products by delineating regulations on adopting the SBOM for software products, such as the guidelines for security of the IoT from ENISA [77], the proposal of cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020 by the European Commission [76], and the United States Federal Government, per President Biden's Executive Order

[2]Different hosts of a library often depend on different versions of that library. Some hosts might be late or reluctant to update their dependencies.

14028 [108]. The executive order mandated NTIA[3] to publish the minimum elements of SBOMs, including the standards to be used to produce and consume SBOMs. To this end, Software Package Data eXchange [13], CycloneDX [9], and SoftWare IDentification tags [5] have been gradually proposed to describe a list of "ingredients" that make up software components and related threats, such as vulnerabilities and license information. Corresponding SBOM generation tools are also integrated into security auditing tools.

However, the preliminary results in the literature suggest that the adoption of SBOMs by open-source projects is still low, even if there is an increasing trend probably due to the growing interest and pressure from major players [138]. Moreover, the output of such tools also suffers from several limitations, such as representing dependencies at the level of components or libraries identified from dependency configuration files rather than at the level of code snippets (which could be manually copied from Stack Overflow or ML-generated from license-protected source code repositories).

O18. Sociotechnical Challenges. What will be the standards, best practices, tools, and guidelines for secure software development? In addition to SBOMs, industry consortia, like the **open source security Linux foundation (OpenSSF)**, have created standards, best practices, and tooling to enhance ecosystem-wide software supply chain security. For instance, the OpenSSF developed the Criticality Score [6] for trustworthy library reuse, a set of best practices for PMs [7], guidelines for user dependency management [2], and the ScoreCard project [8] for security threat evaluation. There are tools like OWASP [10] and Dependabot [3] to solicit dependency updates, Sigstore [4] for the verification and provenance of third-party components, SLSA [12], SPIFFE [14] and SSDF [11] as guidelines and infrastructure for secure software development and identity verification.

However, open-source projects are still reluctant to adopt these tools and guidelines, due to resource constraints, the complexity and usability of the tools, interoperability issues across diverse systems, and the need to keep pace with an evolving threat landscape. Additionally, striking a balance between rigorous security measures and maintaining development productivity poses a significant challenge.

3.4 Beyond Memory Safety

Today's most critical security flaws are due to violations of memory safety. For instance, 78% of confirmed exploited "in-the-wild" vulnerabilities on Android devices [174] and 70% of vulnerabilities in Google Chrome [160] are violations of memory safety. However, we see memory safety increasingly addressed at the programming-language level. For instance, when the Android team adopted the Rust programming language for new code, the proportion of memory safety-based vulnerabilities dropped from 76% down to 35% [174]. As memory safety vulnerabilities decrease in abundance and our security tools become more effective, attackers will focus on other types of vulnerabilities to exploit.

O19. Emerging Vulnerability Types. In the absence of memory safety issues, which other types of software vulnerabilities exist and how can they be mitigated? How can we operationalize and detect violations of privacy or the General Data Protection Regulation? Memory safety issues are conceptually easy to detect. Other types of vulnerabilities, such as injection attacks (e.g., command/code injection or deserialization attacks), data races (e.g., time-of-check/time-of-use), or information leaks (e.g., side channels/information flow) require much more threat modeling from the security practitioner's point of view. In the future, we should support this modeling process. More generally, we should design practical methods for specifying software properties [47, 133].

[3]The United States Department of Commerce and National Telecommunications and Information Administration is an Executive Branch agency of the United States Department of Commerce that serves as the President's principal adviser on telecommunications and information policy issues.

When memory safety is solved, we expect that offensive security will move on to the next low-hanging fruit. From an economical perspective, an attacker wants to maximize the likelihood of success with the least possible effort. We should empirically monitor this shift and develop the corresponding mitigations in time.

O20. Input Validation and Sanitization. How can we make sure that adversarial inputs do not compromise the security of a software system? If we assume that any public input to a software system is "tainted" and can be controlled by a malicious user, it is crucial to ensure that such input is properly validated and sanitized. For example, a well-known class of attacks is command injection, where a malicious user injects commands in a public input, and when this input propagates through the software, some component of the software system can unintentionally execute the injected command, resulting in loss of data or leakage of secret information. To track how much tainted data propagates through the software and to detect potentially vulnerable program points, static and dynamic program analysis techniques have been developed [58, 181]. In order to prevent the propagation of malicious inputs, it is crucial to validate (i.e., check if the input matches the expected format) and sanitize (i.e., transform the input to the expected format) the user input [18, 30].

However, the additional validation- and sanitization-related code also increases the attack surface and might introduce security vulnerabilities. As the processed user input is often given as a string, this requires effective string-based program analysis [40, 109]. Another challenge is the fact that input validation and sanitization code is typically distributed in different parts of a software system without a clear specification of the *intended* input validation and sanitization policies. Finally, new types of attacks may require changes to the existing policies and continued modifications to the input validation and sanitization code. Due to these challenges discovering and eliminating errors in input validation and sanitization code will likely continue to be an important area of research in the future.

O21. Sensitive Data Exposure. How to detect, quantify, minimize, and eliminate the leakage of sensitive or secret data, such as customer data or cryptographic keys, due to software or hardware side channels for the next generation of software systems? A sensitive data exposure occurs when (properties of) secret data currently processed can be learned by observing the behavior of the processing software system. The system behavior includes side channels such as the execution time or memory/energy usage. Specifically, non-interference requires that publicly observable properties of program execution are independent of any secret values. However, the binary notion of non-interference is not entirely practical as software systems may be reasonably expected to reveal some amount of information that depends on secret values. For example, the purpose of a password checker is to disclose if the provided input matches the password (which is secret)—violating the non-interference property. Hence, there is increasing interest in **quantitative information flow (QIF)** which asks "how much" rather than "whether" secret information is leaked [169]. The amount of information leaked is quantified using concepts such as channel capacity [183] and Shannon entropy [27, 155].

Going forward, we expect many more types of side channels to be discovered. For every side channel, the various causes of leakage should be systematically identified. For instance, for a timing-based side channel, it is necessary to identify all data-dependent optimizations[4] in the hardware, the software, and the entire software supply chain (including the compiler [38]). In general, we need strategies for defensive programming and automated tooling to measure and minimize any possible information leakage. One interesting direction of research is

[4]Data-dependent optimizations, like speculative execution, might change the (observable) execution time depending on properties of the secret data.

to develop automated attack synthesis techniques for both assessing the criticality of the vulnerability and also informing the mitigation strategies, where the length of the synthesized attack is inversely correlated with the severity of the information leakage [155]. We expect that both noninterference analysis and QIF analysis will continue to be important areas of research to develop techniques that identify, quantify, and eliminate information leaks due to side channels.

3.5 Beyond Monolithic, Fully Virtual Software Systems

Research on software security analysis has traditionally focused on monolithic programs that run on laptops, desktops, or servers. However, as our software systems become more decentralized and integrated more deeply with the physical world, we must develop new software security analysis approaches that work across large distributed systems and on proprietary hardware, when resources such as energy, time, or computing are scarce, and to avoid physical harm.

O22. Security of Distributed Systems. How can we analyze the security of software systems (e.g., IoT, Software-as-a-Service, distributed algorithm implementations, blockchain systems, smart contracts), which are distributed across many machines and devices and often dynamically composed at runtime (e.g., as the workload changes, as components are updated, or as machines or devices become (un)available)? How can we minimize data exposure and information leaks for software components that interact via the internet? While we believe that the following challenges apply, at least in part, to all these systems, we pick IoT as a concrete example. The IoT revolution has been enabled by affordable and reusable embedded and cloud software platforms that integrate solutions for various design challenges.

Supply Chain Security. Practically all systems use additional third-party components (for IoT, these are peripheral drivers, board support packages, and application-specific libraries). These third-party components can represent 90–99% of the code base. Yet, keeping up to date with issues and security releases across a diverse collection of externally maintained components is notoriously difficult. Hence, IoT supply chains are intricate systems that involve the production and integration of both hardware and software components, as well as the establishment of trust between different parties. The supply chain of the entire software system may not be considered as localized only within individual devices but rather as spanning across a larger system of systems, i.e., edge, fog, and cloud. The dynamic nature of these systems brings additional challenges to the supply chain security analysis. Different devices may run different (versions of) third-party components. Going forward, improving supply chain security is critical and standardized, shareable SBOM or Hardware Bills of Materials are a major advance toward that goal as they bring transparency to an otherwise opaque supply chain.

Security Analysis. From an analysis perspective, distributed systems are particularly challenging for several reasons. First, distributed systems may change dynamically. Computing nodes of different kinds with different versions may be added or removed elastically at any time. It is difficult to simulate this flexibility in a static analysis or in a lab/testing environment. Second, there is often no central agent that can be analyzed or tested. Third, there are domain-specific types of vulnerabilities, like the Byzantine generals problem where decentralized parties need to arrive at consensus without relying on a trusted central party. Some security analysis questions center around the trustworthiness of the individual nodes in a distributed system. These analysis challenges require specialized approaches.

O23. Data Segregation in Cloud Systems. How do we ensure the privacy of the immense amount of sensitive data stored in cloud systems? Software systems that have high and dynamic demands on computational resources often run on compute clouds using platforms such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform. Protecting the integrity and confidentiality of sensitive user data stored in these clouds is a critical problem now and will continue to be a critical problem in the future. Today, *access control rules* [50, 103, 161, 195] are explicitly and often manually specified to identify exactly who has access to what part of data, while denying unauthorized accesses. However, errors in access control policy specifications can result in the exposure of millions of customers' private data to the public [17, 25, 73, 184]. To check the correctness of access control policies, formal verification techniques have been applied in the past [80, 101]; and in recent years they have shown promising results in practical applications [26, 75].

We find that access control verification is an area where the scalability of formal verification techniques seem to be sufficient for practical use. However, many challenges remain. The capabilities of existing constraint solvers should be extended to handle the types of constraints encountered in this domain. The usability of the analysis tools should be improved by providing easy ways to specify correctness properties (such as relying on differential analysis rather than asking users to write assertions). Fully automated analysis should be achieved without generating any inconclusive results (which may require hybrid approaches that combine formal reasoning with automated testing in cases where formal reasoning is unable to prove or disprove correctness).

O24. CPS. How do we ensure the security and safety of software systems that physically interact with an evolving and incompletely perceived environment, where insecurity can cause physical damage and potentially loss of life (e.g., autonomous vehicles, IoT devices, robots, or virtual reality headsets)? CPSs are difficult analyze both physically and virtually. When testing a CPS *physically* (i.e., in the actual environment), the testing procedure ma may be relatively too slow and the CPS may break or get damaged. When analyzing a CPS *virtually* using static analysis (SAST) or simulation, the CPS cannot be analyzed without certain assumptions about the environment. These assumptions may or may not be true in a physical setup.

3.6 Resilient Computing

Widespread successful attacks on IT systems and even critical infrastructures have repeatedly proven that current common security measures are insufficient. System developers and maintainers commonly try to secure their systems by identifying, triaging, and fixing vulnerabilities. A system is commonly deemed secure when it is free of *known* vulnerabilities—but what about the ones we do not know yet? Those vulnerabilities pose a strong technical debt that future attackers will happily exploit.

The only useful paradigm is, therefore, to "assume breach." We must design systems with the mindset that they are and will be vulnerable and, nonetheless, should be able to successfully withstand at least certain classes of common attacks, they must become attack resilient. The guiding principle toward attack resilience, defense in depth, has long been known, yet is at the core of too few systems we build today. Most software architectures still assume a single protection layer, which, if penetrated, puts the entire software system at risk of failure. But how to best implement multi-layered defenses?

O25. Risk Analysis. How can we perform effective risk analysis already at the design level to judge a system's attack resilience? An analysis of attack resilience must incorporate different and complementary possible types of mitigations, ranging from process-based mitigations, such as a four-eye principle or the use of validation tools during coding, to application-level measures, such as proper password hashing to platform-level measures, such as proper, risk-centric compartmentalization of individual subsystems.

O26. Risk-Centric Compartmentalization requires research in systems security: Currently, we can isolate subsystems essentially only by turning them into separate OS processes, which comes with huge computational and maintenance overhead. But new technologies are on the horizon. WebAssembly, for instance, allows one to sandbox untrusted parts of applications with relatively low overhead. Existing compilers allow one to create WebAssembly for C/C++ components. Other opportunities to isolate computations within applications may be yielded by using hardware support such as Intel's Memory Protection Keys, ARM Memory Tagging and CHERI [190]. These are relatively low-level features, yet researchers can test their efficacy as security barriers when properly integrated into programming languages and environments. Further, GraalVM, in a feature still under development, now supports the proper isolation of code executing within the same OS process, currently in polyglot but soon also in single-language settings.

3.7 Emergent Behaviors and Vulnerability Composition

Computer security research is sometimes seen as an arms race where the advances in defensive techniques that eliminate vulnerabilities lead to advancements in offensive techniques that result in new types of attacks, which then inspire new defensive strategies, resulting in a continuous cycle of security measures and counter-measures. In this race, the defensive techniques must be in the lead to succeed in securing computer systems. An increasingly critical set of offensive techniques relies on emergent behaviors and compositions of vulnerabilities to create exploit chains that achieve the adversarial goals of the attacker.

The concept of "weird machines" characterizes exploit generation as a task of programming a weird machine, where the instructions of the weird machine are vulnerabilities or unintended behaviors that take the system to an unintended "weird" state [37, 74]. In this characterization, exploit chaining corresponds to writing a weird machine program that achieves the attacker's goal. Defending against this type of sophisticated attack strategy requires a defensive strategy that not only detects emergent behaviors but also analyzes their compositions.

O27. Identifying Emergent Behaviors across Layers of Abstraction. How can we identify emergent behaviors that cut across different layers of software, for example, emergent behaviors that involve a combination of vulnerabilities in the firmware, OS, and application code? Software is built at different layers, where a layer exposes an interface providing an abstraction to the next layer. However, in many cases, these interfaces lead to unintended behaviors due to misuse, under-specification, or ambiguous specification of the interface constraints. Although analysis of software interfaces has been investigated in the past, novel techniques that can identify unintended/emergent behaviors across multiple layers of abstraction that involve multiple interfaces are needed to assess the security of the whole software system. This will require innovative analysis techniques that can track the behavior of the system across multiple layers of abstraction.

O28. Discovering and Defending against Exploit Chains. How can we analyze collections of vulnerable or unintended behaviors to prevent attacks that chain multiple vulnerabilities? An unintended behavior in a computer system may not be a security vulnerability by itself. However, an attacker who is aware of a set of unintended behaviors can combine them to generate a programmable collection of adversarial operations (which correspond to instructions of a weird machine) and then write programs that chain these operations. In order to defend against this type of attacks, we have to first discover them. Exploit chains can be seen as a composition of emergent/unintended behaviors. Techniques that can automatically compose emergent/unintended behaviors can lead to techniques that can automatically search for exploit chains. Once the exploit chains are discovered, mitigation strategies can be developed to identify the most effective ways to eliminate them.

3.8 Security Education

O29. Security as a Mandatory Course. There is a gap between the need for personnel skilled in software security and the availability of that personnel [136]. Computer science students still graduate with scarce or no secure programming knowledge [20]. It becomes necessary to provide adequate security training to the next generation of developers. This will be possible by (i) improving students' engagement in producing secure software (traditional, embedded systems, CPS, and IoT systems), (ii) helping students acquire security skills and knowledge on the use of development methods for secure development, (iii) training students on **dynamic analysis security tools (DAST)** and SASTs, and (iv) educating students on the perils related to the use of AI-generated code and code snippets from Questions and Answers forums. In continuous education, the education of "security champions" within companies has shown great successes. How can this be leveraged on a global scale?

O30. Integrate Security Tool Usage into Software Engineering Courses. While introducing some security concerns in the source code is unavoidable (e.g., because a certain vulnerability is still unknown), others can be fixed before the delivery of software systems (or even developers could avoid their introduction). Educators should train the next generation of developers so that they identify known security concerns and then fix them before the delivery of software systems (e.g., [139, 140]). This could be achieved by training students on DAST and SAST that are well known in both the academy and industry. Specifically, DAST analyze software systems at run-time, while SAST examine the code of software systems without executing them. There are three scenarios in which developers can leverage SAST: (i) while developing code, by highlighting the presence of security concerns directly in IDEs; (ii) within a CI pipeline, which could make a build fail if the code is not compliant with given security rules (e.g., the committed code must not contain critical vulnerabilities); and (iii) during a code review. A DAST could be used in the last two scenarios together with a SAST.

3.9 Law and Policy Making

O31. Law and policy making. Apart from the sociotechnical challenges, how can we ensure that our critical software infrastructure is robust against cyberattacks, we need effective policies and regulation? How can we balance legal and technical aspects for critical as well as non-critical systems, to both empower users and secure the interactions? Should the vulnerability disclosure process or supply chain security practice be regulated, and if so how? While various industries where software plays a key role are already strictly regulated (such as safety-critical domains like aerospace), software products themselves lack such legal safeguards. We have previously discussed regulations for the software supply chain in some domains like the IoT, including the need to accompany a software product with a SBOM [76, 77, 108]. However, much more work is needed with legal scholars, economics researchers, policy makers, and software vendors to create an effective legal framework and incentive system. It is important for software researchers and practitioners to engage in shaping the legal framework to ensure that appropriate cybersecurity practices are applied and lessons are learned after each cybersecurity incident.

4 Perspective

Over the last several decades, software engineering has mainly focused on the *design and implementation* of the software systems that underpin our digital world, including our digital economy, our digital entertainment and social networks, as well as our apps and devices that we use every day. Going forward, we expect, the focus will slowly move to the *evolution, maintenance, and* *security-hardening* of these systems. Once the software supply chain is set up during design and implementation, even if the software system itself is properly maintained, it is important also to monitor and maintain the integrity of its supply chain. This will require *ecosystem-scale analyses* of large networks of dependencies.

Software systems will compose more dynamically and work in concert with many other systems that are in many cases unknown at development time. Such systems of systems will be *highly interconnected and distributed* to an arbitrary complexity. Individual nodes may be elastically added or removed at any time depending on the current user demand or compute availability. The dynamic organization of computation makes it difficult to analyze the security of such systems *a priori*. The distributed nature of computation poses challenges for the confidentiality and integrity of the data that is being processed. Highly scalable, incremental, dynamic, distributed security analysis methods for the deployed software system will be needed that work under uncertainty.

Software systems will change more rapidly. Many software systems are built and maintained with a CI/continuous deployment pipeline enabled which allows for a rapid testing and deployment of new features, bug fixes, or security patches. With the advent of effective tools for (semi-)automated programming [208] and vulnerability remediation, we expect the rate of change to increase further. Submitted changes, bug fixes, and security patches may be produced or reviewed entirely mechanically [81]. This increasing level of automation in the development process poses interesting opportunities for future work in automatic software security analysis.

Software systems will be more heterogeneous and potentially integrating ML components. Some components may be written in a memory-safe language while others are not. Some components may be implemented in a type-safe language while another may not. Some components may be realized as ML model that returns approximate, probabilistic results while the remainder of the system may be entirely deterministic (modulo a thread schedule). An effective software security analysis tool should be able to deal with such heterogeneity.

Software systems will be embedded further into our physical world. Virtual reality systems will project virtual objects into our physical space for us to interact with. Embodied AI will turn virtual chat bots into embodied robot assistants that interact with the real world to test their hypotheses about the real world. More devices will be connected to the internet (via fog and cloud computing). Cars receive software updates over the air. Transport is increasingly automated. While rich with opportunities, from a safety and security perspective, the increasing ability to interact with the physical world also poses increased risks of physical damage up to loss of life.

All of these factors point to interesting new challenges and opportunities for the software engineering community toward the security analysis of the software systems of the future. We are excited about these prospects and look forward to reconciling our predictions today with our reflections 5 years (or five decades) from now.

References

[1] Synopsis. What is software supply chain security and how does it Work?—synopsys. Retrieved March 04, 2024 from https://www.synopsys.com/glossary/what-is-software-supply-chain-security.html

[2] Google. 2024. Best practices for dependency management—Google Cloud blog. Retrieved March 14, 2024 from https://cloud.google.com/blog/topics/developers-practitioners/best-practices-dependency-management

[3] Github. 2024. Dependabot. Retrieved March 14, 2024 from https://github.com/dependabot

[4] Sigstore. 2024. Home. Sigstore. Retrieved March 14, 2024 from https://www.sigstore.dev/

[5] NIST. 2024. Nvd - Swid. Retrieved March 15, 2024 from https://nvd.nist.gov/products/swid

[6] OpenSSF. 2024. Ossf/criticality_score: Gives criticality score for an open source project. Retrieved March 14, 2024 from https://github.com/ossf/criticality_score

[7] OpenSSF. 2024. Ossf/package-manager-best-practices: Collection of security best practices for package managers. Retrieved March 14, 2024 from https://github.com/ossf/package-manager-best-practices/tree/main

[8] OpenSSF. 2024. Ossf/scorecard: OpenSSF scorecard - security health metrics for open source. Retrieved March 14, 2024 from https://github.com/ossf/scorecard

[9] OWASP. 2024. OWASP CycloneDX software bill of materials (SBOM) standard. Retrieved March 15, 2024 from https://cyclonedx.org/

[10] OWASP. 2024. OWASP dependency-Check—OWASP foundation. Retrieved March 14, 2024 from https://owasp.org/www-project-dependency-check/

[11] NIST. 2024. Secure software development framework—CSRC. Retrieved March 14, 2024 from https://csrc.nist.gov/projects/ssdf

[12] Linux Foundation. 2024. SLSA supply-chain levels for software artifacts. Retrieved March 14, 2024 from https://slsa.dev/

[13] Linux Foundation. 2024. SPDX – Linux foundation projects site. Retrieved March 15, 2024 from https://spdx.dev/

[14] Cloud Native Computing Foundation. 2024. SPIFFE – secure production identity framework for everyone. Retrieved March 14, 2024 from https://spiffe.io/

[15] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1 (2009), 1–40.

[16] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP 2008)*. IEEE, 263–277.

[17] aleak. 2017. Another misconfigured Amazon S3 server leaks data of 50,000 Australians. Retrieved from https://www.scmagazineuk.com/another-misconfigured-amazon-s3-server-leaks-data-of-50000-australians/article/705125/

[18] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. 2014. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 225–236.

[19] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical assessment of machine learning-based malware detectors for Android: Measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering* 21 (2016), 183–211.

[20] Majed Almansoori, Jessica Lam, Elias Fang, Kieran Mulligan, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. 2020. How secure are our computer systems courses?. In *Proceedings of the Conference on International Computing Education Research.* ACM, 271–281.

[21] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-throughput fuzzing of network applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22).*

[22] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *Proceedings of the USENIX Security Symposium.*

[23] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, 288–298.

[24] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.

[25] azureflaw. 2021. Microsoft Azure Cloud vulnerability is the 'Worst you Can imagine'. Retrieved from https://www.theverge.com/2021/8/27/22644161/microsoft-azure-database-vulnerabilty-chaosdb?fbclid=IwAR2nKV8uslH4EGDslnogYT4ulQRGz7NsD0xuIb3lgK2sP1-WG_O1tJbR-eE

[26] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachu, and Carsten Varming. 2018. Semantic-based automated reasoning for AWS access policies using SMT. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD '18)*, 1–9.

[27] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S & P 2009)*, 141–153.

[28] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. 2023. Modern code reviews–survey of literature and practice. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–61.

[29] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys* 51, 3 (2018), 1–39.

[30] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08).* IEEE, 387–401.

[31] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*, 2352–2364.

[32] Mohamed Ben-Daya, Elkafi Hassini, and Zied Bahroun. 2019. Internet of things and supply chain management: A literature review. *International Journal of Production Research* 57, 15–16 (2019), 4719–4742.

[33] Marcel Böhme. 2022. Statistical reasoning about programs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 5 pages. DOI: https://doi.org/10.1145/3510455.3512796

[34] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2021), 79–86. DOI: https://doi.org/10.1109/MS.2020.3016773

[35] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2329–2344.

[36] Dillon Bowen, Brendan Murphy, Will Cai, David Khachaturov, Adam Gleave, and Kellin Pelrine. 2024. Scaling laws for data poisoning in LLMs. arXiv:2408.02946. Retrieved from https://arxiv.org/abs/2408.02946

[37] Sergey Bratus, Michael Locasto, Meredith Patterson, Len Sassaman, and Anna Shubina. 2011. From buffer overflows to weird machines and and theory of computation. USENIX*;login*, 13–21.

[38] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT leaks: Inducing timing side channels through just-in-time compilation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE, 1207–1222.

[39] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. Jshrink: In-depth investigation Into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 135–146.

[40] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2017. *String Analysis for Software Verification and Security*. Springer.

[41] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running symbolic execution forever. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, 63–74.

[42] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, 209–224.

[43] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '06)*, 322–335.

[44] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, 1066–1071.

[45] Cristian Cadar and Martin Nowack. 2020. KLEE symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer* 23, 6 (2020), 867–870. DOI: https://doi.org/10.1007/s10009-020-00570-3

[46] Cristian Cadar, Luís Pina, and John Regehr. 2015. Multi-version execution defeats a compiler-bug-based backdoor. Retrieved from https://ccadar.blogspot.co.uk/2015/11/multi-version-execution-defeats.html

[47] Cristian Cadar, Daniel Schemmel, and Arindam Sharma. 2023. Patch specifications via product programs. In *Proceedings of the 2023 International Conference on Formal Methods in Software Engineering (FormaliSE '23)*, 39–43. DOI: https://doi.org/10.1109/FormaliSE58978.2023.00012

[48] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Communications of the Association for Computing Machinery* 56, 2 (2013), 82–90.

[49] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

[50] cancan 2015. ryanb/cancan GitHub. Retrieved from https://github.com/ryanb/cancan

[51] Anthony James Cartwright. 2023. The elephant in the room: Cybersecurity in healthcare. *Journal of Clinical Monitoring and Computing* 37, 5 (2023), 1123–1132.

[52] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[53] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 09 (Sept. 2022), 3280–3296. 1939–3520. DOI: https://doi.org/10.1109/TSE.2021.3087402

[54] Yang Chen, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 90–99.

[55] Clang Static Analyzer. [n.d.]. Clang static analyzer. Retrieved from https://clang-analyzer.llvm.org

[56] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* 3 (1976), 215–222.

[57] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[58] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 196–206.

[59] CLion 2024. CLion IDE. Retrieved from https://www.jetbrains.com/clion/

[60] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2024. Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 280–292.

[61] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys* 28, 2 (1996), 324–328.

[62] Coverity Software. [n.d.]. Coverity software. Retrieved from http://www.coverity.com

[63] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, Vol. 98, 63–78.

[64] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. 2007. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *ACM SIGOPS Operating Systems Review* 41, 6 (Oct. 2007), 221–236. DOI: https://doi.org/10.1145/1323293.1294283

[65] 2024. CWE TOP 25 most dangerous software errors. Retrieved from https://www.sans.org/top25-software-errors/

[66] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR '21).* IEEE, 560–564.

[67] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through PoC migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, New York, NY, 3300–3317. DOI: https://doi.org/10.1145/3460120.3484594

[68] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. 2021. Identifying challenges for OSS vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3613–3625.

[69] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, 653–656.

[70] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM* 54, 9 (2011), 69–77.

[71] Alexandre Decan, Tom Mens, Ahmed Zerouali, and Coen De Roover. 2021. Back to the past–analysing backporting practices in package dependency networks. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4087–4099.

[72] Dorothy E. Denning. 1987. An intrusion-detection model. *IEEE Transactions on Software Engineering* 2 (1987), 222–232.

[73] djleak. [n.d.]. Cloud Leak: WSJ parent company Dow Jones exposed customer data. Retrieved from https://www.upguard.com/breaches/cloud-leak-dow-jones

[74] Thomas Dullien. 2011. Weird machines, exploitability, and provable unexploitability. Retrieved from http://www.dullien.net/thomas/weird-machines-exploitability.pdf

[75] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying permissiveness of access control policies. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE '22)*. ACM, 1805–1817.

[76] European Parliament. 2022. *Proposal for a regulation of the European Parliament and of the Council on horizontal cybersecurity requirements for products with digital elements and amending regulation (EU) 2019/1020*. Retrieved from https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A52022PC0454

[77] European Union Agency for Cybersecurity. 2020. Guidelines for securing the Internet of things - ENISA. Retrieved from http://archive.md/2023.04.18-071548/https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things/

[78] Yong Fang, Yongcheng Liu, Cheng Huang, and Liang Liu. 2020. FastEmbed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. *PLOS One* 15, 2 (2020), e0228439.

[79] Shiwei Feng, Guanhong Tao, Siyuan Cheng, Guangyu Shen, Xiangzhe Xu, Yingqi Liu, Kaiyuan Zhang, Shiqing Ma, and Xiangyu Zhang. 2023.Detecting backdoors in pre-trained encoders. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 16352–16362.

[80] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05),* 196–205.

[81] Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, et al. 2024. Resolving code review comments with machine learning. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24).*

[82] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, 608–620.

[83] Dr Bhargav Gangadhara. 2023. Optimizing cloud-based manufacturing: A study on service and development models. *International Journal of Science and Research* 12, 6 (2023), 2487–2491.

[84] Jun Gao, Pingfan Kong, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Negative results on mining crypto-Api usage rules in Android Apps. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR '19).* IEEE, 388–398.

[85] GitHub 2024. GitHub website. Retrieved from https://github.com/

[86] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM* 55, 3 (2012), 40–44.

[87] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–41.

[88] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold token-based code clone detection. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '21).*

[89] Qingyuan Gong, Jiayun Zhang, Yang Chen, Qi Li, Yu Xiao, Xin Wang, and Pan Hui. 2019. Detecting malicious accounts in online developer communities using deep learning. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 1251–1260.

[90] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. 2021. Anomalicious: Automated detection of anomalous and potentially malicious commits on GitHub. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '21)*, 258–267. DOI: https://doi.org/10.1109/ICSE-SEIP52600.2021.00035

[91] Jesus M. Gonzalez-Barahona. 2020. Characterizing outdateness with technical lag: An exploratory study. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 735–741.

[92] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. 2023. Investigating package related security threats in software registries. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP '23).* IEEE, 1578–1595.

[93] Shangwei Guo, Chunlong Xie, Jiwei Li, Lingjuan Lyu, and Tianwei Zhang. 2022. Threats to pre-trained language models: Survey and taxonomy. arXiv:2202.06862. Retrieved from https://arxiv.org/abs/2202.06862

[94] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An empirical study of malicious code in PyPI ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23).* IEEE, 166–177.

[95] Danny Halawi, Alexander Wei, Eric Wallace, Tony T. Wang, Nika Haghtalab, and Jacob Steinhardt. 2024. Covert malicious finetuning: Challenges in safeguarding LLM adaptation. arXiv:2406.20053. Retrieved from https://arxiv.org/abs/2406.20053

[96] Subir Halder, Amrita Ghosal, and Mauro Conti. 2020. Secure over-the-air software updates in connected vehicles: A survey. *Computer Networks* 178 (2020), 107343.

[97] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. 2021. A multi-metric ranking approach for library migration recommendations. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '21).* IEEE, 72–83.

[98] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*, 612–621.

[99] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2023. Empirical analysis of vulnerabilities life cycle in golang ecosystem. arXiv:2401.00515. Retrieved from https://arxiv.org/abs/2401.00515

[100] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.

[101] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *Software Tools for Technology Transfer* 10, 6 (2008), 503–520.

[102] Yu-Liang Hung and Shingo Takada. 2020. CPPCD: A token-based approach to detecting potential clones. In *Proceedings of the IEEE 14th International Workshop on Software Clones (IWSC '20).*

[103] IAM. [n.d.]. AWS IAM policy language. Retrieved from http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html

[104] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21) (ESEM '21).* ACM, New York, NY, Article 5, 11 pages. DOI: https://doi.org/10.1145/3475716.3475769

[105] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23).* ACM, New York, NY, 1383–1395. DOI: https://doi.org/10.1145/3597926.3598143

[106] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R. Schorlemmer, George K. Thiruvathukal, and James C. Davis. 2022. An empirical study of artifacts and security risks in the pre-trained model supply chain. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 105–114.

[107] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. 2021. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR '21)*. IEEE, 432–443. DOI: https://doi.org/10.1109/MSR52588.2021.00055

[108] Joe Biden. 2021. *Executive Order on Improving the Nation's Cybersecurity*. Retrieved from https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[109] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing summaries of string loops in C for better testing and refactoring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19),* 874–888.

[110] Tushar Khinvasara, Stephanie Ness, and Nikolaos Tzenios. 2023. Risk management in medical device industry. *Journal of Engineering Research and Reports* 25, 8 (2023), 130–140.

[111] James C. King. 1976. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (1976), 385–394.

[112] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2123–2138.

[113] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 207–220.

[114] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in CI/CD pipelines. In *Proceedings of the 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT '23),* 25–32. DOI: https://doi.org/10.1109/SBFT59156.2023.00015

[115] Padmanabhan Krishnan, Cristina Cifuentes, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2023. Why is static application security testing hard to learn? *IEEE Security & Privacy* 21, 5 (2023), 68–72.

[116] Tomasz Kuchta and Bartosz Zator. 2022. Auto off-target: Enabling thorough and scalable testing for complex software systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–12.

[117] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2018. Code-pointer integrity. In *the Continuing Arms Race: Code-Reuse Attacks and Defenses*, 81–116.

[118] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *11th USENIX Symposium on Operating System Design and Implementation*, 216–226.

[119] Seongmin Lee and Marcel Böhme. 2023. Statistical reachability analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*, 12. DOI: https://doi.org/10.1145/3611643.3616268

[120] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - a formally verified optimizing compiler. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS '16).*

[121] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A symbolic execution and automatic Test generation tool for C++ programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11).* Springer, 609–615.

[122] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android Apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[123] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. Libd: Scalable and precise third-party library detection in Android markets. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE '17).* IEEE, 335–346.

[124] Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. 2023. Protecting intellectual property of large language model-based code generation apis via watermarks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2336–2350.

[125] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Network and Distributed Systems Security*. The Internet Society.

[126] LibFuzzer 2022. LibFuzzer website. Retrieved from http://llvm.org/docs/LibFuzzer.html

[127] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, 672–684.

[128] Antonio López Martínez, Manuel Gil Pérez, and Antonio Ruiz-Martínez. 2023. A comprehensive review of the state-of-the-art on security and privacy issues in healthcare. *ACM Computing Surveys* 55, 12 (2023), 1–38.

[129] Ethirajan Manavalan and Kandasamy Jayakrishna. 2019. A review of Internet of things (IoT) embedded sustainable supply chain for industry 4.0 requirements. *Computers & Industrial Engineering* 127 (2019), 925–953.

[130] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '13)*. 235–245.

[131] Ankur Maurya and Divya Kumar. 2020. Reliability of safety-critical systems: A state-of-the-art review. *Quality and Reliability Engineering International* 36, 7 (2020), 2547–2568.

[132] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '24)*, 15 pages.

[133] Bertrand Meyer. 1992. Applying 'Design by contract'. *IEEE Computer* 25, 10 (1992), 40–51.

[134] Microsoft. [n.d.]. Copilot. Retrieved March 3, 2024 from https://copilot.microsoft.com/

[135] Imanol Mugarza, Jose Luis Flores, and Jose Luis Montero. 2020. Security issues and software updates management in the industrial internet of things (Iiot) era. *Sensors* 20, 24 (2020), 7160.

[136] Phil Muncaster. 2021. Global security skills shortage falls to 2.7 million workers - Infosecurity Magazine. Retrieved October 12, 2023 from https://www.infosecurity-magazine.com/news/global-security-skills-shortage/

[137] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *Proceedings of the 30th International Symposium on Software Testing and Analysis (ISSTA '21).*

[138] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. 2023. Software bill of materials adoption: A mining study from GitHub. In *Proceedings of International Conference on Software Maintenance and Evolution.* IEEE, 39–49. DOI: https://doi.org/10.1109/ICSME58846.2023.00016

[139] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. 2023. Training for security: Planning the use of a SAT in the development pipeline of web apps. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training (SEET@ICSE '23),* 40–45. DOI: https://doi.org/10.1109/ICSE-SEET58685.2023.00010

[140] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. 2024. Training for security: Results from using a static analysis tool in the development pipeline of web apps. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '24).* ACM, New York, NY, 253–263. DOI: https://doi.org/10.1145/3639474.3640073

[141] Philippe Ombredanne. 2020. Free and open source software license compliance: Tools for software composition analysis. *Computer* 53, 10 (2020), 105–109. DOI: https://doi.org/10.1109/MC.2020.3011082

[142] OSS-Fuzz. 2024. OSS-Fuzz - continuous fuzzing for open source software. Retrieved from https://github.com/google/oss-fuzz

[143] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19),* 398–401. DOI: https://doi.org/10.1145/3293882.3339002

[144] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the International Conference on Software Engineering (ICSE '16),* 1181–1192.

[145] Fatemeh Khoda Parast, Chandni Sindhav, Seema Nikam, Hadiseh Izadi Yekta, Kenneth B. Kent, and Saqib Hakak. 2022. Cloud computing security: A survey of service-based models. *Computers & Security* 114 (2022), 102580.

[146] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 179–180.

[147] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 1513–1531.

[148] Pankayaraj Pathmanathan, Souradip Chakraborty, Xiangyu Liu, Yongyuan Liang, and Furong Huang. 2024. Is poisoning a real threat to LLM alignment? Maybe more so than you think. arXiv:2406.12091. Retrieved from https://arxiv.org/abs/2406.12091

[149] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools Keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 331–341.

[150] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP '22).* IEEE, 754–768.

[151] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of github Copilot'S code contributions. In *2022 IEEE Symposium on Security and Privacy (SP).* IEEE, 754–768.

[152] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23).* ACM, New York, NY, 2785–2799. DOI: https://doi.org/10.1145/3576915.3623157

[153] Jonas Peters, Dominik Janzing, and Bernhard Schlkopf. 2017. *Elements of Causal Inference: Foundations and Learning Algorithms.* The MIT Press.

[154] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2021. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. DOI: https://doi.org/10.1109/TSE.2019.2941681

[155] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of adaptive side-channel attacks. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF '17).* IEEE Computer Society, 328–342.

[156] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19),* 573–585.

[157] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 181–191.

[158] Goran Piskachev, Ranjith Krishnamurthy, and Eric Bodden. 2021. Secucheck: Engineering configurable taint analysis for software developers. In *Proceedings of the 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM '21).* IEEE, 24–29.

[159] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, 181–198.

[160] Chromium Project. 2021. Memory safety. Retrieved from https://www.chromium.org/Home/chromium-security/memory-safety/

[161] pundit. 2016. GitHub - elabs/pundit: Minimal authorization through OO design and pure Ruby classes. Retrieved from https://github.com/elabs/pundit

[162] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20).* USENIX Association, 667–682. Retrieved from https://www.usenix.org/conference/osdi20/presentation/rigger

[163] Niklas Risse and Marcel Böhme. 2023. Limits of machine learning for automatic vulnerability detection. arXiv:2306.17193.

[164] Yaman Roumani. 2021. Patching zero-day vulnerabilities: An empirical analysis. *Journal of Cybersecurity* 7, 1 (2021), tyab023.

[165] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023.Lost at C: A user study on the security implications of large language model code assistants. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23),* 2205–2222.

[166] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[167] Kostya Serebryany. 2017. OSS-fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[168] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12),* 309–318.

[169] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS '09)*, 288–302.

[170] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. 2022. The effects of continuous integration on software development: A systematic literature review. *Empirical Software Engineering* 27, 3 (2022), 78.

[171] X. Song, Y. Wang, X. Cheng, G. Liang, W. Qianxiang, and Z. Zhu. 2024. Efficiently trimming the fat: Streamlining software dependencies with Java reflection and dependency analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24).* IEEE Computer Society, Los Alamitos, CA, 887–887. Retrieved from https://doi.ieeecomputersociety.org/

[172] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[173] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15).* IEEE, 46–55.

[174] Jeffrey Vander Stoep. 2022. Memory safe languages in Android 13. Retrieved from https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[175] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical lag of dependencies in major package managers. In *Proceedings of the 2020 27th Asia-Pacific Software Engineering Conference (APSEC '20).* IEEE, 228–237.

[176] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. 2023. Demystifying the composition and code reuse in solidity smart contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 796–807.

[177] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, 652–660.

[178] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal War in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62.

[179] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *Proceedings of the 14th International Conference Network and System Security (NSS '20).* Springer, 112–131.

[180] Ken Thompson. 1984. Reflections on trusting trust. *Communications of the ACM* 27, 8 (Aug. 1984), 761–763. DOI: https://doi.org/10.1145/358198.358210

[181] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[182] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147.

[183] Celina G. Val, Michael A. Enescu, Sam Bayless, William Aiello, and Alan J. Hu. 2016. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS & P '16)*, 31–46. DOI: https://doi.org/10.1109/EuroSP.2016.15

[184] verizonleak. [n.d.]. 14 million verizon subscribers' details leak from crappily configured AWS S3 data store. Retrieved from https://www.theregister.co.uk/2017/07/12/14m_verizon_customers_details_out/

[185] Garima Verma and Sandhya Adhikari. 2020. Cloud computing security issues: A stakeholder's perspective. *SN Computer Science* 1, 6 (2020), 329.

[186] VSCode. 2024. VSCode IDE. Retrieved from https://code.visualstudio.com/

[187] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the Python ecosystem. In *Proceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops (Euros & PW '20).* IEEE, 509–514.

[188] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023. Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. *IEEE Transactions on Software Engineering* (2023).

[189] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. arXiv:2201.11903. Retrieved from https://arxiv.org/abs/2201.11903

[190] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 457–468.

[191] Fangzhou Wu, Ning Zhang, Somesh Jha, Patrick McDaniel, and Chaowei Xiao. 2024. A new era in llm security: Exploring security concerns in real-world llm-based systems. arXiv:2402.18649. Retrieved from https://arxiv.org/abs/2402.18649

[192] Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. 2020. Parallelization techniques for verifying neural networks. In *Proceedings of the 2020 Formal Methods in Computer Aided Design (FMCAD '20),* 128–137. DOI: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_20

[193] Yueming Wu, Chengwei Liu, and Yang Liu. 2023. The software genome project: Venture to the genomic pathways of open source software and its applications. arXiv:2311.09881. Retrieved from https://arxiv.org/abs/2311.09881

[194] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23).* IEEE, 1046–1058.

[195] XACML. 2003. EXtensible access control markup language (XACML) Version 1.0. OASIS Standard. Retrieved from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacmlhttp://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[196] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12).* ACM, New York, NY, 144–154. DOI: https://doi.org/10.1145/2338965.2336771

[197] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 283–294.

[198] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.

[199] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. 2020.A first look at the deprecation of RESTful APIs: An empirical study. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME '20).* IEEE, 151–161.

[200] Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. 2023. CODEIPPROMPT: Intellectual property infringement assessment of code language models. In *Proceedings of the International Conference on Machine Learning.* PMLR, 40373–40389.

[201] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 331–340.

[202] Michal Zalewski. [n.d.]. Technical "whitepaper" for AFL-fuzz. Retrieved from http://lcamtuf.coredump.cx/afl/technical_details.txt

[203] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE '21).* IEEE, 1695–1707.

[204] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in Maven ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 191–203.

[205] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2023. Has my release disobeyed semantic versioning? Static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22).* ACM, New York, NY, Article 51, 12 pages. DOI: https://doi.org/10.1145/3551349.3556956

[206] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible remediation on vulnerabilities from third-party libraries for Java projects. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23).* IEEE, 2540–2552.

[207] Yanan Zhang, Yuqiao Ning, Chao Ma, Longhai Yu, and Zhen Guo. 2023. Empirical study for open source libraries in automotive software systems. *IEEE Access* (2023).

[208] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. arXiv:2404.05427. Retrieved from https://arxiv.org/abs/2404.05427

[209] Zhengyan Zhang, Guangxuan Xiao, Yongwei Li, Tian Lv, Fanchao Qi, Zhiyuan Liu, Yasheng Wang, Xin Jiang, and Maosong Sun. 2023. Red alarm for pre-trained models: Universal vulnerability to neuron-level backdoor attacks. *Machine Intelligence Research* 20, 2 (2023), 180–193.

[210] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. 2024. Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs. arXiv:2409.09368. Retrieved from https://arxiv.org/abs/2409.09368

[211] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems,* Vol. 32.

[212] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, 995–1010.

Received 27 May 2024; revised 26 September 2024; accepted 4 November 2024