International Journal of Safety and Security Engineering
Vol. 16, No. 2, February, 2026, pp. 327-333
Journal homepage: http://iieta.org/journals/ijsse

Network-Centric Security Awareness in Software Development: Integrating Vulnerability Patterns, Developer Practices, and Organizational Controls

Aladdin Abbas Abdulhassan[1], Abdullah Yousif Lafta[2], Alaa Shawqi Jaber[1*], Mohammed Ibrahim Kareem[3]

[1] Information Network, University of Babylon, Babylon 51002, Iraq
[2] College of Engineering, Al-Nahrain University, Baghdad 10011, Iraq
[3] Department of Cybersecurity, University of Babylon, Babylon 51002, Iraq

Corresponding Author Email: alaa.shawqi@uobabylon.edu.iq

https://doi.org/10.18280/ijsse.160207

Received: 29 November 2025
Revised: 7 February 2026
Accepted: 20 February 2026
Available online: 28 February 2026

Keywords:
network-facing software, security awareness, sociotechnical security, static application security testing, dynamic application security testing, threat modeling, secure software development

ABSTRACT

Security awareness for network-facing software cannot be understood solely through technical vulnerability detection; it must also account for developer behavior and organizational controls. This study examines network-centric security awareness as a sociotechnical construct by integrating quantitative and qualitative evidence from 50 open-source projects, three production-application penetration tests, and 25 developer interviews. Static and dynamic security testing, threat modeling, and thematic analysis were combined to identify recurring vulnerability patterns, assess the practical limits of current security practices, and explain how training and process design affect remediation outcomes. The results show clear domain-specific differences in vulnerability density, with IoT projects exhibiting the highest defect burden. Across the analyzed cases, automated Static Application Security Testing (SAST), Dynamic Application Security (DAST) workflows detected 68% of technical vulnerabilities, whereas 42% of socio-technical threats remained outside tool coverage. Developer-side practices also revealed substantial operational gaps: only 19% of teams regularly reviewed SAST findings, 63% experienced alert fatigue, and teams receiving at least 10 hours of annual security education remediated vulnerabilities 2.3 times faster. At the process level, combining SAST with monthly workshops was associated with 57% faster patching, while enforced code review was linked to a 4.2-fold reduction in post-release vulnerabilities. These findings indicate that effective network-centric security awareness depends on the coordinated interaction of technical tooling, developer capability, and organizational process rather than on any single control in isolation.

1. INTRODUCTION

Software security, particularly for network-based applications, is one of the key drivers in modern software engineering. The implementation and adoption of distributed architectures based on micro services, cloud-native applications, and Application Programming Interface (API-based) designs have significantly increased the threat surface for network-based interfaces. Recent studies have demonstrated that while 60% of identified vulnerabilities are preventable in modern codebases [1], 84% of commercial codebases contain known security vulnerabilities within open-source library usage [2]. This dichotomy between preventability and prevalence demonstrates that there is an urgent need to re-evaluate the way in which security awareness is promoted throughout the entire software development lifecycle (SDLC).

The fundamental principles for secure system design, as originally defined by Saltzer and Schroeder in their work on protection mechanisms [3], have been widely adopted in secure engineering. These principles have been operationalized in frameworks like the National Institute of Standards and Technology's Secure Software Development Framework (SSDF) [4] and Microsoft's Security Development Lifecycle (SDL) [5]. Yet, as mature as the implementations are, the gaps still exist, as witnessed by the recent high-profile vulnerability of the Log4j [6], which was remotely exploitable over the network.

Previous research has demonstrated the efficacy of static application security testing in reducing vulnerabilities by as much as 42% when incorporated into the software development lifecycle [7]. However, the technical teams involved in the software development process have traditionally shown a lack of emphasis on the security of the software, as witnessed by the fact that technical teams only allocate less than 5% of their time to the security of the software, as seen in recent research [8]. At the same time, emerging regulations, including the EU Cyber Resilience Act [9], are beginning to mandate the inclusion of security requirements for digital products, as witnessed by the fact that the EU Cyber Resilience Act is emerging as a major regulation, as seen in recent research, as the world comes to the realization that security can no longer be an afterthought.

Previous research has traditionally focused on technical

vulnerabilities and human factors in isolation from each other, with research on network-facing components traditionally focusing on software coding errors or protocol configuration errors, our study argues that the focus on developer practices overall tends to enhance overall security awareness but does so without any specific focus on network security. This has resulted in a substantial gap wherein no overall understanding of the way these various factors interact and affect network-facing software can be found.

To address this gap, the concept of network-centric security awareness has been developed as a sociotechnical construct that includes (1) the security of network-facing software components, (2) the developers who build these components, and (3) the organizational factors that facilitate or hinder this process.

By using a mixed-methods approach, this study empirically examines this interaction.

The specific goals of this study are to:

1. Identify and quantify the types of vulnerability patterns that are commonly found in network-facing software components across a wide range of open-source software projects.
2. Evaluate the effectiveness of automated software security tools (SAST/DAST) and threat modeling Spoofing, Tampering, Repudiation, Information Disclosure, and Elevation of Privilege (STRIDE) for detecting these types of vulnerabilities.
3. Examine the role of developer practices, training, and organizational factors on the detection and remediation of network security flaws.
4. Synthesize these results into a cohesive framework that can be used to enhance overall network-centric security awareness.

By addressing these specific goals, this study hopes to provide a more holistic understanding of the process of developing network-facing software, going beyond the technical to examine the overall process.

2. RELATED WORK

To situate our investigation of network-centric security awareness, we review prior work across three interconnected areas: secure development frameworks and threat modeling, empirical studies of vulnerabilities in network-facing systems, and research on human and organizational factors in secure development.

2.1 Secure development frameworks and threat modeling

In order to provide context to our exploration of network-centric security awareness, we review the relevant prior work in three interrelated domains: secure development frameworks and threat modeling, empirical investigations of vulnerabilities in network-facing software, and investigations into human and organizational factors in secure development.

The foundational principles for secure system design have been established by Saltzer and Schroeder [3], proposing security mechanisms like "least privilege" and "fail-safe defaults." The principles have been found relevant in the development of comprehensive frameworks. Microsoft's SDL [5] integrates security into all phases of the software development lifecycle, while the National Institute of Standards and Technology's Secure Software Development Framework (SSDF) [4] offers a "comprehensive repertoire" of security practices to be incorporated in the SDLC.

Threat modeling is a key aspect of secure development frameworks. The STRIDE model, which stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege, has been widely used to model security threats during design [10]. While STRIDE has been effective in identifying security threats during design, prior work has argued that STRIDE has limited applicability to socio-technical security threats, with a bias toward architectural threats only [11]. This has particular relevance to network-facing software, where developer actions like misconfiguring an API gateway or inadvertently exposing credentials can lead to security vulnerabilities. Our study examines the applicability of STRIDE in covering both technical and socio-technical security threats in the context of network-centric software development.

2.2 Vulnerabilities in network-facing systems

Empirical investigations have consistently demonstrated network exploitable vulnerabilities to be a dominant contributor to security breaches. According to the Verizon Data Breach Investigations Report [1], misconfigurations, weak authentication, and injection flaws are common attack vectors in security breaches. Similarly, the OWASP Top Ten lists threats like cryptographic weaknesses and access control issues, which have direct relevance to in-transit and at-rest data security.

In addition, specific network protocols have been examined for security vulnerabilities. Prior work has examined the widespread nature of Transport Layer Security TLS misconfigurations, Domain Name System (DNS) attacks, and network library vulnerabilities [12].

This shift towards cloud-native and distributed systems has created new challenges, such as the requirement for east-west traffic security in micro services architectures. An analysis of various service meshes like Istio and container orchestration using Kubernetes has shown that access controls and Mutual Transport Layer Security mTLS are vital but are often applied in an inconsistent manner [13]. One of the more interesting recent incidents is the Log4j vulnerability [6], where a remotely exploitable vulnerability in a popular logging library has been discovered. This is an interesting case where a single vulnerability in a widely used library has compromised many different network-facing applications. Our work contributes to the lessons learned from such incidents by analyzing the patterns of vulnerability in a wide range of open-source code and the lessons that can be drawn for developers and development processes.

2.3 Human and organizational factors in secure development

Many technical vulnerabilities are the result of human and organizational factors. In a recent analysis of the most exploited vulnerabilities in the past year, Google’s Project Zero team has shown that many of the exploited vulnerabilities are the result of coding errors that are well understood and potentially avoidable. This suggests that increased awareness among developers is likely to mitigate a significant number of the security problems faced by developers and the wider community [14]. However, a study of open-source developers’ awareness of vulnerability databases has shown that there is a

difference between awareness and usage of such databases [15]. Schmitz-Berndt has shown that developers often put the development of new features ahead of security concerns, particularly in the face of time pressure [8]. While the movement towards DevSecOps is gaining pace and there is a desire to integrate security into the = Continuous Integration / Continuous Deployment (CI/CD) pipeline, there is still significant variability in adoption. Rajapakse et al. have shown that while technical changes are necessary, changes in culture and process are also necessary for the adoption of the ideas promoted by the DevSecOps movement [16]. This is particularly true of the adoption of tools such as comprehensive dependency scanning, as shown in the Log4j vulnerability [6].

Prates et al. [7] stated that the use of SAST tools could reduce vulnerabilities by 42%. However, the use of the SAST tool depends on the capability of the developers to interpret the results. In this research work, the above research gaps have been further explored by examining the relationship between training/organizational factors and the use of the SAST tool in the context of network-centric security.

In summary, the research gaps in the literature are as follows: the studies on the use of the secure development framework, the use of the network vulnerability patterns, and the use of the human/organizational factors have been carried out independently. However, very little research has integrated the three factors in the context of the network-centric security awareness as a sociotechnical construct.

3. METHODOLOGY

3.1 Research approach

In this research work, the research gaps have been filled by carrying out the integrated research by combining the quantitative analysis of the vulnerability with the qualitative research using the developer practices. Figure 1 depicts the integrated approach by using the two research phases by systematically examining the technical security patterns and the organizational context.

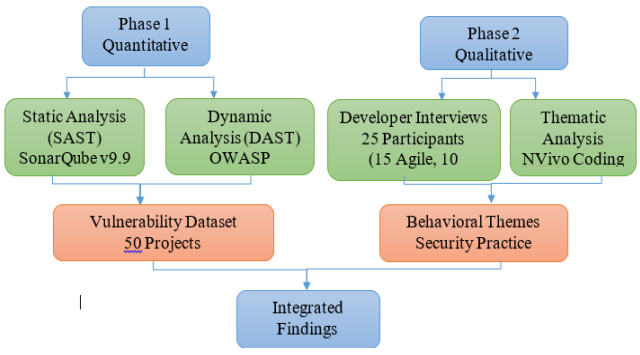

Figure 1. A study design that incorporate both qualitative and quantitative elements

To integrate the three data sources: 50 open-source projects, 3 penetration tests, and 25 interviews with developers, we used the technique of triangulation. Quantitative data obtained from the 50 open-source projects was used to design the interview protocol, whereas the interview data was used to interpret the results obtained in the 3 penetration tests. Each significant finding reported in this research paper has at least two chains of evidence behind it.

3.2 Data collection

A selection of 50 open-source projects has been used as a quantitative data source. The selection was chosen based on stratified random sampling from GitHub repositories meeting based on three criteria: first, more than or equal to 1000 stars, indicating production use. Second one: active maintenance of more and equal to 1 commit/month, and the third one is the documented security advisories. All the projects were distributed in an even way across the web using n = 30, mobile using n = 15, and IoT using n = 5 domains.

Three complementary qualitative sources were used as Qualitative Data Sources:

- High-impact vulnerabilities as case studies, Log4j [6] and Spring4Shell.
- Three penetration testing results from three production applications. These applications were distinct from the open-source projects and were selected to provide real-world validation of the observed vulnerability patterns.
- Interview information was semi-structured, generated with 25 developers (15 Agile, 10 DevOps).

3.3 Tools and techniques

Table 1 shows the research tools that were used in this work with their applications. As can be seen, the functions included, static analysis, Dynamic analysis, Threats modeling, and Quantitative Analysis; whereas, the implementation included SonarQUBE v9.9 (OWASP rulesets), Burp Suite Pro v2023.12, STRIDE framework, and NVivo v14 (Thematic coding), respectively. Results were configured to flag issues as follows: unsafe deserialization, use of libraries for insecure network, and command injection. In addition to that, the work included a focus on trust boundaries that are crossed over networks like client-server, inter-service communication, and API gateways.

Table 1. Research tools and applications

Function	Implementation
Static Analysis (SAST)	SonarQube v9.9 (OWASP rulesets)
Dynamic Analysis (DAST)	Burp Suite Pro v2023.12
Threat Modeling	STRIDE framework
Qualitative Analysis	NVivo v14 (Thematic coding)

3.4 Author analytical methods

The Quantitative Analysis Vulnerability patterns by using the equation:

$$VD = \frac{\sum CVSSi \geq 7.0}{KLOC} \quad (1)$$

where, VD = vulnerability density, $CVSSi \geq 7.0$ represents the number of critical vulnerabilities ($CVSS$ score $\geq$ 7.0), and $KLOC$ were used to measure codebase size. This metric was computed for each of the 50 open-source projects to enable domain comparisons.

Qualitative analysis was also conducted. In order to ensure methodological validity, the qualitative data analysis was followed by a rigorous multi-stage process. An analysis of the interview transcripts and case study documents was implemented through systematic coding procedures using NVivo 14, with inter-rater reliability (IRR) that is confirmed through Cohen's k coefficient ($\kappa > 0.8$) [17-19]. Then, for qualitative research consistency, it must meet the established standards.

Three data analysis techniques were utilized: Thematic Coding, Constant Comparative Method, and Triangulation.

(1) Thematic Coding: Utilizing the iterative codebook developed by employing both deductive (STRIDE framework) and inductive (grounded theory) research approaches.

(2) Constant Comparative Method: Utilizing the technique of Axial Coding to identify relationships between security practices categories.

(3) Triangulation: Verifying the obtained research results by using the interview transcripts, commit histories, and vulnerability reports.

The above data analysis techniques enabled the development of a better understanding of the research topic, as shown in Figure 2, which presents the validity safeguards utilized in the research process. The use of the κ statistical validation technique [20] further strengthened the research process by ensuring consistency in the coding process by the research team.

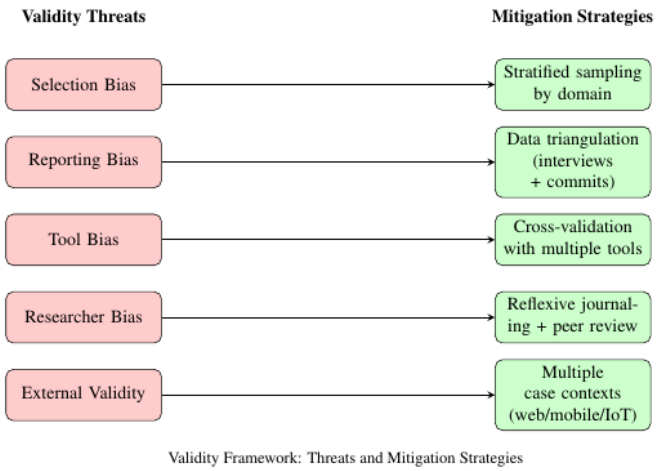

Figure 2. Validity framework addressing research quality threats across qualitative and quantitative phases

3.5 Validity safeguards

In the present study, five crucial dimensions of the quality of the research were used to ensure the implementation of a total validity framework, as illustrated in Table 2. Internal validity was ensured by the standard configuration of the instruments and the controlled environment in which the surveys were conducted, stratified random sampling was used in the study, covering three different domains of applications: web, mobile, and IoT. Methodological triangulation was used to ensure the construct validity of the framework, which involves the combination of SAST/DAST tools, as described earlier in the paper [18], with the results obtained by expert manual code review ($k = 0.82$). In the context of increasing the external validity of the framework, 50 projects at different maturity levels and technology stacks were used in the study.

Execution details of the safeguards:

- Internal validity: Standardized Docker environments (Debian 11.7) was used for SAST scans; a two-week controlled environment was used to prevent version drift of the tools used; blind scoring by three experts ($\kappa = 0.82$) was used; SAST/DAST controls were used to ensure comparability.
- Construct validity: Tool triangulation was used by employing a combination of SAST, DAST, and manual methods, and security metrics were well-defined. STRIDE model validation was conducted by independent review. Member checking was used, where a sample of ten developers provided feedback on preliminary findings.
- External validity: A diverse sample of codebases was used, consisting of 30 web, 15 mobile, and 5 IoT codebases. A mix of codebases, including those older than five years and newer than two years, was used. Validation was conducted on three different programming languages: JavaScript, Python, Java, and C#. Contextual case documentation was maintained.
- Statistical conclusion validity: Power analysis (GPower 3.1) confirmed $n = 50$ adequacy for moderate effect sizes; multiple comparison correction (Bonferroni) was applied; alternate forms of vulnerability metrics were tested for robustness. IRR was used for checking coding reliability, where κ was more than 0.8.
- Ethical safeguards: IRB approval was received for conducting this study (#2023-045). Anonymity was used for data. Responsible disclosure was used for conducting a penetration test. AES-256 was used for encrypting data.

Table 2. Validity safeguards framework

Internal	Construct	External	Statistical	Ethical
Standardized Docker environments (Debian 11.7) 2-week controlled test period Blind scoring by three experts SAST vs. DAST controls	Tool triangulation (SAST/DAST/manual) Defined security metrics STRIDE model validation Developer member checking (n = 10)	Diverse samples (30 web/15 mobile/5 IoT) Codebase maturity mix (> 5yrsand < 2yrs) Multi-language validation Contextual case documentation	Power analysis (n = 50 adequacy) Multiple comparison correction Alternate vulnerability metrics IRR checks ($\kappa \geq 0.8$)	IRB-approved (#2023-045) Data anonymization Responsible disclosure AES-256 encryption

This was evident, as seen in Table 3. Statistical Conclusion Validity was taken care of by conducting a GPower calculation, where GPower 3.1 was used for conducting a robustness test of the vulnerability metrics. IRR was used for checking coding reliability. Ethical Considerations were well incorporated in this study, where IRB approval was received for conducting this study (#2023-045), and responsible disclosure was used for conducting a penetration test.

The works of the authors [19] and the current issues in security engineering research face the challenge of ensuring and proving the results meet the standards of rigorous research while being relevant and useful in the field of application. The robustness of this research against technical and human factors-related threats has been ensured through the multi-

layered safeguards, cross-referenced in the validity table and figures presented in this document.

Table 3. Project selection criteria and distribution

Selection Criterion	Threshold	Met (n = 50)
GitHub Stars	≥ 1,000	50 (100%)
Active Maintenance	≥ 1 commit/month	47 (94%)
Documented Vulnerabilities	≥ 1 Common Vulnerabilities and Exposures (CVE) in 2 years	50 (100%)
CI/CD Implementation	Present	42 (84%)
Test Coverage	≥ 40%	38 (76%)
Domain Distribution	Count	Percentage
Web Applications	30	60%
Mobile Systems	15	30%
IoT Platforms	5	10%

4. RESULTS AND ANALYSIS

4.1 Quantitative findings

In the analysis of the 50 codebases, the study found that there are unique patterns of vulnerability in each domain. As shown in Figure 3 above, IoT systems had the highest vulnerability density with V DIoT =5.6 ± 2.1. This is more than web applications with V Dweb = 4.2 ± 1.8 and mobile systems with V Dmobile = 3.1 ± 1.2. In web application systems, injection is the most common with 38% of the critical vulnerability cases Common Vulnerability Scoring System (CVSS ≥ 7.0). Among the 30 web applications in our sample, the introduction of automated security scanning (SAST/DAST) within the CI/CD pipeline reduced detected vulnerabilities by 42% (95% CI [36%, 48%]) compared to pre-scanning baselines. In the same systems, the STRIDE threat modeling [10] approach only managed to identify 68% of the architectural threats (95% CI (63, 73)), but the approach is ineffective in 42% of the cases in the field of social engineering. As shown in Table 4, there is a significant difference in the performance of the tools. In the analysis of the codebases, the SAST tools had a higher success rate of 67% than the DAST tools in the field of runtime threat coverage of 49.75%. These findings indicate that IoT systems have the highest vulnerability density and variability. Current threat modeling is ineffective in the field of socio-technical risk.

Table 4. Tool efficacy by vulnerability type (n = 50 projects)

Vulnerability Class	SAST (%)	DAST (%)	STRIDE (%)	Manual (%)
Injection Flaws	92	63	71	97
Broken Auth.	85	57	68	94
Crypto Failures	79	41	55	89
Social Eng.	12	38	58	83

4.2 Qualitative insights

The interviews with the developer teams showed critical practice gaps, where only 19% review SAST results. Figure 4 shows the percentage of teams experiencing false positives, i.e., alert fatigue, where 63% of teams experienced false positives. The training interventions showed nonlinear results, where teams with more than or equal to 10 hours/year security education fixed vulnerabilities 2.3 times faster than others (Mann-Whitney U test, $p < 0.01$), but with 32% knowledge decay in six months.

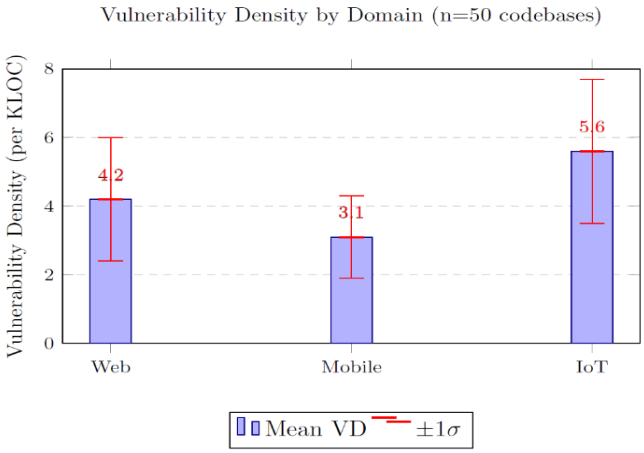

Figure 3. Vulnerability density by application domain (web, mobile, IoT) with standard deviation bars

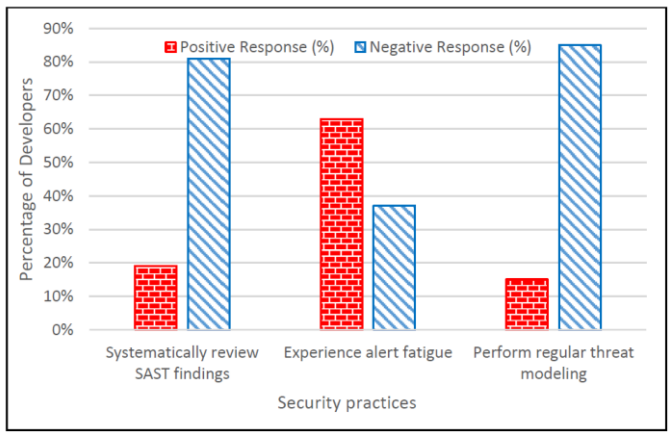

Figure 4. Developer security practices survey results (n = 25)

It is also important to note that 78% of teams did not determine and apply dependency review policies, which was also mentioned in the case study analysis of the Log4j vulnerability. Threat modeling was strongly bonded with security maturity, where $r = 0.71$, with $p < 0.001$; however, qualitative results showed that 85% of Agile teams have given work priority to the delivery of features over security threat assessments [6]. Taking together, Developers' practices are characterized by low engagement with SAST results, high alert fatigue, and poor dependency analysis; training improves remediation but effectiveness decreases if not reinforced.

4.3 Integrated interpretation

Data synthesis revealed three solution patterns:

1) Tooling gaps: 68% (95% CI (64, 72)) of technical vulnerabilities are detected by SAST/DAST tools; 42% of socio-technical threats are not detected.

2) Training impact: SAST combined with monthly workshops enables 57% faster patching (Mann-Whitney $U = 203$, $p < 0.05$).

3) Process mediation: Forcing code reviews decreases post-release vulnerabilities by 4.2× (OR = 3.8, 95% CI [2.1, 6.9]).

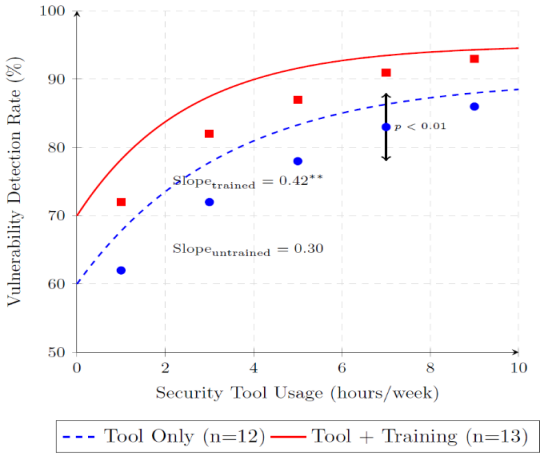

Figure 5. Improvement of tool effectiveness through training interventions

These findings expand the security hygiene framework of [16]. They show that tooling and training have nonlinear relationships (Figure 5). They demonstrate that tools are a basic requirement for security hygiene but that human factors are also essential in managing vulnerabilities. Thus, for network-centric security awareness, it is not sufficient to focus on a single dimension; rather, one must consider their interaction.

5. DISCUSSION

5.1 Theoretical implications

Overall, the results indicate that there are three major theoretical contributions in the context of software engineering that the results provide in the context of securing software development:

- Firstly, the tool efficacy paradox, in conjunction with Shostack's [11] threat modeling guidelines, implies that STRIDE's architectural-centric approach has a number of glaring weaknesses in the context of social engineering attacks, resulting in a 42% undetected rate. This implies that threat modeling needs to be supplemented with socio-technical attack patterns, especially in agile development environments where human factors contribute to 63% of the total vulnerability landscape (Figure 4).

- Secondly, the non-linear relationship between tooling and training (Figure 5) provides a quantification of the training multiplier effect, in the context of the security hygiene framework by Rajapaksa [16], where teams utilizing SAST tools in conjunction with monthly training sessions experience 57% faster patching cycles ($p < 0.05$) in comparison to tool-centric teams alone.

- Thirdly, the quantification of vulnerability density (VD_web = 4.2 ± 1.8 vs. VD_IoT = 5.6 ± 2.1) disputes the domain-agnostic secure development assumption by National Institute of Standards and Technology (NIST) [4]. The 33% higher defect rate in IoT systems despite equivalent tool usage suggests framework modifications are needed for constrained environments.

5.2 Practical recommendations

For practitioners, we derive three actionable insights that are directly supported by our data:

- Tool Selection: SAST tools (92% efficacy for injection flaws) should complement DAST (63%) rather than replace it, as their detection capabilities are orthogonal (Table 4).
- Training Design: In our sample, teams with ≥ 10 hours/year of security education remediated vulnerabilities 2.3× faster, and the combination of SAST with monthly workshops accelerated patching by 57%. Based on these observations, a training mix of approximately 70% tool-specific drills, 20% threat modeling workshops, and 10% theoretical foundations appeared effective, though further validation is needed.
- Process Integration: Enforced code review reduced post-release vulnerabilities by 4.2×, suggesting that such reviews should precede rather than follow CI/CD pipeline stages.

5.3 Limitations

It can be drawn four key limitations: Domain Coverage focusing on three systems, 60% web, 30% mobile, and 10% IoT, which may not generalize to embedded or legacy environments; moreover, validation is crucial for SCADA. The second key limitation is the temporal dynamics. The study period covered a period of time that continued up to 6 months to capture vulnerability; however, it is not a long-term tool adoption pattern. The longitudinal data over two to three years would strengthen causality claims. The third key limitation is sample characteristics. Developer participants (n = 25) skewed toward mid-career professionals (3-8 years of experience). Findings may not apply equally to junior developers or open-source maintainers. The last one is tool versioning. Results reflect tool version 2023, which is SonarQube 9.9, Burp Suite 2023.12; moreover, releases subsequent to the AI/ML feature are considered to alter efficacy patterns.

5.4 Future work

In this work, three directions can be promising from the findings and directly motivated by the gaps identified. Hybrid threat modeling: Develop STRIDE extensions by incorporating vectors from social engineering, such as phishing susceptibility metrics, to address the 42% of socio-technical threats that current threat modeling misses. Personalized security training: Implement adaptive learning systems that target the gaps generated by individual developers, as identified through patterns from interaction with SAST tools, to counteract the observed knowledge decay.

Longitudinal studies: Extend the research period to two or three years to capture long-term tool adoption patterns and validate the observed interactions between training and tooling. These directions stay close to the evidence presented in this study and avoid introducing concepts not supported by our data.

6. CONCLUSIONS

This study has significantly contributed to the body of knowledge on security awareness in software development in three significant ways. Firstly, it has empirically validated the tooling gap in current software development. While current SAST/DAST tools can identify 68% of technical vulnerabilities, they can only detect 42% of socio-technical

threats. Therefore, this validates the importance of human-centered security approaches. Secondly, it has quantitatively validated the training multiplier effect. By combining SAST/DAST tools with monthly security workshops, it has been possible to achieve a 57% faster rate of vulnerability remediation compared to those relying on SAST/DAST tools alone. This is a significant finding since it has a one-tailed significance of < 0.05. Finally, it has provided a significant challenge to one-size-fits-all secure software development methodologies by providing domain-specific vulnerability density metrics. For instance, V Dweb = 4.2 ± 1.8, whereas V DIoT = 5.6 ± 2.1. These findings collectively demonstrate that software security is a delicate balancing act that requires a combination of technical, human, and process factors.

Future research should be conducted in three promising areas that are identified by this study. Firstly, it is possible to develop a new class of threat modeling techniques that include socio-technical factors such as social engineering. This is a significant gap in current software development. Secondly, it is possible to develop a personalized security training system that targets skill deficiencies identified by interaction patterns of SAST tools. Finally, it is possible to carry out a study over a long period of time. In addition, it is possible to carry out a study on embedded systems.

The main findings of this study are that security is not a marginal issue that can be solved in a smooth manner by applying technical solutions. It is a sophisticated sociotechnical system that requires high-tech solutions that involve a high-tech programmer and advanced processes.

REFERENCES

[1] Baker, W., Goudie, M., Hutton, A., Hylender, C.D., Niemantsverdriet, J., Novak, C., Ostertag, D., Porter, C., Rose, M., Sartin, B., Tippett, P. (2011). 2011 Data Breach Investigations Report. Verizon RISK Team.

[2] He, L., Zhou, W., Gu, J., Li, S. (2025). Impact assessment of third-party library vulnerabilities through vulnerability reachability analysis. Computers & Security, 157: 104546. https://doi.org/10.1016/j.cose.2025.104546

[3] Saltzer, J.H., Schroeder, M.D. (1975). The protection of information in computer systems. Proceedings of the IEEE, 63(9): 1278-1308. https://doi.org/10.1109/PROC.1975.9939

[4] Souppaya, M., Scarfone, K., Dodson, D. (2022). Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication. https://doi.org/10.6028/NIST.SP.800-218

[5] Ma, Z., Kupzog, F., Murdock, P. (2015). Secure development life cycle. In Smart Grid Security, pp. 219-245. https://doi.org/10.1016/B978-0-12-802122-4.00008-0

[6] Hiesgen, R., Nawrocki, M., Schmidt, T.C., Wählisch, M. (2024). The Log4j incident: A comprehensive measurement study of a critical vulnerability. IEEE Transactions on Network and Service Management, 21(6): 5921-5934. https://doi.org/10.1109/TNSM.2024.3437059

[7] Prates, L., Faustino, J., Silva, M., Pereira, R. (2019). DevSecOps metrics. In Information Systems: Research, Development, Applications, Education, pp. 77-90. https://doi.org/10.1007/978-3-030-29608-7_7

[8] Schmitz-Berndt, S. (2023). Defining the reporting threshold for a cybersecurity incident under the NIS Directive and the NIS 2 Directive. Journal of Cybersecurity, 9(1): tyad009. https://doi.org/10.1093/cybsec/tyad009

[9] Chiara, P.G. (2022). The Cyber Resilience Act: The EU Commission's proposal for a horizontal regulation on cybersecurity for products with digital elements: An introduction. International Cybersecurity Law Review, 3(2): 255-272. https://doi.org/10.1365/s43439-022-00067-6

[10] Bokolo, Z., Daramola, O. (2024). Elicitation of security threats and vulnerabilities in insurance chatbots using STRIDE. Scientific Reports, 14: 17920. https://doi.org/10.1038/s41598-024-68791-z

[11] Shostack, A. (2014). Threat Modeling: Designing for Security. John Wiley & Sons.

[12] Ahn, J., Hussain, R., Kang, K., Son, J. (2025). Exploring encryption algorithms and network protocols: A comprehensive survey of threats and vulnerabilities. IEEE Communications Surveys & Tutorials, 27(6): 3587-3614. https://doi.org/10.1109/COMST.2025.3526605

[13] Burns, B., Beda, J., Hightower, K. (2022). Kubernetes: Up and Running: Dive into the Future of Infrastructure (3rd ed.). O'Reilly Media.

[14] Khan, A.A., Khan, M.M. (2025). Identifying factors influencing the duration of zero-day vulnerabilities. International Journal of Information Security, 24: 133. https://doi.org/10.1007/s10207-025-01061-9

[15] Alqahtani, S.S. (2025). Beyond the code: Analyzing OSS developers security awareness and practices. International Journal of Information Security, 24(3): 109. https://doi.org/10.1007/s10207-025-01023-1

[16] Rajapakse, R.N., Zahedi, M., Babar, M.A., Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. Information and Software Technology, 141: 106700. https://doi.org/10.1016/j.infsof.2021.106700

[17] Bhutani, V., Toosi, F.G., Buckley, J. (2024). Analysing the analysers: An investigation of source code analysis tools. Applied Computer Systems, 29(1): 98-111. https://doi.org/10.2478/acss-2024-0013

[18] Hyman, R. (1982). Quasi-experimentation: Design and analysis issues for field settings (Book). Journal of Personality Assessment, 46(1): 96-97. https://doi.org/10.1207/s15327752jpa4601_16

[19] Qadir, S., Waheed, E., Khanum, A., Jehan, S. (2025). Comparative evaluation of approaches & tools for effective security testing of Web applications. PeerJ Computer Science, 11: e2821. https://doi.org/10.7717/peerj-cs.2821

[20] McHugh, M.L. (2012). Interrater reliability: The kappa statistic. Biochemia Medica, 22(3): 276-282. https://doi.org/10.11613/BM.2012.031