Secure Coding Drift in LLM-Assisted Post-Quantum Cryptography Development: A Gamified Fix

R.D.N. Shakya
University of Moratuwa
Sri-Lanka, Moratuwa
shakyardn.26@uom.lk

C.P. Wijesiriwardana
University of Moratuwa
Sri-Lanka, Moratuwa
chaman@uom.lk

S.M.Vidanagamachchi
University of Ruhuna
Sri-Lanka, Matara
smv@dcs.ruh.ac.lk

Nalin A.G. Arachchilage
RMIT University
Australia, Melbourne
nalin.arachchilage@rmit.edu.au

Abstract

The transition to Post-Quantum Cryptography (PQC) introduces considerable implementation complexity, requiring strict adherence to constant-time execution, side-channel resistance, and precise parametrisation. Simultaneously, large language models (LLMs) are heavily embedded in software development workflows, including cryptographic engineering. While LLMs improve productivity, evidence shows that they frequently generate insecure or suboptimal code, particularly in security-critical domains. This paper introduces Secure Coding Drift in PQC, a novel socio-technical vulnerability model capturing the gradual degradation of secure coding practices due to sustained reliance on LLM-generated code. Unlike prior work that focuses on static vulnerabilities, we conceptualise security risk as a longitudinal behavioural phenomenon rising from human–AI interaction. To mitigate this, we propose a gamified, LLM-augmented secure coding framework that embeds adversarial evaluation, behavioural feedback, and security scoring into development workflows. Our approach reframes LLMs from passive assistants into active security co-pilots, contributing toward safer PQC implementation in AI-mediated environments.

CCS Concepts

• **Security and privacy** → **Cryptography**; **Software security engineering**; • **Computing methodologies** → **Artificial intelligence**.

Keywords

AI-Native development, Gamification framework, LLM, LLM-as-a-judge, PQC, Secure Coding Drift, Vibe coding

ACM Reference Format:

R.D.N. Shakya, C.P. Wijesiriwardana, S.M.Vidanagamachchi, and Nalin A.G. Arachchilage. 2026. Secure Coding Drift in LLM-Assisted Post-Quantum Cryptography Development: A Gamified Fix. In *Proceedings of International Workshop on Vulnerabilities in Generative Systems for Information Retrieval (SIGIR VulGen '26).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

SIGIR VulGen '26, Melbourne, Australia

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction and Background

The rapid advancement of quantum computing threatens widely used public-key cryptographic systems such as RSA and elliptic curve cryptography, as quantum algorithms like Shor's algorithm can solve their underlying mathematical problems in polynomial time [4, 7]. In response, Post-Quantum Cryptography (PQC) has emerged as the primary defence against quantum attacks, with the National Institute of Standards and Technology (NIST) [17] leading the standardisation of quantum-resistant algorithms. However, migrating to PQC is not a simple replacement of algorithms. It requires integration into complex systems while maintaining security, performance, and interoperability [4]. PQC implementations also introduce engineering challenges such as large key sizes, performance overhead, and strict parameter control [13]. More critically, they must satisfy strict security requirements, including constant-time execution and resistance to side-channel attacks. Even minor implementation errors can break theoretical security guarantees and lead to practical attacks [13, 23]. Therefore, secure coding is not merely desirable in PQC software development; it is a fundamental requirement for preserving cryptographic security.

At the same time, software engineering is shifting toward AI-native development methods, such as *vibe coding* [14]. In this approach, developers rely heavily on large language models (LLMs) for code generation and debugging [12]. While this improves productivity, it introduces serious security risks. Studies show that LLM-generated code often violates secure coding practices and may contain vulnerabilities even when functionally correct [18, 21]. This is especially problematic in cryptographic contexts [18]. For example, empirical studies on tools like GitHub Copilot show systematic failures to meet secure coding standards, especially in cryptographic contexts[18]. These risks are further amplified by prompt injection, data poisoning, and other adversarial attacks on LLM pipelines [6]. In addition, developers often accept AI-generated code with limited verification, especially when the output appears syntactically correct or authoritative [19, 24]. Over time, this over-reliance can reduce developers' critical evaluation skills and internal security

reasoning, creating *cognitive debt*: a gradual erosion of security-related knowledge, behaviours and vigilance caused by habitual dependence on AI-generated solutions [9, 15, 20, 25].

Existing research typically treats LLM-related security issues as static problems in generated code [18, 22, 24]. While this perspective is valuable, it overlooks the dynamic behavioural changes that emerge through prolonged interaction between developers and AI assistants [9, 25]. In practice, the security impact of LLM assistance is not only static but cumulative. Repeated exposure to auto-generated solutions can progressively reshape how developers think, reason, and validate code. This phenomenon motivates the concept of *Secure Coding Drift* (SCD) [3], which describes the gradual degradation of secure coding behaviour over time due to increasing reliance on AI tools. Developers may perform less verification, weaker threat analysis, reduced awareness of edge cases, and accept insecure patterns more easily [3, 10]. Rather than appearing as an immediate failure, drift manifests incrementally through behavioural adaptation, making it difficult to detect until insecure habits become normalised. Thus, the core risk is not only insecure generated code but also the long-term weakening of developers' security mindset. These issues are more critical in PQC development. Unlike traditional software development, PQC implementation requires deep understanding of subtle cryptographic properties that are often non-intuitive and highly sensitive to small coding decisions. A developer relying on vibe coding may receive code that appears mathematically valid and functional yet silently violates critical security assumptions such as constant-time execution or side-channel resistance. Since many PQC vulnerabilities are not visible through functional testing, insecure implementations may remain undetected until exploited [5, 13]. Furthermore, if developers increasingly trust LLM outputs without rigorous validation, secure coding drift can compound PQC implementation risks. Cognitive debt reduces the possibility that developers will question hidden cryptographic assumptions, while LLM vulnerabilities such as poisoned training data or adversarial prompting may further propagate insecure implementation patterns [9, 25]. Consequently, the intersection of vibe coding and PQC creates a high-risk environment in which both machine-generated vulnerabilities and human behavioural degradation jointly threaten software security.

Despite growing research on AI-assisted coding and PQC, limited work examines their long-term behavioural interaction. Most mitigation approaches focus on static solutions such as vulnerability detection tools rather than developer behaviour. To address this gap, we introduce *Secure Coding Drift in PQC (SCD-PQC)*, a behavioural and longitudinal model that explains how secure coding practices degrade in LLM-assisted PQC development environments. Building on this model, we propose a mitigation framework grounded in gamification and continuous feedback to reinforce secure coding awareness, encourage active verification of AI-generated outputs, and sustain long-term secure development habits. By shifting the focus from one-time vulnerability detection to behavioural resilience, this work aims to support safer AI-assisted adoption of PQC in real-world software engineering.

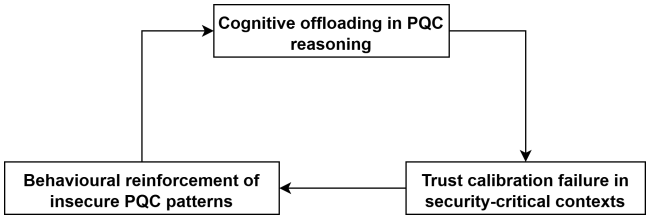

Figure 1: Secure Coding Drift in PQC (SCD-PQC) Model

2 Secure Coding Drift in PQC (SCD-PQC)

We define **SCD-PQC** as a progressive, behaviourally reinforced deviation from secure Post-Quantum Cryptography (PQC) implementation practices caused by sustained reliance on LLM-generated code within AI-mediated development workflows. This phenomenon reflects a gradual weakening of security-critical decision-making, driven by repeated interaction with AI systems that produce syntactically valid but not necessarily secure cryptographic code [10].

To illustrate, consider a developer named John implementing a lattice-based PQC key encapsulation mechanism. Initially, John carefully verifies constant-time properties and parameter selections. Over time, however, he increasingly accepts LLM-generated implementations for sampling noise vectors and performing modular arithmetic without full validation. Although the generated code appears correct and passes functional tests, it may introduce timing leakage or incorrect usage of randomness. As this pattern persists, John's ability and motivation to independently reason about PQC security properties gradually deteriorate, resulting in a measurable shift from secure to insecure coding practices.

2.1 SCD-PQC model

The SCD-PQC model consists of three interacting stages as illustrated in Figure 1 :

- **Cognitive offloading in PQC reasoning:** Developers delegate complex PQC implementation tasks, such as modular arithmetic optimisations or noise sampling, to LLMs, reducing their direct involvement in security-critical logic.
- **Trust calibration failure in security-critical contexts:** Repeated exposure to syntactically correct and plausible outputs leads developers to overestimate the correctness of LLM-generated cryptographic code, even though it may lack guarantees for constant-time execution or side-channel resistance.
- **Behavioural reinforcement of insecure PQC patterns:** Acceptance and reuse of generated code reinforce insecure practices, such as improper parameter selection, unsafe randomness usage, or leakage-prone implementations. Over time, these practices become default development behaviour rather than exceptions.

2.2 Possible Threat Vector and Impact

The SCD-PQC model assumes an adversarial and partially untrusted LLM integrated into the software development pipeline. Within this setting, security risks emerge from three primary sources: (i)

the generation of syntactically correct but insecure cryptographic code, (ii) adversarial manipulation through prompt injection or poisoned training data, and (iii) developer over-reliance that weakens independent validation of cryptographic correctness.

In this threat setting, the attacker does not require direct compromise of the target system. Instead, security degradation arises indirectly through routine developer interaction with the LLM. Even in the absence of malicious intent, repeated acceptance of AI-generated PQC code can introduce systematic vulnerabilities, including non-constant-time execution leading to timing and side-channel leakage, improper randomness generation, and incorrect or unsafe parameter selection.

The primary manifestations of this threat vector include:

- Introduction of non-constant-time implementations that are vulnerable to timing and side-channel attacks
- Misuse of cryptographic APIs and incorrect parameter configuration
- Reinforcement and propagation of insecure coding patterns present in training data
- Targeted exploitation via adversarial prompts that manipulate cryptographic code generation

The impact of this threat vector is particularly critical in PQC-enabled software systems, where failures are often non-functional and therefore difficult to detect. Unlike traditional software defects that typically result in immediate system malfunction, PQC implementation flaws may silently weaken security guarantees while preserving apparent functional correctness [5, 13]. As a result, such vulnerabilities can remain latent until they are exploited under adversarial conditions.

Overall, SCD-PQC reframes the security problem from isolated implementation errors to a continuous and cumulative degradation process that simultaneously affects developer behaviour and system-level cryptographic assurance.

3 Behavioural Amplification in AI-Native Developers

Developers operating within AI-mediated environments prioritise speed, iteration, and convenience in their interaction patterns [19, 24]. Although these behaviours increase productivity, they reduce opportunities for deep inspection of cryptographic implementations [9, 25]. In PQC contexts, where the correctness depends on subtle properties (e.g., leakage resistance), these behaviours amplify SCD-PQC [23]. Importantly, this is not a failure of developers, but a systemic shift in software engineering practices driven by AI integration [16]. This observation motivates interventions that target behavioural dynamics, rather than solely improving the quality of code generation.

To illustrate this behavioural shift, consider the implementation example of a PQC key encapsulation mechanism as previously described in Section 2. A developer named John initially applies careful verification of constant-time properties and parameter selection. However, as development progresses within an AI-mediated workflow, he increasingly relies on LLM-generated code for tasks such as noise sampling and modular arithmetic, often without performing full security validation. Although the generated code appears correct and passes functional tests, it may still introduce timing leakage or misuse of randomness.

A similar pattern emerges when the developer requests an LLM to generate PQC implementation code. The resulting implementation produces correct outputs under standard execution and satisfies unit tests. Due to time pressure and perceived reliability of prior AI outputs, the developer integrates the code without auditing its source or execution-time behaviour. Over successive iterations, such AI-assisted decisions accumulate across the system. While each integration appears reasonable, the overall codebase gradually incorporates non-verified PQC components, increasing exposure to side-channel leakage, non-constant-time executions, and weak randomness generation.

This example demonstrates how behavioural amplification in AI-native development reinforces the Secure Coding Drift process described in Section 2, transforming isolated acts of code acceptance into a sustained degradation of PQC assurance.

4 The Gamified Fix

We propose a Gamified LLM-augmented PQC Framework with three core components as illustrated in Figure 2. The operational structure of the framework is organised into three interacting layers. The interaction between these layers defines a structured execution flow that is grounded in both software security engineering principles and behavioural computing theory.

- **LLM-based code generation layer:** This layer generates PQC implementations while exposing intermediate outputs for inspection and traceability. It enhances transparency in code synthesis by enabling developers to observe and critically evaluate the generation process rather than treating outputs as final artefacts. Conceptually, this layer operationalises AI-assisted synthesis of PQC implementations, where the LLM functions as a probabilistic program generator conditioned on developer intent. This aligns with contemporary software engineering practice, where generative models are increasingly embedded into development pipelines to reduce implementation overhead while maintaining functional correctness.

- **Security evaluation layer:** This layer performs automated security assessment by combining rule-based static analysis with LLM-as-a-Judge mechanisms. Generative AI tools are leveraged to support semantic reasoning about cryptographic correctness that is difficult to capture through deterministic rules alone [2]. This hybrid design strengthens vulnerability detection for issues such as timing leakage, insecure randomness, and cryptographic parameter misuse, while complementing traditional static verification techniques.
  More specifically, the security evaluation layer introduces a hybrid verification paradigm that integrates deterministic and probabilistic reasoning. Rule-based static analysis provides formalised checks for known vulnerability classes, whereas LLM-as-a-Judge components approximate expert-level reasoning for cryptographic code review. This dual mechanism is particularly relevant in PQC settings, where

vulnerabilities often arise from subtle violations of implementation constraints such as constant-time execution and parameter validity. By embedding generative AI within the evaluation loop, the framework extends conventional static analysis with context-sensitive judgment, enabling the detection of non-trivial security defects that are difficult to encode as explicit rules [2].
In addition, the evaluation process produces a drift indicator, which quantifies the extent to which developer behaviour is shifting toward insecure implementation patterns based on observed vulnerabilities and repeated coding decisions.

- **Gamification layer:** This layer operationalises behavioural reinforcement through scoring, feedback, progression, and challenge-based learning. It incentivises correct verification actions, rewards secure code corrections, and promotes active identification of vulnerabilities, while discouraging unvalidated acceptance of LLM-generated outputs.
From a behavioural perspective, the gamification layer is grounded in security and human-computer interaction research, which shows that sustained behavioural change is more effectively achieved through reinforcement mechanisms than through passive instruction alone. By introducing structured feedback loops, visible progression, and reward-based incentives, this layer directly shapes developer decision-making across repeated interactions. Prior studies demonstrate that such mechanisms improve engagement with secure coding practices and strengthen long-term adherence to security guidelines by reinforcing intrinsic motivation and desired behavioural patterns [1, 8, 11]. Within the proposed framework, this layer functions as a behavioural control mechanism that regulates how developers respond to AI-generated code and security evaluation feedback.

Collectively, the three layers form a comprehensive feedback system in which code generation, automated evaluation, and behavioural reinforcement operate in a continuous loop. This design is consistent with cybernetic principles of adaptive systems, where feedback is used to regulate system behaviour over time. In the present context, feedback is not only technical (i.e., vulnerability detection) but also behavioural (i.e., shaping developer interaction patterns), thereby enabling simultaneous improvement of code quality and mitigation of Secure Coding Drift in PQC (SCD-PQC).

4.1 Gamification Rationale and Behavioural Alignment

Gamification is promising in this context because Secure Coding Drift is fundamentally a behavioural phenomenon rather than a purely technical one. In AI-native development, developers optimise for speed and correctness signals provided by LLMs, often at the expense of security reasoning. Gamification reshapes this incentive structure by introducing explicit rewards for secure behaviour, such as identifying vulnerabilities, rejecting insecure suggestions, and correctly validating cryptographic properties [1, 8, 11]. This shifts developer attention from output acceptance to process quality, thereby improving trust calibration and reducing cognitive offloading.

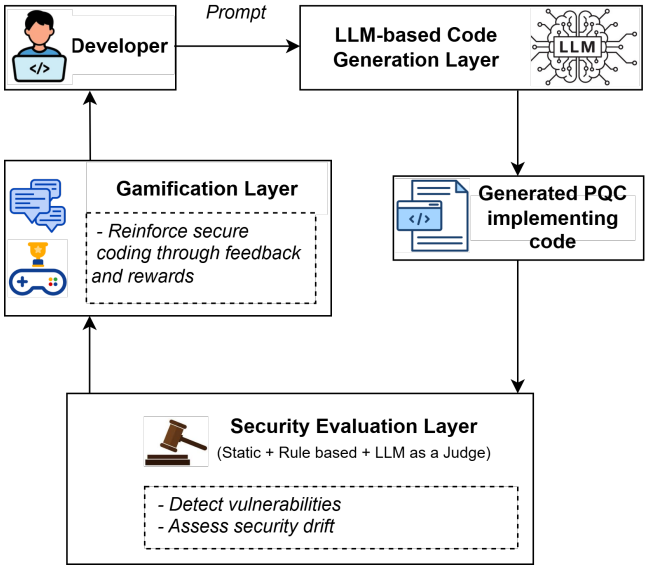

Figure 2: LLM-augmented Gamified PQC engineering framework

4.2 Metrics and Mechanisms

The proposed framework operationalises behavioural change through measurable indicators distributed across the security evaluation and gamification layers [9]. These metrics are designed to quantify both technical security quality and behavioural adaptation over time.

Within the **security evaluation layer**, the generated PQC code is analysed to compute a *drift indicator*, which estimates the degree to which the developer's coding behaviour is moving toward insecure implementation patterns. This indicator is derived from multiple security-related metrics, including vulnerability density (number of detected vulnerabilities per code segment), constant-time violations, and cryptographic misuse rate, such as unsafe randomness usage or incorrect parameter selection. Collectively, these metrics provide a technical measure of implementation quality and act as early warning signals of SCD-PQC.

Within the **gamification layer**, evaluation outcomes are translated into behavioural feedback using scores, rewards, and challenge-based learning tasks. This layer aims to influence developer decision-making by reinforcing secure validation behaviours and discouraging blind acceptance of AI-generated outputs. This layer uses the following core metrics:

- **Security correctness score:** Measures whether the generated PQC code satisfies key security requirements, including constant-time execution, correct parameter selection, and safe randomness usage.
- **Verification engagement index:** Measures the frequency and depth of developer validation actions, such as manual code review, invocation of static analysis tools, and adversarial testing attempts.

- **Drift reduction score:** Quantifies the reduction of repeated insecure coding patterns over time, indicating whether secure coding behaviour improves through repeated interaction with the framework.

Gamification mechanisms are implemented through points, levels, security scores, and challenge tasks. Developers receive higher rewards when they successfully identify subtle cryptographic flaws, reject insecure LLM-generated suggestions, or apply secure corrections. Conversely, repeated acceptance of unverified or insecure code lowers progression rates and triggers corrective challenges, thereby encouraging reflective and security-aware development behaviour.

4.3 Design Principles

The framework is governed by four design principles:

- **Immediate Feedback:** Security-related actions are reflected in real-time scoring updates.
- **Cognitive Minimalism:** The system avoids excessive cognitive load on developers.
- **Interpretability:** All scoring signals are transparent and explainable.
- **Robustness to Gaming:** The reward structure discourages superficial compliance behaviours.

4.4 Illustrative Case Study

Consider a developer implementing a lattice-based PQC key encapsulation mechanism using the proposed framework. Initially, the LLM generates a valid implementation of a noise sampling function. The security evaluation layer flags a potential non-constant-time operation. The gamification layer assigns the developer a challenge: identify and correct the vulnerability to earn progression points. The developer modifies the implementation by introducing constant-time sampling and verifies randomness sources using static analysis tools. As a result, the system rewards secure correction rather than passive acceptance. Over repeated tasks, the developer transitions from accepting default LLM outputs to actively auditing cryptographic properties before integration.

4.5 Proposed Evaluation Plan

The effectiveness of the proposed framework will be evaluated through a mixed-method experimental design. We plan to conduct a controlled study involving two groups of AI-assisted developers: one group using standard LLM-based development tools and another using the proposed gamified PQC framework. Both groups will be assigned PQC implementation tasks containing intentionally embedded vulnerabilities.

The framework will be assessed using both quantitative and qualitative measures, including vulnerability detection rate, secure code acceptance rate, time-to-correction, and behavioural consistency across repeated tasks. In addition, longitudinal analysis will be performed to examine whether Secure Coding Drift decreases over time in the intervention group.

Statistical comparison between the two groups will be used to determine the effectiveness of gamification in mitigating behavioural degradation in AI-native PQC development environments. This evaluation will provide empirical evidence on whether the proposed framework improves both secure coding behaviour and overall cryptographic implementation quality.

5 Discussion and Research Directions

The proposed SCD-PQC model and gamified mitigation framework open several important research directions at the intersection of generative AI, PQC, software engineering, and human-centred security. More broadly, this work shifts the discussion from viewing AI-assisted secure coding as a purely technical problem toward understanding it as a socio-technical challenge involving both machine-generated vulnerabilities and human behavioural adaptation.

The framework provides a basis for studying secure coding behaviour in AI-native development environments. It enables longitudinal analysis of PQC coding drift and comparative studies between human-only, standard AI-assisted, and gamified workflows to understand how interaction models affect security outcomes.

An important direction for future work is the empirical evaluation of the proposed framework. Controlled and longitudinal studies are required to evaluate whether the framework reduces SCD-PQC. Such studies may measure trust calibration, verification frequency, and vulnerability acceptance rates. The framework can also be implemented as an IDE plugin or development tool to support real-world deployment and evaluation.

From a generative AI perspective, future research should focus on improving LLM robustness, LLM-as-a-Judge reliability, prompt injection resistance, and explainable security reasoning. Domain-specific LLMs for cryptographic programming are another promising direction.

From a PQC software engineering perspective, the work motivates new security metrics such as side-channel vulnerability density, PQC compliance scores, and cryptographic misuse rates. It also encourages integration of formal verification and symbolic analysis into AI-assisted development pipelines.

From a human behavioural perspective, future studies should investigate cognitive offloading, trust calibration, and over-reliance in AI-native developers. These factors are critical for designing interventions that sustain secure coding behaviour, especially as personalised AI tools become more prevalent.

Overall, addressing these challenges requires interdisciplinary collaboration across generative AI, PQC, software engineering, and human-centred security. Such collaboration will be critical for ensuring that future AI-assisted software development improves productivity without compromising long-term PQC security.

6 Conclusion

This paper introduced SCD-PQC, a behavioural vulnerability emerging from LLM integration in PQC software development. It reframes security degradation as a longitudinal human–AI interaction rather than a static coding issue, extending prior vulnerability analysis. To address this, a gamified LLM-augmented framework was proposed, incorporating continuous feedback, adversarial evaluation, and behavioural incentives into development workflows. This approach repositions LLMs as active security-aware agents rather than passive code generators. Overall, the work contributes a conceptual

model and mitigation strategy for secure AI-assisted PQC development and identifies future research at the intersection of LLMs, software security, and human behaviour.

7 Relevance to Generative IR

Although this work is situated in secure software engineering, it is closely related to generative information retrieval systems. Modern LLM-assisted coding environments operate through a retrieval-and-generation process in which prompts, contextual information, and retrieved code examples influence generated outputs. From this perspective, secure coding drift can be viewed as a consequence of repeated interaction with generative systems, where developers increasingly rely on retrieved and generated artifacts without sufficient verification. Understanding and mitigating such behavioural effects contributes to broader efforts to improve trustworthiness, reliability, and human oversight in generative IR workflows.

Positionality Statement

This work is positioned at the intersection of PQC, software engineering, generative AI, and human-centred security. As researchers with backgrounds in cybersecurity, secure software engineering, and human-centred security, we approach PQC migration not only as a cryptographic challenge but also as a socio-technical problem involving developer cognition, behaviour, and decision-making. Our perspective is shaped by the assumption that the security risks of AI-assisted software development cannot be fully understood by code-level vulnerability analysis alone. We argue that sustained interaction with LLMs influences how developers reason about security, calibrate trust, and perform verification. This assumption directly informs our proposal of Secure Coding Drift in PQC (SCD-PQC) as a behavioural model rather than a purely technical failure model. We also adopt a human-centred security perspective, which emphasises that insecure outcomes often emerge from the interaction between humans, tools, and workflows rather than from isolated developer mistakes. Consequently, our proposed mitigation framework prioritises behavioural interventions, particularly gamification and continuous feedback, to reinforce secure coding practices in AI-native development environments. We acknowledge that our perspective may bias the proposed framework towards behavioural interventions over organisational, economic, or regulatory solutions. Furthermore, our model is currently conceptual and has not yet been empirically validated in large-scale industrial settings. Future empirical studies involving diverse developer populations and real-world PQC implementations are necessary to evaluate and refine these assumptions.

Acknowledgement of Country

The authors acknowledge the peoples of the Woi Wurrung and Boon Wurrung language groups of the eastern Kulin Nation on whose unceded lands ACM SIGIR 2026 is hosted. We pay our respects to their Elders past and present, and extend that respect to all Aboriginal and Torres Strait Islander peoples today and their continuing connection to land, sea, sky, and community.

References

[1] Abdulrahman Hassan Alhazmi, Mumtaz Abdul Hameed, and Nalin Asanka Gamagedara Arachchilage. 2022. Developers' Privacy Education: A game framework to stimulate secure coding behaviour. In *2022 IEEE Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta)*. IEEE, Haikou, China, 2255–2264. doi:10.1109/SmartWorld-UIC-ATC-ScalCom-DigitalTwin-PriComp-Metaverse56740.2022.00321

[2] Sathwik Amburi, Tiago Espinha Gasiba, Ulrike Lechner, and Maria Pinto-Albuquerque. 2025. Enabling Secure Coding: Exploring GenAI for Developer Training and Education. In *6th International Computer Programming Education Conference (ICPEC 2025) (OpenAccess Series in Informatics (OASIcs), Vol. 114)*, Ricardo Queirós, Mário Pinto, Filipe Portela, and Alberto Simões (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:14. doi:10.4230/OASIcs.ICPEC.2025.2

[3] Apiiro. 2026. What Is Security Drift? Indicators & Common Causes. https://apiiro.com/glossary/security-drift/

[4] Abdullah Aydeger, Engin Zeydan, Awaneesh Kumar Yadav, Kasun T. Hemachandra, and Madhusanka Liyanage. 2024. Towards a Quantum-Resilient Future: Strategies for Transitioning to Post-Quantum Cryptography. In *2024 15th International Conference on Network of the Future (NoF)*. IEEE, Castelldefels, Spain, 195–203. doi:10.1109/NoF62948.2024.10741441

[5] Emils Bagirovs, Grigory Provodin, Tuomo Sipola, and Jari Hautamäki. 2024. Applications of Post-quantum Cryptography. *Proceedings of the 23rd European Conference on Cyber Warfare and Security (2024)* 23, 1 (2024), 49–57. doi:10.48550/arXiv.2406.13258

[6] Aymen Dia Eddine Berini, Norziana Jamil, Ala-Eddine Benrazek, Abderrahmane Lakas, Leila Ismail, Mohamed Amine Ferrag, and Kwok-Yan Lam. 2026. Security and privacy in LLMs: A comprehensive survey of threats and mitigation strategies. *Information Fusion* 132 (2026), 104241. doi:10.1016/j.inffus.2026.104241

[7] Daniel J. Bernstein and Tanja Lange. 2017. Post-quantum cryptography. *Nature* 549, 7671 (2017), 188–194. doi:10.1038/nature23461

[8] Iasonas Diakoumakos. 2023. Enhancing Cyber Security Education and Training through Gamification. In *Proceedings of the 2nd International Conference of the ACM Greek SIGCHI Chapter* (Athens, Greece) *(CHIGREECE '23)*. Association for Computing Machinery, New York, NY, USA, Article 31, 5 pages. doi:10.1145/3609987.3610016

[9] Ramtin Ehsani, Shriya Rawal, Yuanfang Cai, and Preetha Chatterjee. 2026. Faster Code, Deeper Debt? A Multivocal Literature Review on Technical Debt and Its Early Signs in LLM-Assisted Software Development. *ACM Trans. Softw. Eng. Methodol.* 35, 7 (June 2026), 1–38. doi:10.1145/3820165 Just Accepted.

[10] Magdalena Glas, Christoph Nirschl, Bar Lanyado, and Johan van Niekerk. 2026. Insecure by design? A human-centric security perspective on AI-assisted software development. *Computers & Security* 164 (2026), 104842. doi:10.1016/j.cose.2026.104842

[11] Raden Budiarto Hadiprakoso and Rudolf Paris Parlindungan Sihombing. 2024. CodeGuardians: A Gamified Learning for Enhancing Secure Coding Practices with AI-Driven Feedback. *Ultima Infosys: Jurnal Ilmu Sistem Informasi* 15, 2 (December 2024), 105–112. https://www.researchgate.net/publication/387376972_CodeGuardians_A_Gamified_Learning_for_Enhancing_Secure_Coding_Practices_with_AI-Driven_Feedback

[12] Md. Asraful Haque. 2025. LLMs: A game-changer for software engineers? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 5, 1 (2025), 100204. doi:10.1016/j.tbench.2025.100204

[13] Julius Hekkala, Mari Muurman, Kimmo Halunen, and Visa Antero Vallivaara. 2023. Implementing Post-quantum Cryptography for Developers. *SN Computer Science* 4, 4 (2023), 365. doi:10.1007/s42979-023-01724-1

[14] Andrej Karpathy. 2025. I am 'vibe coding' now. I just vibe with the code, look at it, see if it looks okay, then run it, see what happens, and iterate. I am no longer really a programmer, I am an LLM reviewer. X (formerly Twitter). https://x.com Accessed: 2026-06-17.

[15] Shiri Melumad and Jin Ho Yun. 2025. Experimental evidence of the effects of large language models versus web search on depth of learning. *PNAS Nexus* 4, 10 (10 2025), pgaf316. arXiv:https://academic.oup.com/pnasnexus/article-pdf/4/10/pgaf316/64956155/pgaf316.pdf doi:10.1093/pnasnexus/pgaf316

[16] Jasmine Moreira. 2026. Technical Foundation Document IACDM: Interactive Adversarial Convergence Development Methodology. arXiv:2604.16399v2 [cs.SE] https://doi.org/10.48550/arXiv.2604.16399

[17] National Institute of Standards and Technology. 2026. Post-Quantum Cryptography Project. https://csrc.nist.gov/projects/post-quantum-cryptography. Accessed: 2026-06-17.

[18] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[19] 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* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[20] Duncan Pritchard. 2026. Why Technology Doesn't Normally Make You Dumber, but Agentic AI Will. *International Journal of Human–Computer Interaction* 0, 0 (2026), 1–11. arXiv:https://doi.org/10.1080/10447318.2026.2631678 doi:10.1080/10447318.2026.2631678

[21] Leonardo Criollo Ramírez, Xavier Limón, Ángel J. Sánchez-García, and Juan Carlos Pérez-Arriaga. 2025. State of the Art of the Security of Code Generated by LLMs: A Multivocal Literature Review. *Programming and Computer Software* 51, 8 (dec 2025), 587–604. doi:10.1134/S0361768825700446

[22] Shivani Shukla, Himanshu Joshi, and Romilla Syed. 2025. Security Degradation in Iterative AI Code Generation: A Systematic Analysis of the Paradox. In *2025 IEEE International Symposium on Technology and Society (ISTAS)*. IEEE, Santa Clara, CA, USA, 1–8. doi:10.1109/ISTAS65609.2025.11269659

[23] Marthin Toruan, R. D. N. Shakya, Samuel Tseitkin, Raymond K. Zhao, and Nalin Arachchilage. 2026. When Security Meets Usability: An Empirical Investigation of Post-Quantum Cryptography APIs. In *Proceedings of the 2026 Symposium on Usable Security and Privacy (USEC) (NDSS Symposium 2026)*. Internet Society, San Diego, CA, USA, 1–16. doi:10.14722/usec.2026.23076

[24] Asli Yardim, Raphael Serafini, Nadine Jost, Anna-Marie Ortloff, Joshua Gabriel Speckels, and Alena Naiakshina. 2026. "The AI tool can't make it any worse." Investigating Developers' Security Behavior with AI Assistants in a Password Storage Study. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, Article 1216, 33 pages. doi:10.1145/3772318.3791693

[25] Xinyi Zhou, Zeinadsadat Saghi, Sadra Sabouri, Rahul Pandita, Mollie McGuire, and Souti Chattopadhyay. 2026. Cognitive Biases in LLM-Assisted Software Development. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering* (Rio de Janeiro, Brazil) *(ICSE '26)*. ACM, New York, NY, USA, 13 pages. doi:10.1145/3744916.3773104

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009