

Does Indirection Hurt? Paired Evidence on LLM Review of Indirectly Written Cryptographic API Misuse

Sunny Chokshi

*School of Computer and Information Sciences
University of the Cumberland
Williamsburg, KY, USA
schokshi03616@ucumberlands.edu*

Abstract—LLM code reviewers are trustworthy only if their verdicts follow what the code does. I test this for cryptographic API misuse with 90 pairs of Python snippets across nine misuse classes, each pairing a vulnerable snippet with a secure twin. For 80 pairs I wrote a plain form, with the misuse at the call site under its familiar name, and an indirect form that reaches the same choice through an alias, constant, default, or configuration value. Across 16 open models, the secure twins show that two models flag nearly every snippet and four more flag over 70% of secure code. On the 80 pairs, the indirect form lowered the balanced accuracy of the seven strongest models by 4.5 points (pair-cluster 95% CI 1.3 to 7.6), and by 5.3 points (CI 2.1 to 8.5) on 47 pairs that survive a post hoc stimulus audit. After the audit the effect is mainly extra flags on correct code: false positives rose 9.1 points (CI 4.2 to 14.6), mostly in three models. In exploratory splits, most of the net false-alarm rise sits in pairs where the safe value has left its usual spelling or place at the call site, for example a 3072-bit key size written as 0xC00. Detection of real misuse fell 3.6 points over all 80 pairs (CI 1.0 to 6.5) and 1.5 points (CI -1.6 to 4.8) on the audited pairs.

Index Terms—trustworthy AI, LLM code review, cryptographic misuse, shortcut learning, vulnerability detection, evaluation

I. INTRODUCTION

A security reviewer earns trust when its verdict depends on the property it claims to check. A human reviewer who flags `hashlib.md5` because the word MD5 looks dangerous, and who passes the same broken hash when it arrives through an unfamiliar call, is not checking hash strength. The same question applies to large language models (LLMs), which teams place in pull-request checks and pre-commit hooks as a first security pass.

Cryptographic API misuse is a useful place to ask it. The flaw class is common and damaging, and static analysis work has documented it in real code for more than a decade [1], [2]. Many misuse patterns also have a well-known token (MD5, `MODE_ECB`, `verify=False`) that a model may have learned to associate with insecurity. If a model reasons about what the

code does, moving the misuse away from that token should barely matter. If it partly matches tokens, the move should cost accuracy.

In earlier work I released CryptoBench, a small matched-control benchmark, and ran seven small local models on it; the three that could tell secure from vulnerable code caught misuse with a known-bad name far more often than misuse inside ordinary API use, which I offered as a hypothesis [3], [4]. That study could not separate the name from the misuse class, because the two moved together. Here I test the form of the code directly, on the same pairs. My contributions are:

- CryptoBench v2, 90 vulnerable and secure pairs across nine misuse classes, 80 of them written in both a plain form and an indirect form.
- An evaluation of 16 open models from 0.5B to 120B parameters, which shows why matched secure twins are needed before any detection rate is read.
- A within-pair comparison on the seven strongest models, with pair-cluster intervals, a stimulus audit, and a reading of the replies that shows where the effect sits.

II. RELATED WORK

Cryptographic misuse. CryptoLint measured how often Android apps misuse cryptographic APIs and named the recurring patterns [1]. CogniCrypt generated secure code for common cryptographic tasks and ran static analyses in the IDE to flag misuse [5], and CryptoGuard scaled precise detection to large Java projects [2]. My nine classes and their CWE mappings follow this line of work.

LLMs as vulnerability detectors. Ullah et al. found non-deterministic answers and unfaithful reasoning in LLM security reviewers, and showed that semantics-preserving changes such as renaming functions and variables can flip their verdicts [6]. Risse and Böhme showed that learned detectors often rely on features unrelated to the vulnerability [7], and PrimeVul added a large set of paired vulnerable and patched functions and showed that code models rarely tell the two apart [8]. Li et al. report that with enough code context, larger models and more test-time compute improve detection, though with

diminishing returns, and extended reasoning lowers recall [9]. For cryptographic misuse, Firouzi et al. compared GPT-3.5-turbo with CryptoGuard [10], Masood and Vargas Martin evaluated two LLMs against three static tools [11], Xia et al. traced most false alarms to faulty cryptographic knowledge and misread code semantics [12], and Baek et al. evaluated four models for the task and recommended CodeT5 [13]. My work is closest to Ullah et al.’s renaming test and to PrimeVul’s pairing. I narrow both to cryptographic misuse, keep the misuse fixed within each pair, and vary only how the code reaches it.

Shortcuts. Geirhos et al. describe shortcut learning, where a model succeeds through cues that do not transfer to the intended task [14], and Yefet et al. showed that renaming variables can change predictions of models of code [15]. My indirect form is non-adversarial. I do not search for evasive code; I write the misuse in forms a developer could produce, although some, such as a key size written in hexadecimal, are unusual.

III. CRYPTOBENCH V2

A. Pairs and classes

The nine classes are weak hashing for integrity (CWE-328), ECB mode (CWE-327), a hardcoded key (CWE-321), a static IV or nonce (CWE-1204), a weak random generator for a security value (CWE-338), a short asymmetric key (CWE-326), an obsolete cipher (CWE-327), an unsalted, fixed-salt, or fast password hash (CWE-759, CWE-760, CWE-916), and disabled certificate verification (CWE-295). Each class has ten pairs, 180 snippets in total. A secure twin performs the same task correctly and keeps the structure where possible. Some twins needed a restructured body, for example ECB twins that move to an authenticated mode.

B. Plain and indirect forms

In the plain form the misuse appears at the call site under its usual name, for example `AES.new(key, AES.MODE_ECB)`. In the indirect form the same misuse is reached through an alias, a module constant, a library default, a number, or a configuration value. The indirect form does not always remove the familiar token from the file. For example, the indirect weak-hash snippet sets `DIGEST = "md5"` and calls `hashlib.new(DIGEST, ...)`. I therefore classify each indirect vulnerable snippet by whether the class’s signature token is still present (Section V-B). In v2, 45 pairs were written plain, 35 indirect, and 10 in an alias-only variant of the indirect form.

C. Protocol

Each snippet is sent alone with a fixed instruction: decide whether the code contains a security vulnerability and answer `VERDICT: VULNERABLE` or `VERDICT: SAFE`, followed by a one-sentence reason. A parser reads the verdict line and falls back to the reply text when the line is missing. Models ran through Ollama with temperature 0, seed 42, a 4,096-token

TABLE I
STUDY 1: ALL 180 SNIPPETS, THREE REPEATS (270 VULNERABLE AND 270 SECURE CALLS PER MODEL).

Model	TPR (%)	FPR (%)	bAcc
gpt-oss 20B	93.7	3.3	0.952
gpt-oss 120B	95.9	9.6	0.931
qwen2.5-coder 32B	85.9	20.0	0.830
llama3.3 70B	91.1	34.1	0.785
codestral 22B	86.3	38.1	0.741
qwen3-coder 30B	93.0	45.9	0.735
qwen2.5-coder 14B	93.0	50.4	0.713
qwen2.5-coder 7B	77.8	45.2	0.663
codellama 34B	96.7	74.8	0.609
devstral 24B	100.0	82.6	0.587
qwen2.5-coder 3B	84.4	71.1	0.567
deepseek-coder-v2 16B	63.7	51.9	0.559
deepseek-coder 6.7B	37.8	26.3	0.557
qwen2.5-coder 1.5B	87.8	80.0	0.539
codellama 7B	100.0	99.6	0.502
qwen2.5-coder 0.5B	100.0	100.0	0.500

context, and a 256-token output cap (2,048 for the two gpt-oss models, which ran with reasoning effort set to low and whose reasoning tokens count against the cap). Sixteen Study 1 replies reached the cap (14 from codellama 7B, 2 from deepseek-coder 6.7B), all after the verdict line; none of the seven Study 2 models reached it. Every snippet ran three times per model. I report the true positive rate (TPR) on vulnerable snippets, the false positive rate (FPR) on secure twins, and balanced accuracy, $bAcc = (TPR + 1 - FPR)/2$. A model that flags everything scores 0.5.

IV. STUDY 1: SIXTEEN MODELS

I ran all 180 snippets on 16 open models, 8,640 scored calls with no infrastructure errors (Table I; Wilson intervals [16] are in the released results).

The secure twins change the ranking. Judged on TPR alone, codellama 7B, devstral 24B, and qwen2.5-coder 0.5B would rank near the top at or near 100%. Their FPRs show that they flag most correct code too. codellama 7B and qwen2.5-coder 0.5B flagged 539 and 540 of 540 snippets, and four others (devstral 24B, codellama 34B, qwen2.5-coder 1.5B and 3B) flagged more than 70% of secure twins. The 0.5B model also omitted the requested verdict line in all but 37 of 540 replies. A benchmark without matched secure code would have ranked these models near the top.

Size is not the whole story. gpt-oss 20B had the best balanced accuracy, while codellama 34B and devstral 24B were among the weakest of the larger models. Within the qwen2.5-coder family accuracy rose from 0.5B to 32B, and over all 16 models size and balanced accuracy are clearly related (Spearman 0.76), but across families parameter count did not order the results.

A confounded first look. Comparing plain pairs with indirect pairs in this design, balanced accuracy was lower on indirect pairs in 14 of the 16 models; codellama 7B was slightly higher (0.504 against 0.500) and qwen2.5-coder 0.5B tied, both models that flag almost everything. Plain and indirect

TABLE II
STUDY 2: BALANCED ACCURACY ON THE SAME 80 PAIRS (240
VULNERABLE AND 240 SECURE CALLS PER MODEL AND FORM). CI:
PAIR-CLUSTER BOOTSTRAP 95% INTERVAL OF THE CHANGE.

Model	Plain	Indirect	Change	CI
qwen2.5-coder 32B	0.869	0.750	-0.119	[-0.173, -0.067]
llama3.3 70B	0.819	0.763	-0.056	[-0.119, 0.008]
qwen2.5-coder 14B	0.727	0.675	-0.052	[-0.106, 0.002]
gpt-oss 120B	0.944	0.894	-0.050	[-0.092, -0.010]
gpt-oss 20B	0.952	0.925	-0.027	[-0.073, 0.017]
qwen3-coder 30B	0.723	0.706	-0.017	[-0.071, 0.037]
codestral 22B	0.763	0.767	+0.004	[-0.058, 0.065]
Pooled	0.828	0.783	-0.045	[-0.076, -0.013]

snippets here are different pairs, so the indirect pairs might simply be harder. Study 2 compares the same pairs.

V. STUDY 2: THE SAME PAIR IN BOTH FORMS

A. Design

For 80 of the 90 pairs I wrote the opposite form of each snippet (so 45 originally plain pairs gained an indirect form and 35 pairs written indirectly (25) or alias-only (10) gained a plain form). I excluded ten pairs where I judged that the code had no well-known name to reveal or hide, so no opposite form could be written (for example, base64 used as encryption, or uuid1 used for a session identifier). This judgment was looser than the token rule used later: the 80 pairs still include 25 from classes with no standard token, where the two forms differ in where a literal key, IV, or salt choice sits rather than in a named API. I chose these exclusions, and the seven models (Study 1 bAcc of at least 0.70), after seeing Study 1. I ran the 160 new snippets with the same prompt, parser, settings, and three repeats (3,360 calls, no errors), so each model saw every pair in both forms. In 14 pairs the secure twin is byte-identical in both forms, because the correct code has nothing to move. These twins appear in both arms; on them the two sessions flagged 82 and 81 of 294 calls. Each session also ran both directions (45 pairs gained their indirect form in the second session and 35 their plain form), so a session effect would largely cancel in the pooled estimate.

Calls on the same pair are not independent, so all intervals in this section come from a bootstrap that resamples the 80 pairs (4,000 draws), and the pair-level test compares each pair’s accuracy pooled over models and repeats.

B. Results

Table II gives each model. Pooled over the seven models, balanced accuracy fell from 0.828 to 0.783, a drop of 4.5 points (95% CI 1.3 to 7.6). The plain form scored better on 46 pairs, the indirect form on 19, and 15 tied (sign test $p = 0.001$; sign-flip permutation $p < 0.01$). Both directions show the drop, though the second interval includes zero: 5.0 points on originally plain pairs (CI 0.6 to 9.3) and 3.9 on originally indirect pairs (CI -1.0 to 8.4). These intervals treat the seven models as fixed; resampling models as well gives an interval of about 0 to 9.

The effect is concentrated. Six of seven models scored lower on the indirect form, but only two intervals exclude zero, and qwen2.5-coder 32B alone lost 11.9 points. By class, short asymmetric keys dropped most (0.947 to 0.786), followed by ECB mode (0.795 to 0.714) and weak randomness (0.863 to 0.789). Without short keys the drop is 3.1 points (CI -0.2 to 6.2); without qwen2.5-coder 32B it is 3.3 (CI 0.1 to 6.2); without both it is 1.7 (CI -1.5 to 4.7, permutation $p = 0.31$). With the nine classes as the unit, 7 of 9 fell (exact sign-flip $p = 0.06$). I read the result as a modest pooled effect carried mainly by some models and classes, not as a uniform property of LLM reviewers.

Not every pair is a clean match. An audit of the stimuli after the runs found that in 33 of the 80 pairs the two forms differ in more than the route to the misuse. Some rewrites change the API itself (cv08 relies on the defaults of `ssl.SSLContext`, and sk05 selects a 192-bit curve by OID); some make the misuse depend on an environment variable; four load the module through `importlib`; two secure twins hold the bad token in an unused dictionary entry; some plain forms keep an unused leftover of the indirect form; and in five pairs (the certificate pairs cv01, cv02, cv04 and cv05, and the salt pair ns01) the indirect secure twin states a safe value that the plain twin leaves out. The release lists every id with its reason. On the 47 clean pairs balanced accuracy fell 5.3 points (CI 2.1 to 8.5; 26 pairs better in the plain form, 10 in the indirect form, sign test $p = 0.01$), so the headline does not depend on the flawed pairs.

After the audit, the effect is mainly false alarms. On all 80 pairs TPR fell from 92.0% [88.3, 95.3] to 88.4% [83.8, 92.6], a drop of 3.6 points (CI 1.0 to 6.5), and FPR rose from 26.4% [21.2, 31.7] to 31.8% [26.2, 37.6], a rise of 5.4 points (CI -0.1 to 10.8). The audit moves the two parts in opposite directions. The detection drop is 2.3 points (CI -0.1 to 4.9) without the six pairs whose rewrite changes the API and 1.5 points (CI -1.6 to 4.8) on the clean pairs. The false-alarm rise is 8.0 points (CI 3.1 to 13.2) without the four unmatched certificate pairs and 9.1 points (CI 4.2 to 14.6) on the clean pairs. On those four pairs the plain secure twins drew 36 flags in 84 calls (15, 0, 18, and 3) and the indirect twins none, but 33 of the 36 cited SSRF, input validation, or error handling rather than certificate checks, so I treat this as an artifact of unmatched twins and do not interpret it. The fifth unmatched pair, ns01, is not a certificate pair and stays in the 8.0-point figure. TPR counts a vulnerable verdict given for any reason, and I did not measure whether each detection named the right misuse.

The extra flags are concentrated, and many are mistakes. On the clean pairs three models carry the false-alarm rise: llama3.3 70B (+19.1 points), qwen2.5-coder 14B (+18.4), and qwen2.5-coder 32B (+17.7); the other four rose between 1.4 and 2.8 points. Over all 80 pairs, flags on secure twins rose in some pairs (185 more flags) and fell in others (94 fewer), a net rise of 91. By class, short keys (+49) and ECB mode (+31) rose most, and certificates fell by 33. The replies show why. Two short-key secure twins write a 3072-bit RSA key size as

0xC00 or $3 * 2 ** 10$; their plain versions with a literal 3072 drew no flags, the indirect versions drew 12 of 21 each, and all 24 flagging replies (four models, three repeats each) decoded the value correctly as 3072, 21 of them then called it too small, weak, or possibly insufficient (one model says it “is below the recommended minimum of 2048 bits”), two called it a non-standard size, and one cited the backend; the two plain twins are the same 3072-bit call under different function names. The ECB secure twin ecb04 selects GCM through the PyCryptodome constant 11; it drew 20 flags in 21 calls against 9 for its plain version, and 16 of the 20 replies called 11 an invalid, non-standard, or unspecified AES mode. In these three twins the stated reasons for the flags are wrong. A smaller set of flags is defensible, for example where a certificate or key-size twin takes its setting from an environment variable that a deployment could change.

Where the effect sits (exploratory). After reading the replies I split the pairs two ways. Both splits were chosen after seeing the results and they overlap, so I report them together as hypotheses for a confirmatory study. The first split asks whether the familiar token is still in the indirect vulnerable file. The six token classes (MD5 or SHA-1; ECB; obsolete cipher names; False, CERT_NONE or `_create_unverified_context`; short key sizes; the `random` module) hold 55 pairs; bc05, a hand-rolled XOR cipher with no cipher name, goes with the 25 no-token pairs. The token stays in 44 pairs and is gone in 10 (cv03, cv06, cv08, ecb04, ecb06, ecb10, sk01, sk04, sk05, sk08). Where it stays, balanced accuracy fell 0.9 points (CI -3.8 to 5.2); where it is gone, 23.3 points (CI 19.5 to 27.4), mostly through false alarms (FPR $+34.3$), because in nine of these ten pairs the secure twin also loses its familiar safe spelling. Without cv08 and sk05, which change the API, the gone group still drops 22.9 points. On detection alone the contrast does not hold: detection fell 2.3 points where the token stayed and 12.4 where it was gone, but without cv08 and sk05 the difference is 2.5 points (CI -2.3 to 7.4 , $p = 0.51$), and the largest losses of detection are in three weak-randomness pairs that keep `random` in the file (pr01 to pr03, 22 lost detections). The second split takes the nine pairs whose indirect form writes a parameter as a bare number or arithmetic expression (cv03, cv10, ecb04, ecb06, ecb10, ns03, sk01, sk04, sk08). They dropped 18.5 points (CI 11.1 to 25.4), but seven of the nine are also in the token-gone group, so the two splits cannot be separated. The encoding is not required: in sk02 a decimal 3072 held in a dictionary drew nine new flags from three models that call the size weak, too small, or possibly insufficient, and in sk06 one model called a constant 3072 weak; sk09, with 4096 in a dictionary, drew none. What the flagged secure twins share is that the safe value has left its usual spelling or place at the call site. The ten pairs that only wrap the API in an alias, a helper, or a lookup table dropped 0.2 points (CI -6.2 to 6.4), an interval too wide to read. Leaving out the ten token-gone pairs, the remaining 70 dropped 1.8 points (CI -1.4 to 4.7).

VI. DISCUSSION

For a team relying on an LLM reviewer, the practical reading is cautious. With the same task and the same misuse written two ways, routing a cryptographic choice through a constant, a default, or a configuration value made several of these reviewers less accurate, mostly by flagging correct code they misjudged, such as a key size written in hexadecimal or a mode selected by number. Missed misuse rose 3.6 points over all pairs, but the rise could not be distinguished from zero on the audited pairs. The flags sit where the safe value has left its usual spelling or place at the call site: models that pass a literal 3072 at the call site flag it when it is written as 0xC00 (four models) or held in a constant or dictionary (three models, two of them among the four). The replies describe a threshold error or an unknown constant, but model explanations are not reliable evidence of mechanism [6], so I do not claim one. The pattern fits partial reliance on familiar spellings, as renaming results for LLMs and models of code suggest [6], [15], but my design cannot separate that from the added difficulty of unusual code.

The broader lesson is about measurement. A ranking by TPR alone can be badly wrong (Study 1), and a benchmark without indirect variants and matched secure twins would miss most of the false-alarm effect reported here. Benchmarks for LLM security review should include both, report balanced accuracy with intervals that respect clustering, and audit pairs for differences beyond the one intended.

VII. LIMITATIONS

The snippets are short Python functions reviewed one at a time, so the results speak to a single-snippet review pass. Thirty-three of the 80 pairs are not clean matches; I report results with and without them, and the audit came after the runs. Both exploratory splits were chosen after seeing the results, and the design has no neutral-rewrite arm that changes the form of the code without moving the familiar spelling. Inside the token-gone group some pairs share a manipulation: three select an AES mode by the integer 11 and two write the 3072-bit key size as 0xC00 or $3 * 2 ** 10$. Together they give 49 of the group’s 72 extra flags, so the pair bootstrap treats them as more independent than they are and the group interval is too narrow. With 47 audited pairs the standard error of the detection change is about 1.6 points, so a drop near 5 points cannot be ruled out. TPR counts a vulnerable verdict given for any reason; a right-reason analysis could move the detection figure in either direction, and I did not do one. The prompt asks about any security vulnerability, so some flags on secure twins cite issues outside the target misuse. Study 2 covers seven open-weight models chosen after Study 1, the intervals treat them as fixed, and the ten exclusions were decided after Study 1. The counterpart snippets were written after the Study 1 results were known, and half of the paired rows are Study 1 rows that also selected the seven models. Some plain snippets are shared across pairs (seven groups, for example the secure twins of bc01 and bc02); keeping one of each gives a 4.6-point drop (CI 1.2 to 7.8). In Study 2, 97% of model and

snippet cells gave the same verdict on all three repeats, so the repeats add little beyond the first call; in Study 1, between 0 and 32 of the 180 snippets per model changed verdict across repeats, because Ollama batches requests in parallel. In the indirect form a pair’s two snippets are about 75 characters longer in total, but length difference does not predict a pair’s drop (Pearson $r = 0.03$). The v2 data files tag the static-IV class as CWE-329, which is specific to CBC; the release uses CWE-1204. Finally, I am the only author; the snippets and labels went through a writing pass and separate review passes, carried out with AI assistance and checked by me, which reduces but does not remove labeling risk.

VIII. CONCLUSION

On 80 pairs seen in two forms by seven open models, writing a cryptographic misuse indirectly lowered balanced accuracy by 4.5 points (CI 1.3 to 7.6), and by 5.3 points (CI 2.1 to 8.5) on the 47 pairs that survive a stimulus audit. The effect holds with pairs as the unit and is weaker with classes as the unit (7 of 9 fell). After the audit, most of it is extra flags on correct code, concentrated in three of the seven models. Detection of real misuse fell 3.6 points over all pairs (CI 1.0 to 6.5) but only 1.5 points on the audited pairs (CI -1.6 to 4.8). Exploratory splits place most of the net false-alarm rise in pairs where the safe value has left its usual spelling or place at the call site; they were chosen after seeing the results and need a confirmatory study. CryptoBench v1 is public [3], and I will release the v2 pairs in both forms, the audit list, all per-call results, and the analysis scripts with this paper.

ACKNOWLEDGMENT

I used an AI assistant (Claude, Anthropic) at three levels in this work. It drafted many of the benchmark snippets and their secure twins (Sections III and V), which I reviewed and labeled. It wrote the analysis scripts behind Sections IV and V and checked the numbers and references. It drafted most of the text in every section, which I revised. I reviewed all snippets, labels, results, and claims and take full responsibility for them.

REFERENCES

- [1] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in Android applications,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2013, pp. 73–84, doi: 10.1145/2508859.2516693.
- [2] S. Rahaman et al., “CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2019, pp. 2455–2472, doi: 10.1145/3319535.3345659.
- [3] S. Chokshi, “CryptoBench: A benchmark of cryptographic API misuse for evaluating LLM code reviewers,” version 1.0.0, Zenodo, Sep. 30, 2026, dataset, doi: 10.5281/zenodo.23067052.
- [4] S. Chokshi, “Known-bad names, unknown-bad uses: What local code models detect when they review cryptographic API misuse,” preprint, Zenodo, Oct. 3, 2026, doi: 10.5281/zenodo.23113861.
- [5] S. Krüger et al., “CogniCrypt: Supporting developers in using cryptography,” in *Proc. 32nd IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2017, pp. 931–936, doi: 10.1109/ASE.2017.8115707.
- [6] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, “LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks,” in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2024, pp. 862–880, doi: 10.1109/SP54263.2024.00210.

- [7] N. Risse and M. Böhme, “Uncovering the limits of machine learning for automatic vulnerability detection,” in *Proc. 33rd USENIX Security Symp.*, 2024, pp. 4247–4264. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/risse>
- [8] Y. Ding et al., “Vulnerability detection with code language models: How far are we?,” in *Proc. IEEE/ACM 47th Int. Conf. Software Engineering (ICSE)*, 2025, pp. 1729–1741, doi: 10.1109/ICSE55347.2025.00038.
- [9] Y. Li et al., “Everything you wanted to know about LLM-based vulnerability detection but were afraid to ask,” arXiv:2504.13474, 2025.
- [10] E. Firouzi, M. Ghafari, and M. Ebrahimi, “ChatGPT’s potential in cryptography misuse detection: A comparative analysis with static analysis tools,” in *Proc. 18th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM)*, 2024, pp. 582–588, doi: 10.1145/3674805.3695408.
- [11] Z. Masood and M. Vargas Martin, “Beyond static tools: Evaluating large language models for cryptographic misuse detection,” arXiv:2411.09772, 2024.
- [12] Y. Xia et al., “Beyond static pattern matching? Rethinking automatic cryptographic API misuse detection in the era of LLMs,” *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 113–136, 2025, doi: 10.1145/3728875.
- [13] H. Baek, M. Lee, and H. Kim, “CryptoLLM: Harnessing the power of LLMs to detect cryptographic API misuse,” in *Computer Security – ESORICS 2024*, ser. Lecture Notes in Computer Science, vol. 14982. Cham, Switzerland: Springer, 2024, pp. 353–373, doi: 10.1007/978-3-031-70879-4_18.
- [14] R. Geirhos et al., “Shortcut learning in deep neural networks,” *Nature Machine Intelligence*, vol. 2, no. 11, pp. 665–673, 2020, doi: 10.1038/s42256-020-00257-z.
- [15] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, Nov. 2020, doi: 10.1145/3428230.
- [16] E. B. Wilson, “Probable inference, the law of succession, and statistical inference,” *J. Amer. Statist. Assoc.*, vol. 22, no. 158, pp. 209–212, 1927, doi: 10.1080/01621459.1927.10502953.