Automatically Learning Vulnerability Patterns for Scalable Static Analysis of Web Applications

Penghui Li
Columbia University
New York, USA
Nanyang Technological University
Singapore
penghui.li@ntu.edu.sg

Songchen Yao
Columbia University
New York, USA
sy2743@columbia.edu

Josef Sarfati Korich
Columbia University
New York, USA
jsk2266@columbia.edu

Changhua Luo
Independent Researcher
Wuhan, China
chdeluo@gmail.com

Jianjia Yu
Johns Hopkins University
Baltimore, USA
jyu122@jhu.edu

Yinzhi Cao
Johns Hopkins University
Baltimore, USA
yinzhi.cao@jhu.edu

Junfeng Yang
Columbia University
New York, USA
junfeng@cs.columbia.edu

Abstract

Static analysis has been widely used to analyze web applications, yet its effectiveness largely relies on the quality of vulnerability patterns. In this work, we present MoCQ, which leverages large language models (LLMs) to automatically generate executable detection queries that capture these vulnerability patterns. To address challenges like LLM hallucinations and the complexity of industrial static analysis tools, MoCQ introduces a DSL-based harness. This harness extracts and models a core subset of domain-specific languages (DSLs) to provide the formal structure and guidance needed to generate valid queries. MoCQ employs an iterative refinement loop with trace-driven symbolic validation that provides precise feedback for query correction and optimization. Such a neuro-symbolic approach thus combines the scalability of pattern-based static analysis with the semantic understanding and automation capabilities of LLMs. We evaluated MoCQ on multiple vulnerability types across web programming languages, including Java, PHP, and JavaScript. MoCQ achieves detection performance comparable to expert-developed patterns while requiring only hours of query generation versus weeks of manual effort. Notably, MoCQ uncovered 45 new vulnerability patterns that security experts had missed and discovered 25 previously unknown vulnerabilities in real-world applications. MoCQ also outperforms prior approaches with stronger analysis capabilities.

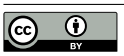
CCS Concepts

• **Security and privacy → Web application security**.

Keywords

Static Vulnerability Detection; Web Application Security; Neuro-Symbolic Methods

ACM Reference Format:
Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. 2026. Automatically Learning Vulnerability Patterns for Scalable Static Analysis of Web Applications. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846556



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846556

1 Introduction

Pattern-based static analysis has become a cornerstone of vulnerability detection for web applications. Compared to dynamic techniques [9, 28, 63], it scalably analyzes large codebases with high code coverage and requires minimal setup effort (*e.g.*, no need for software deployment). It has thus been widely adopted in production systems [7, 59, 60, 72] and has discovered numerous high-impact vulnerabilities (*e.g.*, Log4Shell [45, 57]). For instance, CodeQL [13] is extensively integrated into GitHub's CI/CD pipelines [53]. Joern [7, 72] has been recognized as one of the most widely used tools for automated vulnerability discovery [61], and has detected many zero-day vulnerabilities on the web [7, 59, 60, 72].

The effectiveness of static approaches relies on *the quality of vulnerability patterns*. These patterns describe vulnerable code structures or behaviors and are used to match new vulnerability instances in target codebases. However, the creation of these patterns is largely manual and often requires in-depth expertise in both security problems and underlying analysis tools. This process is not only time-consuming (*e.g.*, several weeks or more [2, 22, 40]), but also prone to inaccuracies due to limited human knowledge, as evidenced by real-world issues in existing detection patterns [1, 3, 46]. Furthermore, as codebases evolve and new threats emerge, each

new vulnerability class requires substantial manual effort and expertise to develop effective detection patterns, making it increasingly difficult to keep pace with the evolving threat landscape.

Researchers have alternatively explored learning-based approaches that statically analyze source code. Most learning-based methods operate at the function level to classify individual functions as vulnerable or benign. However, they often lack *interprocedural or project-wide* context [36, 55], and require a large volume of labeled vulnerabilities for training [11], which may not be available. More recently, several works [18, 39, 76] have used Large Language Models (LLMs) to directly analyze code without relying on traditional program analysis. Leveraging massive training corpora, LLMs can recognize typical (vulnerable) usage patterns and reason about code semantics across diverse programming contexts. While promising, LLMs face challenges such as scalability, token length limits, and susceptibility to hallucinations.

The distinct challenges of classic and learning-based methods suggest that a hybrid neuro-symbolic approach[1] is a promising direction. Specifically, we can use LLMs to automatically generate vulnerability patterns that can be leveraged by static tools for scalable vulnerability analysis. While recent works have taken steps toward such integration for web security, *they fall short of the automation level or analysis capability*. IRIS [34] and Artemis [25] use LLMs to extract taint sources and sinks, and incorporate them into existing detection patterns for Java and PHP applications, respectively. *They automate only parts of the detection logic*, while the core vulnerability patterns must still be manually crafted by security experts. For instance, IRIS expects experts to manually design the complete data-flow tracking logic and control-flow conditions in CodeQL queries, leaving only placeholder slots for sources/sinks to be filled by the LLM. This means that for each new vulnerability type, experts must first spend days to weeks developing the base query structure before IRIS can contribute.

In this paper, we design MoCQ (Model-Generated Code Queries), which leverages LLMs to automatically generate *complete end-to-end vulnerability detection patterns, including sources, sinks, vulnerable operations, control-flow logic, and sanitization checks, rather than just taint specifications*. It incorporates these LLM-generated patterns into established static analysis platforms (*e.g.*, Joern), preserving their scalability while automating what was previously labor-intensive. MoCQ is not limited to generating patterns from scratch for new vulnerability classes; it can also automatically optimize and refine existing legacy patterns. MoCQ supports multiple programming languages and handles complex analysis tasks within a unified framework.

However, such a pattern generation solution faces multiple challenges. LLMs often do not have enough knowledge of domain-specific languages (DSLs) used by static vulnerability detection tools (*e.g.*, Scala-like DSL for Joern [72] and SQL-like DSL for CodeQL). Specifically, our study shows that even state-of-the-art models like Claude Sonnet 4.6 often fail to generate valid queries, frequently producing syntax errors or triggering execution-time exceptions. To deal with this challenge, *we introduce a DSL-based harness that guides the query generation process*. This harness serves as the structure that restricts the LLM to a valid search space by specifying the formal grammar, API signatures, and semantic constraints of the query DSL. We observe that while modern DSLs comprise thousands of APIs, they are inherently redundant. MoCQ thus employs *DSL subsetting* to extract a core, high-expressivity set of APIs and specifications as the harness. By operating within this harness, MoCQ significantly reduces syntax errors and hallucinations while ensuring that the generated queries remain strictly compatible with the underlying static analysis engine.

Even when the LLM-generated queries are syntactically correct and executable, they may fail to capture vulnerabilities effectively. Specifically, they may trigger semantic errors (*i.e.*, fail to report vulnerabilities), be overly specific (*i.e.*, overfitting to known instances via matching exact variable names or fixed control flow paths), or be overly general (*i.e.*, underfitting by matching many irrelevant code cases). To deal with these, MoCQ adopts a *feedback-driven* approach, which incorporates fine-grained feedback from a symbolic query validator for iterative query refinement. The query validator executes the LLM-generated queries and monitors the exhibited behaviors, including syntax errors, execution exceptions, and semantic errors. Specifically, we instrument the query execution runtime to obtain a block-level program state of the query execution to precisely locate the errors and inconsistencies. These are then used for the LLM to refine the incorrect query. The validator further uses heuristics to detect if a query is too specific and instructs the LLM to generalize it; it also instructs LLMs to eliminate false positives (if any) to improve the precision.

We extensively evaluated MoCQ on representative vulnerability types from the OWASP Top 10 [51] (*e.g.*, deserialization, prototype pollution) across three popular web programming languages (Java, PHP, and JavaScript). MoCQ efficiently generates detection queries within hours from only a few vulnerability examples. Compared with directly prompting Claude Sonnet 4.6 to generate queries, MoCQ powered by the same model achieves a 121% higher success rate in producing valid detection queries. This advantage persists across both stronger frontier models (Claude Opus 4.6 and 4.8) and a smaller open-source backbone (Qwen3.6-35B), indicating that MoCQ's benefit stems from its framework rather than the underlying model alone. On the detection side, we compare MoCQ-generated queries against multiple baselines, including agentic approaches such as Claude Code [4], classic vulnerability-type-specific detectors, and queries manually developed by experts. MoCQ-generated queries demonstrate competitive detection capability across all these approaches. Our ablation study shows that MoCQ's framework remains necessary for query generation across different backbone LLMs, and that it can complement them to further improve overall effectiveness. MoCQ also successfully detected 25 new vulnerabilities in real-world applications.

We further manually analyzed patterns generated by MoCQ and those developed by human experts. Our analysis shows that MoCQ uncovers 45 new vulnerability patterns missed by experts, with 11 patterns already integrated into upstream tool repositories. Each missed pattern can correspond to an entire class of overlooked vulnerabilities. We also observe complementary strengths between LLMs and human experts. LLMs excel at capturing subtle linguistic and syntactic variations that experts may overlook, while experts

[1]Neuro refers to learning-based analysis (particularly LLMs) leveraging neural models for semantic reasoning; symbolic refers to classic static analysis using symbolic representations and formal logic systems.

demonstrate stronger high-level reasoning for complex vulnerabilities. By combining both approaches, we detect more vulnerabilities than either alone.

This paper makes the following contributions.

- We propose a scalable static analysis framework for web applications that combines the complementary strengths of LLMs and classic program analysis.
- We develop MoCQ, which uses a DSL-based harness to automate end-to-end vulnerability pattern generation.
- We demonstrate that MoCQ achieves detection performance comparable to state-of-the-art approaches, with 25 new vulnerabilities discovered in real-world applications.
- We systematically compare expert-developed and LLM-generated patterns, identifying 45 new vulnerability patterns and highlighting their complementary strengths.

2 Background and Motivation

2.1 Pattern-based Static Analysis for Web Applications

Web applications expose many functionalities that process user inputs, such as form submissions and user interactions. Security vulnerabilities arise when such inputs are used in critical operations without proper sanitization. Common examples include cross-site scripting (XSS) [20] and SQL injection (SQLi) [63]. Attackers exploit these flaws by injecting malicious inputs to alter program behavior or execute unintended operations.

Pattern-based static analysis is a widely used approach for detecting vulnerabilities in web applications. Tools such as Joern [7, 72] and CodeQL [13] express vulnerability patterns as *structured queries* written in DSLs. They use language-specific frontends to translate source code into graph-like program representations such as control-flow graphs and code property graphs (CPG) [72]. Security analysts then write precise queries that operate on these representations to identify vulnerabilities. By separating analysis logic (queries) from language-specific semantic reasoning, this approach offers high flexibility and easier maintenance. Joern models programs with CPGs [48], while CodeQL relies on code databases. The DSLs differ as well. Joern employs a Scala-like language, whereas CodeQL uses SQL-like syntax. Static analyzers enforce strict syntax and semantic checks during query execution to ensure accuracy and soundness. **An Example.** JavaScript-based web applications suffer from prototype pollution vulnerabilities. Specifically, JavaScript is a prototype-based language where objects can inherit properties and methods from other objects through a mechanism called *prototypes*. Every object has a hidden link to a prototype object, which provides shared properties and methods. Such a hierarchical object inheritance forms a prototype chain. When accessing an object's property, JavaScript first checks if the property exists. If not, it looks up the prototype chain until it finds the property or reaches the end of the chain. Since objects inherit properties from their prototype, modifying a property on the prototype affects all objects that inherit from it. For example, an attacker can access the prototype via `p=obj["__proto__"]` and modify it via `p["toString"]="Hacked"`. As a result, any object inheriting from the fundamental JavaScript prototype `Object.prototype` now has a tampered `toString` method.

```
1  // find obj["__proto__"] in p=obj["__proto__"]
2  def objProto = cpg.call
3    .where(_.name(Operators.assignment)).argument(2)
4    .isCall.arrayAccess
5
6  // find p["toString"] in p["toString"]="Hacked"
7  def objProp = cpg.call
8    .where(_.name(Operators.assignment)).argument(1)
9    .isCall.arrayAccess
10
11 // find a data-flow path from objProp to objProto
12 objProp.array.reachableBy(objProto).filter(...)
```

Listing 1: Example real-world query in Joern, simplified for clarity.

This type of vulnerability can lead to arbitrary code execution, privilege escalation, and denial of service [15, 27, 33, 37, 59].

Listing 1 shows a simplified version of a real-world Joern query designed to detect JavaScript prototype pollution [22]. It comprises three main steps: (1) identifying the object prototype (lines 1-4), (2) identifying the property access (lines 6-9), and (3) verifying whether there is a data connection between them (lines 11-12). For example, step (1) uses Joern's dot operator (`"."`) to chain operations, where each stage filters results produced by the previous one. It begins by selecting assignment expressions via `Operators.assignment` (`assignment` is modeled as a form of `call` operation in Joern), then extracts the right-hand side via `argument(2)`, and narrows results to array-style accesses using `arrayAccess()` to recover `obj["__proto__"]` from `p=obj["__proto__"]`. Step (2) mirrors this structure but extracts `argument(1)` instead of `argument(2)` to recover `p["toString"]`. Step (3) then uses Joern's data-flow API to check whether the base object from step (2) is reachable by the prototype reference from step (1), establishing that `p` and `obj["__proto__"]` refer to the same underlying object.

2.2 Research Goals

High-quality, well-crafted patterns are essential for pattern-based static analysis. Without them, even powerful analysis engines may miss critical vulnerabilities or generate excessive false positives. These patterns are typically developed manually by experts. In practice, even experienced experts may overlook subtle attack variants and produce imprecise patterns [1, 46]. For instance, attackers can bypass existing patterns in Listing 1 by combining property access and assignment in a single expression (*e.g.*, `obj["__proto__"]["toString"]="Hacked"`). More broadly, Listing 1 only recognizes prototype modification expressed as an `Operators.assignment` call, so it also misses semantically equivalent property writes performed through other JavaScript APIs, such as `Object.assign(obj["__proto__"], {toString: "Hacked"})` or `obj["__proto__"].__defineGetter__("toString", () => "Hacked")`, both of which mutate the prototype without ever appearing as a top-level assignment expression. MoCQ later identifies exactly these two API-based bypasses as new patterns missed by expert-written queries (§5.4). These challenges highlight the need for automated and reliable vulnerability pattern generation.

Motivated by the strong capabilities of LLMs, this work explores automatically generating vulnerability patterns for web application static analysis. Our goal is to reduce the reliance on manual pattern engineering while maintaining high precision and coverage. This approach enables a neuro-symbolic analysis paradigm that can scale to large codebases. In particular, when new threats are

disclosed, developers can rapidly generate accurate and comprehensive patterns for large-scale code scanning, without incurring the significant delays and expertise costs of manual development. It can also help refine existing detection queries for better precision.

2.3 Research Challenges

Inspired by the promising results of LLMs in general-purpose code generation, we explore using them to generate vulnerability detection queries. However, doing so presents substantial challenges. To illustrate, we task an LLM (specifically Claude Sonnet 4.6 [6]) with generating a Joern query to detect JavaScript prototype pollution. The model is prompted in a few-shot setting: we provide (1) two code snippets that contain known prototype pollution vulnerabilities, and (2) several example Joern queries as demonstrations of the target query language and style. We further instruct the model to produce the final query following step-by-step reasoning, with the goal of encouraging structured synthesis. Despite this guidance, the generated query fails to execute in Joern's analysis engine. Below, we present the failed query and walk through the three challenges it surfaces, in the order we encountered them. First, a syntax error prevents the query from parsing at all. After fixing the syntax, the query parses but crashes during execution with a runtime error. After fixing that in turn, the query finally executes without crashing, but a semantic error causes it to silently return no results, missing the known vulnerability entirely.

Incorrect LLM-generated query:

```
1 ...
2 val assign = code.call // omitting `cpg`
3   .nameExact("assignment") // using wrong operator name
```

DSL Syntax Correctness. The LLM-generated queries frequently raise *grammar and syntax errors*, which stem from the model's incomplete understanding of the DSL grammar. Specifically, LLMs typically lack sufficient exposure to the syntax, semantics, and typical usage of complex DSLs. For example, Joern queries based on the code property graph usually require the root object `cpg`. However, the query generated above did not follow this requirement. This issue is further aggravated by the hallucinations of LLMs.

Runtime Execution Correctness. Queries must execute within the domain-specific runtime (*i.e.*, analysis engine) and interact meaningfully with structured program representations. LLM-generated queries often use incorrect or non-existent function calls, missing parameters, and incompatible data types or operations, resulting in *execution-time exceptions*. For example, the `nameExact` operation in the above query used `"assignment"` instead of `"<operator>.assignment"`, which led to execution-time exceptions after we first manually fixed the syntax error.

Semantic Validity. Even when queries are syntactically correct and executable, ensuring their semantic validity remains challenging. They must satisfy runtime constraints and capture the correct traversal logic. Executable queries may still fail to reflect intended program behaviors, leading to *semantic errors*, where the query runs without crashing but misses the vulnerable code. Indeed, even after we corrected the syntax and runtime errors above, the resulting query still failed to retrieve the known-vulnerable example. Matching the right operator name alone does not confirm that the matched assignment is actually connected to a property access via a data-flow path.

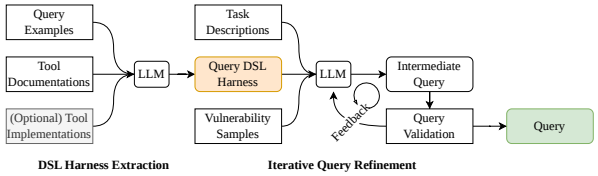


Figure 1: The workflow of MoCQ. MoCQ first extracts a query DSL harness once per static analysis tool. This harness then guides an LLM to generate and iteratively refine a detection query using validation feedback.

3 MoCQ

We design MoCQ, a scalable static analysis system for web applications. Its workflow is outlined in Figure 1. The key idea is to first concisely summarize the DSL of the static analysis engine as the harness to guide query generation, and then provide the LLM with fine-grained runtime feedback to iteratively debug invalid queries. Specifically, MoCQ first analyzes open knowledge sources, such as documentation and tool implementations, to extract the query DSL (§3.1). We derive a subset of core, high-expressivity DSL constructs as the harness, without overwhelming the LLM with the full DSL complexity. MoCQ then generates vulnerability queries through an iterative feedback loop, where a trace-driven symbolic query validator refines (§3.2) and optimizes (§3.3) the queries. Finally, MoCQ outputs the generated query, which can be directly applied to detect vulnerabilities in real-world codebases. It is worth highlighting that, beyond the DSL harness, MoCQ requires only a few vulnerability examples and can automatically generate effective detection queries through iterative refinement and feedback. We summarize two key techniques that address the aforementioned challenges. The DSL harness supplies the syntactic and semantic knowledge a query needs before the generation even begins, while the iterative feedback loop then debugs and refines that generated query against runtime execution and semantic signals.

Query DSL Harness. To enable syntactically and semantically correct query generation, we first obtain formal specifications of the query DSLs used by the static analysis tools, which also define the expected output structure for the LLM. MoCQ extracts DSL specifications, including grammar rules, data types, supported APIs, and their functionalities, by parsing online documentation and tool implementations. However, directly using the full DSL would overwhelm the LLM due to its complexity and would not guarantee efficient generation of valid queries. We thus propose *DSL subsetting*, which selects a compact, high-expressivity subset of DSL features from the full specification. This is motivated by the observation that DSLs often contain redundant or interchangeable constructs that are not essential for common vulnerability detection tasks. This subset improves interpretability for the LLM and significantly reduces query generation complexity.

Iterative Feedback Loop. Built on top of the DSL harness, MoCQ employs an LLM to generate queries and iteratively refines them

using fine-grained feedback from a symbolic query validator. Specifically, we instrument the query execution runtime to execute generated queries and identify the exact sources of syntax and semantic errors. In addition, the validator tracks intermediate execution states and checks whether the generated query successfully retrieves the provided vulnerability examples. The resulting feedback is used to guide the LLM in debugging and refining the query. Through this iterative process, MoCQ gradually produces valid queries capable of analyzing complex real-world programs. Moreover, the validator incorporates heuristics to detect potential overfitting and encourages generalization, helping reduce false positives in the generated queries.

3.1 DSL Harness Extraction

To help LLMs generate syntactically and semantically valid queries, MoCQ extracts formal specifications of the query DSLs used by static analysis tools. Rather than relying on few-shot learning (which provides only a small number of query examples) or fine-tuning (which requires large-scale labeled data), we guide the model using a concise DSL specification, which we call the DSL harness. This specification includes both the *grammar*, which defines the structural rules of the query language, and the *semantics*, which describe the behavior of APIs, data types, and parameters.

3.1.1 DSL Extraction. To extract the DSL, MoCQ first analyzes the static analysis tool's documentation (*e.g.*, online use instructions) and source code (*e.g.*, API implementations and comments). Documentation typically contains structured content (*e.g.*, HTML tables) that describes grammar, operations, and usage patterns. We first crawl the online documentation of the tools and prompt an LLM to extract grammar rules, data types, operations, and API usage. For example, Joern's website [47] lists the query operations (termed "Steps") along with their descriptions. Beyond the brief APIs' behaviors in the documentation, we further seek more precise definitions from their code implementations, which offer ground-truth definitions through function signatures, type annotations, inline comments, *etc.* Based on the results of document analysis (*e.g.*, keywords, API name), we optionally prompt an LLM to retrieve relevant source code files from each tool's implementation, then to automatically summarize the functionality of the related functions. In fact, MoCQ primarily relies on a tool's documentation for extraction, which often includes a large number of demonstration query examples. In the extreme case where no documentation is available either, MoCQ can also extract the DSL harness from existing query examples collected from public repositories and tool distributions.

On the other hand, the DSL also includes language-specific constructs. Vulnerabilities often depend on such constructs of programming languages, which are not fully captured by the analysis tool's DSL abstractions. For example, in JavaScript, assignments and dynamic property accesses are essential for prototype pollution (*e.g.*, `Operators.assignment` in Listing 1). Therefore, MoCQ also enumerates these language-specific operations.

3.1.2 DSL Subsetting. The extracted query DSL specification is complex and could overwhelm the model. For example, static analysis tools like Joern and CodeQL have thousands of APIs. Each API can have multiple parameter choices, resulting in an exponential number of combinations of these rules and operations. Even though MoCQ knows the precise description for each operation, our evaluation (details in §5.5) demonstrates that providing LLMs with the full-set DSL does not efficiently produce good queries.

$$
\begin{aligned}
\langle \textit{traversal} \rangle &::= \langle \textit{cpg\_start} \rangle \text{"."} \langle \textit{step\_chain} \rangle \\
\langle \textit{cpg\_start} \rangle &::= \text{"cpg"} \mid \text{"cpg.method"} \mid \text{"cpg.call"} \mid \ldots \\
\langle \textit{step\_chain} \rangle &::= \langle \textit{step} \rangle \mid \langle \textit{step} \rangle \text{"."} \langle \textit{step\_chain} \rangle \\
\langle \textit{step} \rangle &::= \langle \textit{filter\_step} \rangle \mid \langle \textit{complex\_step} \rangle \\
\langle \textit{filter\_step} \rangle &::= \text{".where("} \langle \textit{predicate} \rangle \text{")"} \\
\langle \textit{complex\_step} \rangle &::= \text{".filter("} \langle \textit{predicate} \rangle \text{")"} \\
&\quad \mid \text{".reachableBy("} \langle \textit{traversal} \rangle \text{")"} \\
\langle \textit{predicate} \rangle &::= \text{"\_.name("} \langle \textit{reference} \rangle \text{")"} \mid \langle \textit{identifier} \rangle \\
\langle \textit{reference} \rangle &::= \text{"Operators."} \langle \textit{operator} \rangle \\
\langle \textit{operator} \rangle &::= \text{"assignment"} \mid \text{"arrayAccess"} \mid \text{"fieldAccess"} \mid \ldots
\end{aligned}
$$

Figure 2: Simplified BNF grammar for Joern, extracted by MoCQ.

We design a *DSL subsetting* technique to resolve the DSL complexity issue. We observe that the DSL specification contains significant redundancy not strictly necessary for constructing useful queries. A small, expressive subset is often sufficient for most vulnerability detection tasks. Instead of exposing the entire DSL, we restrict query generation to this smaller, more tractable subset, which reduces complexity and improves accuracy. Concretely, the subsetting process operates at the granularity of APIs, which can strike a balance between expressiveness and control: it preserves flexibility for query construction while keeping the DSL surface concise and understandable. MoCQ first associates typical parameter choices with each API. It then analyzes the full set of DSL APIs and prompts an LLM to decide and select these important, necessary DSL APIs for common usage. This is feasible since the LLM can refer to features' intended behaviors or functionalities to evaluate if one is necessary. We also perform a lightweight manual audit to verify the quality of the resulting DSL subset.

Figure 2 presents a simplified grammar for Joern using the Backus-Naur Form (BNF) [42]. It details the fundamental grammar rules, such as the `cpg_start` that initializes a `traversal` query, and how various components connect to form complete expressions. The grammar rules explicitly define the structure and permissible compositions of the DSL, guiding models to produce syntactically valid statements. For example, by following the grammar, an LLM knows to begin queries with `cpg` followed by appropriate method chaining through `step_chain` elements. The grammar also clarifies how to construct complex queries using operations like `filter_step` and `complex_step`, enabling more sophisticated code analysis patterns.

Note that this DSL harness extraction process is fully automated for both Joern and CodeQL. It is performed once per tool, and the resulting DSL harness is reused for all subsequent query generation across vulnerability types.

3.2 Query Generation and Refinement

Atop the core DSL subset, MoCQ generates detection queries as described in Algorithm 1. For each run, MoCQ is tasked with producing a query Q^t that targets a specific vulnerability type t. The

Algorithm 1: Iterative query generation in a feedback loop.

```
   Input  : Task description D^t, vulnerability examples E^t
   Output : Query Q^t
1  QList ← []
2  foreach e_i^t ∈ E^t do
3      PState ← null
4      for attempt ← 1 to MaxN do
5          Q_i^t ← LLMQueryGen (e_i^t, D^t, PState)
6          error, PState ← Validate (Q_i^t, e_i^t)
7          if not error then
8              Q_i^t ← Optimize (Q_i^t)
9              QList.add(Q_i^t)
10             break
11         end
12     end
13 end
14 Q^t ← ⋃_{q∈QList} q // Merge per-example queries
15 return Q^t
16
17 function Validate (Q, e):
18     S ← null
19     foreach B ∈ Q do
20         S ← Execute (B, e, S) // Capture fine-grained runtime info.
21     end
22     return S.err, S.State
23 end
```

generator takes as input (1) a task description D^t, which provides a natural-language explanation of the vulnerability type, and (2) a small set of representative vulnerability examples $E^t = \{e_i^t \mid i = 1, \ldots, m\}$. These examples help LLMs learn the semantic characteristics of the vulnerability type and also serve as ground-truth test cases for validating generated queries. Such vulnerability examples are readily available in practice from public sources, including CVE databases [44], security-related Git commits, and patches [73], making MoCQ easily deployable in real-world environments and use cases. MoCQ constructs a program slice for each vulnerability example to capture its relevant data and control dependencies. To make generation more tractable, MoCQ initially produces a per-example query Q_i^t that is expected to retrieve the corresponding example e_i^t. The system then performs optimizations to address overfitting and underfitting (line 8). Finally, the system merges all per-example queries into a unified per-type query ($Q^t = \bigcup_{i=1}^{m} Q_i^t$) for effective real-world vulnerability detection (line 14).

The DSL harness is provided to MoCQ but omitted from the pseudocode for clarity. Concretely, we present the core DSL harness to the LLM to guide query generation. We adopt a *grammar prompting* technique [66] to concisely express the grammar. Specifically, we supply an abstract grammar specification of the DSL using BNF [42], which is a standard way of defining language syntax. Although BNF is itself domain-specific, it appears more frequently in training data than custom query DSLs. Moreover, BNF structure is simpler and more concise than real query examples, while still conveying rich information about DSL usage. We also specify the subset semantics like APIs to the LLM, including their signature, data types, and API functionalities.

3.2.1 Vulnerability Analysis. As an initial step, MoCQ analyzes the input vulnerability example to understand its root cause and detection requirements. Given a vulnerability example e_i^t, MoCQ prompts an LLM to systematically examine the vulnerable code and identify the key security-relevant elements. MoCQ applies a chain-of-thought prompting strategy to decompose the vulnerability analysis into interpretable components. This analysis phase extracts essential vulnerability characteristics, such as the dangerous operations, data flow paths, and missing sanitization that inform subsequent query generation.

Based on this understanding, MoCQ creates a structured detection plan that breaks down the overall task into smaller, well-defined subtasks. For example, to detect prototype pollution vulnerabilities, the plan includes key steps such as: (1) identifying object property modifications, (2) tracking user-controlled data flow to property names, (3) checking for dangerous property values (*e.g.*, `__proto__`), and (4) verifying sanitization presence. Each subtask corresponds to a specific aspect of the vulnerability pattern and can be systematically translated into DSL operations. Importantly, MoCQ performs this decomposition automatically without requiring manual specification of the detection steps. This structured plan not only makes query generation more manageable but also provides a clear framework for iterative refinement when validation feedback reveals errors or incomplete coverage.

3.2.2 Query Refinement with Symbolic Validation. Given the difficulty of query generation, it is almost impossible to produce a qualified query in a single LLM attempt, even with the DSL. Therefore, MoCQ incorporates an iterative feedback loop (shown as the `for` loop on lines 4-12 of Algorithm 1) to debug and refine the query. At each iteration, MoCQ executes the generated query and validates it across syntax compliance, execution correctness, and semantic accuracy. When validation fails, MoCQ captures fine-grained feedback such as error locations, intermediate program states, and provides this information to the LLM for refinement and further fixes. This process terminates either when a valid query is generated or when the maximum attempt threshold is reached.

To enable precise debugging, MoCQ represents each query Q_i^t as a sequence of n blocks $[B_1, B_2, \ldots, B_n]$, where each block corresponds to a detection subtask. By instrumenting the query execution runtime, MoCQ records intermediate program states after executing each block B_j, formulated as $S_j = \{(var, val_j) \mid var \in B_{\leq j}\}$, capturing in-scope variables and their runtime values. These annotated program states $PState = [S_1, S_2, \ldots, S_n]$ allow the LLM to identify exactly where and why the query fails, rather than receiving only coarse-grained boolean feedback. We detail each validation dimension below.

Syntax Validation. Syntax validation checks if the query complies with the extracted DSL grammar. The static analysis tool itself provides a runtime for the queries and is naturally equipped with grammar validation capability. However, it is often coarse-grained without pinpointing the exact error locations. Therefore, we instrument and hook the grammar validator to obtain the corresponding grammar rule of the errors. Alternatively, one can leverage off-the-shelf grammar parsers (*e.g.*, ANTLR [52]).

Execution Validation. MoCQ refers to the collected DSL specifications to correct execution exceptions. There are multiple types of semantic errors, such as undefined variables, missing attributes, type mismatches, or incorrect API usage. Our symbolic validator checks these behaviors and outputs the error locations. Besides merely reporting the errors, MoCQ searches the collected DSL

semantics to suggest correct ones by fuzzy matching name similarities. For example, it could suggest `"<operator>.assignment"` as a similar argument for `"assignment"` to fix the `exactName` exception mentioned in §2.3.

Semantic Validation. Inspired by dynamic software testing [9], MoCQ employs trace-driven validation to provide fine-grained semantic feedback to the LLM. MoCQ evaluates the semantic behaviors of query execution by debugging its runtime behavior against the vulnerability example e_i^t. A straightforward approach is to check whether the query as a whole retrieves the vulnerability example (*i.e.*, boolean feedback). However, this is insufficient for helping an LLM pinpoint the root cause when the query fails to retrieve the example. MoCQ tracks intermediate program states along the query execution trace to enable fine-grained debugging. For example, in the prototype pollution query in Listing 1, after executing the block that identifies `obj["__proto__"]` assignments (variable `objProto`), MoCQ captures concrete code locations as a tuple of (`objProto`, `[Node@line4]`). If `objProto` is empty when it should detect the vulnerability example, this signals that the pattern is incorrectly formulated. MoCQ provides the LLM with the annotated trace values $PState$ filtered to the vulnerability example, enabling precise cross-checking and self-correction.

3.3 Query Optimization

We develop optimizations to mitigate query underfitting and overfitting and to improve query execution efficiency.

3.3.1 Eliminating FPs for Precision. When using the query Q_i^t to analyze the project containing the input vulnerability example e_i^t, it might return false positives. A false positive is often introduced at a certain code block B_j. MoCQ thus similarly leverages the program state $PState$ to help the LLM reason about how the false positive is introduced and propose fixes.

In our evaluation, we use a best-effort approach to manually collect all true-positive vulnerabilities in a project as the example set E^t; therefore, all cases outside are considered false positives. In practice, non-vulnerable code is much more prevalent than vulnerable code, so a report from an under-development query is more likely to be a false positive. Recent research has shown it is possible to leverage an LLM to assess if a case is a false positive or not [34]. We leave this as future work.

3.3.2 Generalization for Recall. To mitigate the overfitting issue, MoCQ generalizes the queries to not just focus on a specific example. This is especially important to make the queries capable of real-world detection later on. As an initial step, we design a few heuristics to assess if a query is overfitting. First, MoCQ checks whether the query relies on exact constant values (*e.g.*, hardcoded variable names and string literals) that are unique to a specific example. Such reliance limits the applicability of the query to other contexts. Second, MoCQ examines whether the query includes structural patterns or constraints that are overly specific to the layout of a particular program, such as deep AST chains or unique call sequences. Once any such situation is found, MoCQ employs the LLM to abstract the query by asking it to express the query in a more general form to resolve the identified overfitting situation. Such generalization improves the reusability of queries and enables the detection of vulnerability variants.

3.3.3 Merging for Efficiency. After generating individual queries for different vulnerability examples, MoCQ performs query merging to consolidate detection queries and improve efficiency. A naive approach would simply apply a logical `OR` operation to combine all per-example queries, sequentially running each one independently. However, this can lead to performance degradation, as many queries share redundant and repetitive conditions. To address that, MoCQ introduces an LLM-assisted query merging stage. All individual queries Q_i^t are given to an LLM, which attempts to merge them into a single optimized query (Q^t) while preserving their detection capability. The merged query is passed through the symbolic query validator for all vulnerability examples to verify its correctness and effectiveness (this process is omitted from Algorithm 1). If the merged query fails validation on any of the examples, MoCQ further undergoes a similar feedback loop to iteratively improve the merging process.

4 Implementation

We implemented MoCQ in a highly modular way and integrated it with Joern [72] and CodeQL [13]. The two leading query-based static analysis tools support a wide range of programming languages, including Java, JavaScript, and PHP. The hardware requirements for running MoCQ align with the standard recommendations for Joern [26] and CodeQL [14], typically 16 GB RAM and 4 to 8 CPU cores. We further show some important implementation details.

Instrumentation of Symbolic Validator. We implemented the runtime program state tracking by instrumenting the query execution runtime. In particular, we enhanced the query parsing process to locate syntax errors. For the other two types, we hooked the internal language interpreter. Specifically, these query languages' interpreters typically contain a main interpretation loop that switches over the instruction types and invokes specific handlers. We thus added additional hooks for selected handlers to collect variables and their runtime values.

Query Execution Server. The iterative query generation would invoke the symbolic validator and static analysis runtime multiple times, and often operate on the same codebase, *i.e.*, the project containing the input vulnerability example. To improve the overall efficiency, we eliminate the repetitive project loading process by starting a local server to host the codebase for interactive, continuous query execution.

5 Evaluation

We evaluate MoCQ to answer the following research questions:

- **RQ1: Query Generation.** How well can MoCQ generate valid detection queries, and how does it compare to the baseline?
- **RQ2: Detection Effectiveness.** How effective are MoCQ-generated queries in identifying vulnerabilities, and how do they compare to state-of-the-art static analysis tools?
- **RQ3: Comparison with Expert Queries.** How do the queries generated by MoCQ compare to manually crafted, expert-developed queries in terms of quality and logic?
- **RQ4: Component-wise Analysis.** How does each component of MoCQ contribute to the overall performance?

- **RQ5: New Vulnerability Discovery.** How effective is MoCQ in discovering new vulnerabilities within real-world applications?

5.1 Experimental Setup

Evaluated Vulnerability Types. We evaluate MoCQ on representative vulnerability types from the OWASP Top 10 [51]. As shown in Table 1, we included types across three of the most widely used web programming languages [65]: Java, PHP, and JavaScript. These languages form the dominant stack of modern web applications and exhibit diverse language-specific security behaviors, covering both statically typed (Java) and dynamically typed (PHP, JavaScript) paradigms. The selected vulnerability types span a broad range of attack vectors (*e.g.*, SQL injection, XSS, prototype pollution). Extending MoCQ to additional languages or vulnerability classes requires almost no system modifications (§6).

Dataset. For each vulnerability type, we curated a dataset of vulnerabilities for both query generation and evaluation. We first grouped candidate vulnerabilities by application and deduplicated them to remove overlap among multiple sources. After that, our dataset comprises a total of 343 vulnerabilities sourced from established benchmarks (*e.g.*, CWE-Bench-Java [34], Silent-Spring [59]) and public repositories, including the CVE database and GitHub security advisories. To ensure a rigorous evaluation, we partition the dataset into separate generation and testing sets to prevent data leakage. Specifically, we perform the split at the project level rather than the vulnerability level to ensure that vulnerabilities from the same codebase do not appear in both sets. We randomly split the projects following an approximate 40/60 ratio, resulting in 138 vulnerabilities in the generation set (86 for Java, 30 for PHP, and 22 for JavaScript) and 205 vulnerabilities in the testing set (132 for Java, 46 for PHP, and 27 for JavaScript). This partitioning ensures complete isolation and a robust assessment of the system's effectiveness.

Tool Configuration. MoCQ comes with multiple configurable components. Due to the high costs associated with large-scale LLM inference, we use Claude Sonnet 4.6 and Joern as our default configuration. To ensure output stability and reproducibility across our experiments, the model is empirically configured with a temperature of 0.2. We ran the experiments on a CPU server equipped with dual 48-thread Intel Xeon processors and 187 GB RAM.

5.2 RQ1: Query Generation

In this section, we evaluate the capability of MoCQ to generate valid detection queries using the vulnerabilities in the generation set (§5.2.1). We compare MoCQ against a baseline (§5.2.2) and analyze the computational efficiency (§5.2.3).

For each vulnerability example, we set a maximum iteration threshold of 20 (Max_N). MoCQ terminates the process if a valid query is not produced after this limit. A generation attempt is considered successful if the resulting query successfully detects the input vulnerability example.

5.2.1 Results of MoCQ. Table 1 presents the query generation statistics across the evaluated vulnerability types. Out of 138 cases, MoCQ successfully generated queries for 113 (81.9%). The generated queries have an average length of 92.7 lines and exhibit sophisticated static analysis logic. Specifically, 45 queries implement data-flow or def-use tracking, 78 perform interprocedural analysis across function boundaries, and 34 utilize control-flow constraints to filter spurious matches. Furthermore, 83 queries incorporate specific sanitization logic to mitigate false positives. These results indicate that MoCQ can reason about complex program semantics beyond simple syntactic pattern matching. Failed cases (reaching the 20-iteration limit) were primarily attributed to two factors. First, certain vulnerabilities involve highly intricate data-flow dependencies that proved difficult to refine within the iteration budget. Second, limitations in the underlying static analysis tools, such as incomplete pointer analysis or missing type information, occasionally prevented the generation of a verifiable pattern.

Table 1: Query generation statistics with 138 cases. `MoCQ` and `Baseline` denote valid queries generated using MoCQ and direct prompting. `Time` reports MoCQ's generation time.

Vul. Type	Total	MoCQ	Baseline	Avg. Iter.	Time
Java cmd.	36	29 (80.6%)	5 (13.9%)	14.2	10.8h
Java SQLi	21	19 (90.5%)	7 (33.3%)	8.9	5.8h
Java path.	29	25 (86.2%)	12 (41.4%)	10.6	8.7h
PHP SQLi	12	9 (75.0%)	4 (33.3%)	10.1	3.4h
PHP XSS	8	6 (75.0%)	6 (75.0%)	11.5	2.4h
PHP type.	4	3 (75.0%)	2 (50.0%)	13.2	0.9h
PHP deser.	6	5 (83.3%)	3 (50.0%)	14.9	2.2h
JS proto.	9	7 (77.8%)	4 (44.4%)	15.3	2.6h
JS cmd.	4	3 (75.0%)	3 (75.0%)	11.1	1.1h
JS XSS	9	7 (77.8%)	5 (55.6%)	12.6	2.7h
Overall	138	113 (81.9%)	51 (37.0%)	12.1	40.6h

5.2.2 Comparison with Direct Prompting Baseline. To the best of our knowledge, no existing work provides a direct equivalent for automated query generation of web application vulnerabilities. Therefore, we established a baseline by prompting the same Claude Sonnet 4.6 model to generate queries directly from the vulnerability examples, allowing up to 20 attempts. This baseline uses the same prompt as MoCQ but operates without our DSL harness and the iterative refinement loop. As shown in Table 1, among the 138 cases, the baseline addressed only 51 cases (37.0%). This demonstrates that while frontier models possess some inherent knowledge of vulnerability patterns, they often struggle to produce syntactically correct and semantically precise static analysis queries.

Other Model Backbones. Beyond Claude Sonnet 4.6, we evaluate MoCQ with stronger frontier backbones (Claude Opus 4.6 and Opus 4.8) and a smaller open-source backbone (Qwen3.6-35B [54]), under both MoCQ and the direct prompting baseline. Direct prompting's generation rate does improve with model capability (37.0%→52.2% from Sonnet 4.6 to Opus 4.8), confirming that raw model strength helps. However, MoCQ retains a large, persistent advantage over direct prompting across this entire capability spectrum, from Qwen3.6-35B (38.4% vs. 25.4%) to Claude Opus 4.8 (89.1% vs. 52.2%), confirming that its advantage stems from the DSL harness and iterative refinement loop rather than raw model capability alone. Full per-model, per-language results are provided in the full version of this paper [32].

5.2.3 Efficiency and Cost. We evaluate the computational efficiency and costs of MoCQ. It is worth noting that the query generation process represents a one-time offline effort. Once generated,

Table 2: Detection results on the testing dataset with 205 vulnerabilities. Metrics are computed as: TPR (True Positive Rate or Recall) = $\frac{TP}{TP+FN}$; FDR (False Discovery Rate) = $\frac{FP}{TP+FP}$; and F1 = $\frac{2 \cdot TP}{2 \cdot TP+FP+FN}$.

Approach	TP	FN	FP	TPR	FDR	F1
MoCQ	172	33	75	83.9%	30.4%	0.76
Claude Code	78	127	19	38.0%	19.6%	0.52

these queries can scan unlimited codebases instantly without additional generation overhead or recurring LLM inference costs.

Generation Efficiency. Across the successfully generated queries, MoCQ required an average of 12.1 iterations to reach convergence. The complexity of the vulnerability pattern directly influenced the refinement speed: simpler patterns, such as Java SQL injection, typically converged faster (avg. 8.9 iterations), whereas complex logic like JavaScript prototype pollution required significantly more refinement (avg. 15.3 iterations). The total wall-clock time for generating the entire suite of queries was 40.6 hours, compared to 17.3 hours for the direct-prompting baseline across the same 138 cases. The baseline is faster in wall-clock terms because MoCQ is explicitly instructed to perform iterative refinement, which the baseline does not.

API Cost. Using Claude Sonnet 4.6 as the default LLM, the average inference cost to generate a valid query per vulnerability type was approximately $39 USD on average. Given the 81.9% success rate and the reusable nature of the resulting patterns, MoCQ represents a cost-effective alternative to manual rule engineering by security experts (§5.4). Furthermore, unlike direct LLM-based vulnerability scanning (which incurs costs proportional to the size of the codebase and frequency of scans), MoCQ-generated queries do not incur per-scan API costs, offering superior scalability for large-scale enterprise deployments.

5.3 RQ2: Detection Effectiveness

We apply the queries generated in §5.2 to detect vulnerabilities within the testing set. The overall detection results are presented in Table 2. Overall, MoCQ-generated queries demonstrated strong detection capabilities across a diverse range of vulnerability types. Out of 205 known vulnerabilities, MoCQ successfully detected 172 cases while incurring 75 false positives. This performance corresponds to a True Positive Rate (TPR) of 83.9% and an F1 score of 0.76.

False Negatives. We analyzed the 33 vulnerabilities (16.1%) missed by MoCQ-generated queries. These false negatives stem from multiple root causes across different categories. 10 cases were caused by incomplete patterns, including missing taint source specifications (particularly involving customized third-party sources) or missing vulnerable operations, which could have been detected with more comprehensive patterns. Another 19 cases involved incomplete data flows where the targets of dynamic function calls were missing from the project's program representations. This occurred because the program representations failed to properly handle dynamic language features and indirect calls. The remaining 4 cases were caused by inaccurate type information. For instance, imprecise type information led to false negatives in PHP type juggling and JavaScript prototype pollution. These latter categories are limitations of the underlying static analysis infrastructure and cannot be resolved by refining the patterns alone.

False Positives. MoCQ-generated queries produced a reasonable amount of false positives across the testing set. As shown in Table 2, the overall False Discovery Rate (FDR) for MoCQ is 30.4%. This rate is reasonable for static analysis, which often prioritizes soundness over precision to ensure vulnerabilities are not missed. Most false positives stem from undecidable path feasibility. For example, a query may flag a syntactically valid data flow that is unreachable at runtime due to unsatisfiable path constraints or environmental checks that the static analysis cannot evaluate. Additionally, the dynamic nature of PHP and JavaScript (*e.g.*, complex object mutations and dynamic type systems) introduces over-approximation in the underlying program representation.

Analysis Time. MoCQ's queries took around 38 CPU hours in total to complete the analysis on all tested projects. The analysis is fully offline without LLM invocations. Analysis time depends on application complexity and query design, but the overall duration remains practical for real-world use.

5.3.1 Comparison to SOTA Static Detectors. We further assess MoCQ's effectiveness by comparing it against a diverse range of state-of-the-art (SOTA) detectors. Unlike MoCQ, which provides broad support for multiple languages and vulnerability classes, most existing solutions are specialized for a single language or a specific vulnerability type. Our comparison includes both LLM-based approaches and classic static analysis tools:

- **Claude Code [4]**: A state-of-the-art autonomous agent that directly analyzes codebases. It supports diverse languages and vulnerability types.
- **IRIS [34]**: An LLM-based assistant that identifies taint sources and sinks to augment existing CodeQL queries for Java. Unlike MoCQ, which generates complete, executable queries from scratch, IRIS is limited to extending the specifications of pre-existing expert-written rules.
- **LChecker [31]**: A specialized static analysis tool for PHP. It combines pattern matching with type inference specifically to target type juggling vulnerabilities.
- **Silent-Spring [59]**: A dedicated static detector for JavaScript that focuses on identifying prototype pollution through complex data-flow analysis.

Claude Code. As an LLM-based agentic baseline, we evaluate Claude Code [4], which automatically navigates codebases, executes shell commands, and reads multiple files to build context. We use the same Claude Sonnet 4.6 model as in MoCQ to ensure a fair comparison. For each target vulnerability type, we provide Claude Code with a task description and two examples of real vulnerabilities as demonstrations, and explicitly instruct the agent to perform step-by-step reasoning before making a final detection decision. The agent is allowed to freely inspect the target codebase, traverse files, and execute analysis commands during detection. The complete prompt template used for this agentic baseline is provided in our public artifact.

The evaluation results in Table 2 show that Claude Code significantly underperforms compared to MoCQ. Out of 205 vulnerabilities in the testing set, the agentic baseline identified only 78

Table 3: Detection results against language/tool-specific SOTA baselines: IRIS on 132 Java applications [34], LChecker on 6 PHP type-juggling vulnerabilities [31], and Silent-Spring on 11 JS prototype pollution vulnerabilities [59].

Lang.	Approach	TP	FN	FP	TPR	FDR	F1
Java	CodeQL + CodeQL sources/sinks	37	95	254	28.0%	87.3%	0.17
	MoCQ + CodeQL sources/sinks	55	77	256	41.7%	82.3%	0.25
	CodeQL + IRIS sources/sinks	68	64	324	51.5%	82.7%	0.26
	MoCQ + IRIS sources/sinks	85	47	383	64.4%	81.8%	0.29
PHP	MoCQ	5	1	22	83.3%	81.5%	0.30
	LChecker	6	0	37	100.0%	86.0%	0.24
JS	MoCQ	10	1	8	90.9%	44.4%	0.69
	Silent-Spring	10	1	12	90.9%	54.5%	0.61

true positives, with a TPR of 38.0% and an FDR of 19.6%. While Claude Code is capable of cross-file navigation, its performance in vulnerability detection is hindered by context window saturation: as the agent reads more files to trace data flows, the increasing token count often leads to degraded reasoning precision [4]. For instance, in complex cases such as JavaScript prototype pollution, the agent frequently failed to maintain the full taint propagation path when the flow spanned multiple nested dependencies, leading to missed detections or false positives based on incomplete local context. MoCQ, on the other hand, uses LLMs to generate structured static analysis queries. It offloads the heavy lifting of path tracing to a deterministic engine, avoiding the reasoning bottlenecks inherent in long-context LLM agents.

IRIS. We evaluated IRIS [34] on the 132 Java vulnerabilities from our testing set, more than half of which are sourced from the CWE-Bench-Java dataset used in IRIS. IRIS augments existing CodeQL queries by synthesizing project-specific taint sources and sinks using LLMs, whereas MoCQ generates complete vulnerability detection queries from scratch.

To comprehensively assess their contributions, we evaluated different combinations of queries and taint specifications using CodeQL as the underlying analysis framework. All configurations use CodeQL instead of Joern for a fair comparison, since IRIS is built on top of CodeQL. Specifically, we compared four configurations: (1) built-in CodeQL queries with CodeQL-defined sources/sinks (baseline), (2) MoCQ-generated queries with CodeQL-defined sources/sinks, (3) built-in CodeQL queries with IRIS-synthesized sources/sinks (IRIS's original approach), and (4) MoCQ-generated queries with IRIS-synthesized sources/sinks (combined approach). This setup allows us to separately measure the effectiveness of MoCQ's query generation and the benefit of IRIS's taint specification synthesis.

The results are shown in Table 3. MoCQ-generated queries consistently outperform built-in CodeQL queries across both taint specification settings. Compared with the baseline configuration (1), IRIS alone (3) improves TPR from 28.0% to 51.5%, demonstrating the benefit of synthesizing project-specific taint sources and sinks. Using MoCQ-generated queries alone (2) also substantially improves detection performance, increasing TPR to 41.7% with comparable false positives. The combined approach (4) achieves the best overall results, reaching 64.4% TPR with the highest number of true positives (85), showing that MoCQ's query generation and IRIS's taint synthesis provide complementary benefits. These results demonstrate that MoCQ generates more effective vulnerability detection queries and can be further enhanced when combined with specialized taint specification techniques.

Table 4: Comparison of detection performance between MoCQ, expert-developed queries, and their combined results on the full testing set (205 vulnerabilities).

Approach	TP	FN	FP	TPR	FDR	F1
MoCQ	172	33	75	83.9%	30.4%	0.76
Expert Queries	130	75	155	63.4%	54.4%	0.53
Combined (Union)	179	26	176	87.3%	49.6%	0.64

LChecker. As shown in Table 3, LChecker achieves perfect TPR (100.0%) but suffers from a very high FDR (86.0%), resulting in many false positives. In contrast, MoCQ achieves a better F1 score (0.30 vs. 0.24) with a lower FDR (81.5%), while still maintaining strong TPR. LChecker's over-approximation mainly stems from conservative type inference in dynamic PHP environments, whereas MoCQ avoids this by generating more targeted queries that focus on specific and validated vulnerability patterns through iterative refinement.

Silent-Spring. For JavaScript prototype pollution, MoCQ matches Silent-Spring in TPR (90.9%) while improving precision by reducing the false discovery rate from 54.5% to 44.4% (Table 3). This indicates that MoCQ can reach the performance of a hand-crafted, specialized state-of-the-art detector while remaining a general-purpose framework applicable across different languages.

5.4 RQ3: Comparison with Expert Queries

Existing expert-developed queries for the same underlying tool represent a critical baseline. Since both MoCQ and expert queries operate on the same static analysis framework, this comparison isolates the effectiveness of our query generation loop from the impact of tool-specific capabilities. We evaluated the collected expert-developed queries on the same testing set. These expert queries are taken directly from public query repositories, including the official CodeQL [23] and Joern [48] query repositories, as well as previous papers and theses [22, 59]. We acknowledge a potential asymmetry: MoCQ is given the specific vulnerability examples in our generation set, whereas the expert queries we compare against were written generically, without access to our dataset. A more tightly matched comparison would have human experts write queries from the same examples, but recruiting external experts for weeks of engineering effort was not logistically feasible; we instead compare directly against these community-standard queries as a meaningful, bias-free reflection of currently deployed security practices.

Detection Performance. Overall, MoCQ-generated queries achieved substantially better performance than expert-developed queries, despite the latter requiring significant security expertise and engineering effort. As shown in Table 4, expert-developed queries detected 130 true positives with 155 false positives, compared to MoCQ's 172 true positives with only 75 false positives. Our manual analysis reveals that MoCQ and expert-developed queries have a substantial overlap, identifying a common core of

Table 5: Overview of 45 new vulnerability patterns uncovered by MoCQ. The row marked with * summarizes 34 distinct `phar`-based PHP deserialization operations, of which 3 have been merged, 5 acknowledged, and 26 remain pending review.

Vul. Type	Pattern	Functionality	Status
General PHP	`htmlspecialchars`	Ctx-sensitive sanitization	Merged
PHP type.	`in_array`	Implicit type casting	Merged
PHP type.	`array_search`	Loose comparison	Merged
PHP type.	`array_flip`	Implicit type casting	Reported
PHP type.	`case` in `switch`	Type coercion in case matching	Reported
*PHP deser.	`phar`	Object deserialization	Partial
PHP deser.	`copy`	Object deserialization	Reported
General JS	`Object.assign`	Data propagation	Merged
General JS	`Reflect.set`	Data manipulation	Merged
JS proto.	`Object.assign`	Data propagation	Merged
JS proto.	`Object.defineProperty`	New property manipulation	Merged
JS proto.	`obj.__defineGetter__`	New property manipulation	Merged

vulnerabilities. However, in several categories (*e.g.*, PHP deserialization, JavaScript command injection, and JavaScript XSS), MoCQ-generated queries detected a broader set of vulnerabilities than those covered by existing expert patterns.

Combination for Complementary Strengths. Our analysis reveals distinct cognitive strengths between LLMs and human experts. LLMs excel at capturing subtle low-level linguistic nuances that humans often overlook, while human experts are better at high-level conceptual reasoning. For example, expert-developed queries for PHP frequently overlooked the language's case-insensitivity (*e.g.*, `echo` vs. `eCHo`), a syntactic nuance MoCQ captured automatically. Conversely, expert queries showed superior design in identifying complex multi-step vulnerabilities that require deep structural understanding of application architecture.

This observation motivated us to combine both approaches: the union of MoCQ-generated and expert-developed queries detected 179 true positives with 176 false positives, achieving the highest coverage of 87.3% of all vulnerabilities and showing that combining MoCQ with human expertise yields a more robust detection strategy than either alone. As a preliminary test, we applied MoCQ to refine an existing expert-developed query for JavaScript prototype pollution. MoCQ ran for approximately one hour and automatically revised the query, identifying 3 previously missed vulnerabilities. This demonstrates that MoCQ can serve both as a primary generator and as a secondary refinement tool for legacy expert rules.

New Vulnerability Patterns. Beyond comparable performance, MoCQ discovered 45 new vulnerability patterns missed by expert-developed queries, including vulnerability-specific patterns and three general patterns applicable broadly across programming languages (Table 5). We define a new pattern as an atomic operation (such as a specific API call, expression type, or data flow mechanism) not recognized by any expert-written query. For example, MoCQ identified four new PHP type juggling mechanisms via built-in functions like `array_search` and `array_flip`, and in JavaScript it uncovered a general data propagation method via `Reflect.set`. Each pattern represents a historical blind spot; missing even one can cause an entire class of vulnerabilities to go undetected. Notably, MoCQ identified a new category of PHP deserialization via `phar://`, generating 34 distinct patterns for that class alone; four additional new types (*e.g.*, case-sensitivity handling) differ structurally from any existing rule, while the remaining seven close smaller gaps in current coverage. We submitted these patterns to the maintainers of the respective static analysis tools, and 11 have been merged into upstream repositories to date. Of the 34 `phar`-based patterns, 3 have been merged and 5 acknowledged by maintainers, with the remaining 26 still pending review.

Development Effort. We analyzed the Git commit history of open-source expert-developed queries (full breakdown in the full version of this paper [32]). Expert development typically spans from several days (*e.g.*, 5 days for Java command injection) to multiple weeks (*e.g.*, 7 weeks for JS prototype pollution). While commit history is an imperfect proxy for total engineering effort, it illustrates the significant human timeline required for manual rule engineering. In contrast, MoCQ generates these patterns in a matter of hours, significantly shortening the development cycle for high-quality detection rules.

5.5 RQ4: Component-wise Analysis

We conduct a component-wise ablation study to understand the contribution of each design choice in MoCQ to vulnerability detection performance. MoCQ consists of four main components: (1) DSL specification design with subsetting, which reduces the complexity of the query space, (2) a multi-stage feedback-driven refinement loop, which provides syntax, runtime, and semantic guidance during query generation, (3) query-level optimization techniques that mitigate overfitting and improve generalization, and (4) the underlying LLM backbone used for query generation. To isolate their effects, we systematically disable or replace each component while keeping all other parts of the system unchanged.

- **DSL design variants**. DSL_{100} uses the full DSL specification without subsetting; DSL_0 removes the DSL specification entirely, relying solely on the LLM's prior knowledge and query examples in the prompt. This is different from the baseline described in §5.2.2 as this variant incorporates the feedback loop.
- **Feedback mechanism variant**. The three feedback mechanisms (syntax, runtime, and semantics) are sequentially dependent. Disabling syntax or runtime feedback prevents the LLM from reaching semantic validation, making those variants largely uninformative. We thus evaluate a single variant, W/O-FB, which disables semantic validation feedback while retaining syntax and runtime error correction.
- **Optimization variants**. W/O-Gen disables generalization heuristics that prevent overfitting to specific examples; W/O-FP disables false-positive elimination feedback; W/O-MG disables the query merging.
- **LLM backbone variants**. MoCQ by default leverages Claude Sonnet 4.6. We further evaluate alternative backbone models, including Claude 3.7 Sonnet [5] and GPT-4o [50].

We primarily report vulnerability detection performance and show the F1 scores of different MoCQ variants in Figure 3, since F1 provides a balanced measure of both FPs and FNs. Full detection results (TPR and FDR) and query generation success rates for the ablation variants are presented in the full version of this paper [32].

DSL Design. Using the full DSL without subsetting (DSL_{100}) reduces F1 to 0.66. Although the full specification retains all necessary features, exposing the LLM to the entire DSL at once forces it to search a much larger space of valid constructs. Removing the DSL

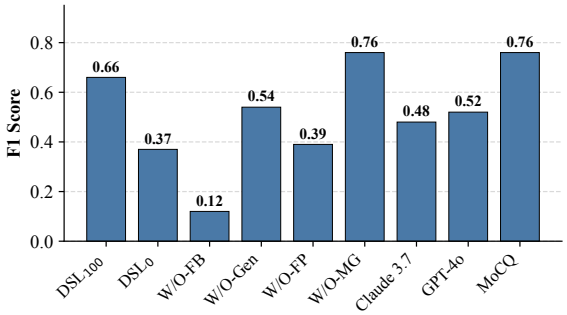


Figure 3: F1 scores of MoCQ and its ablation variants in the component-wise analysis. Full detection results (TPR, FDR, TP, FP) are in the full version of this paper [32].

entirely (DSL_0) causes a more severe drop to an F1 of 0.37. This demonstrates that relying solely on in-context examples and prior knowledge is more prone to errors.

Feedback. W/O-FB suffers from a very low F1 of 0.12. Without semantic validation feedback, the LLM cannot determine whether its queries correctly capture the target vulnerability pattern, resulting in very few effective queries being generated.

Optimization. Without generalization (W/O-Gen), F1 drops to 0.54, as queries overfit to the specific code patterns of input examples and fail to detect structural variations of the same vulnerability class. Disabling false positive elimination (W/O-FP) reduces F1 to 0.39, driven by a rise in FDR from 30.4% to 71.4%. Disabling query merging (W/O-MG) leaves detection results unchanged at an F1 of 0.76, confirming that merging is a pure efficiency optimization with no impact on accuracy.

LLM Backbone. Claude Sonnet 4.6 outperforms both alternative backbones. Claude 3.7 Sonnet achieves an F1 of 0.48, with a higher TPR (62.0%) but substantially higher FDR (60.8%), suggesting it generates broader but less targeted queries. GPT-4o achieves an F1 of 0.52, with lower TPR (46.3%) and lower FDR (41.4%) than Claude 3.7 Sonnet, but detects significantly fewer vulnerabilities overall. These results indicate that MoCQ's performance scales with the capability of the underlying model, and that Claude Sonnet 4.6 provides the best balance of TPR and FDR for query generation. We attribute this to stronger instruction-following ability and more precise use of the DSL: a more capable model better interprets the feedback loop signals and translates them into targeted query refinements, rather than making overly broad or overly conservative adjustments.

These results confirm that the full-fledged MoCQ outperforms all ablation variants; each component contributes to its overall effectiveness, with semantic feedback and false positive elimination being especially critical.

5.5.1 DSL Characteristics. We evaluate the characteristics of the DSL subset extracted by MoCQ. Joern exposes 4,591 DSL APIs and CodeQL exposes 12,934; the subsetting process reduces these to 2,387 (48% reduction) and 4,268 (67% reduction), respectively, substantially shrinking the search space while retaining sufficient expressiveness. Manual analysis of 50 randomly sampled removed APIs, independently rated by two authors with disagreements resolved by discussion, confirmed all were safely excluded, typically because they had equivalent alternatives within the retained subset (*e.g.*, Joern's `whereNot()` can be expressed as `where(not)`), with the rest being non-essential debugging operations or language-specific features irrelevant to our target languages. Conversely, 88% (44/50) of randomly sampled retained APIs were confirmed necessary for expressing the vulnerability patterns in our dataset; the remaining APIs were conservatively kept despite not being strictly required, since retaining extra features is preferable to missing essential ones. Full methodology and per-sample analysis are in the full version of this paper [32].

5.6 RQ5: New Vulnerability Discovery

Evaluation in prior sections already demonstrated that MoCQ can reliably generate detection queries. Here we further apply MoCQ-generated queries to open-source web applications or projects to discover new vulnerabilities. We searched GitHub and NPM Registry platform [49] for actively maintained projects and frameworks using web related keywords (*e.g.*, `web`, `content management`) and applied two selection criteria: (1) actively maintained, with commits within the past year as of Jan 2026, and (2) widely deployed, with at least 500 GitHub stars. We selected 10 projects for each language (30 in total). We scanned the latest version of each project using the MoCQ-generated Joern queries for all applicable vulnerability types and manually triaged all reported findings.

MoCQ discovered 25 previously unknown zero-day vulnerabilities spanning six vulnerability types across three languages, with a full breakdown in the full version of this paper [32]. Among them, 14 have been acknowledged by maintainers as valid security issues and 5 have already been fixed. Confirmed vulnerabilities were responsibly disclosed to the respective maintainers.

6 Discussion

Data Memorization. MoCQ relies on a pretrained LLM to generate queries and its evaluation is subject to data memorization. The model may have seen the vulnerability examples or queries during pretraining. Our experiments took three steps to isolate this risk from MoCQ's measured performance. First, memorizing an existing vulnerability does not by itself solve the *query generation problem*, since the model must still synthesize a valid, tool-specific query from scratch even if it has seen the vulnerable code before. Our direct-prompting baseline shows that raw pretraining memory alone is insufficient to resolve query errors reliably, yielding a significantly lower success rate (§5.2.2). This shows that MoCQ's DSL harness and iterative refinement loop are necessary. Second, we enforce a strict project-level split between the generation and testing sets (§5.1), so no vulnerability or codebase for generating a query is later used to test it. The 25 real-world zero-day vulnerabilities we report (§5.6) are entirely absent from any pretraining corpus, which further demonstrates that MoCQ's detection ability generalizes beyond memorized examples. Finally, the 45 new vulnerability patterns MoCQ discovered beyond existing expert-written queries show that MoCQ is not merely reproducing memorized queries but synthesizing genuinely novel detection logic.

Query Interpretability and Maintainability. Although MoCQ generates these queries with an LLM, they remain interpretable and maintainable by human security engineers. Specifically, these

queries are structured, modular, and expressed using the same standard, declarative DSL and API constructs a human expert would use (§5.5.1). Each query is accompanied by a natural-language explanation of its detection logic. As practical evidence of maintainability, when submitting pull requests to upstream query repositories for the new patterns in Table 5, we manually converted MoCQ-generated queries into structured diffs against the existing human-written queries. This typically took only a few minutes per pull request, indicating high maintainability.

Extensibility. Although we evaluate MoCQ primarily on web applications and their dominant programming languages, the framework naturally extends to other languages and vulnerability classes without significant system changes. This is because MoCQ only requires two inputs: the DSL harness of the underlying static analysis tool and a small number of representative vulnerability examples. To demonstrate this extensibility, we additionally extracted the DSL harness for C/C++ and evaluated MoCQ on use-after-free and heap overflow vulnerabilities. Using vulnerability examples from the Juliet dataset [10], MoCQ successfully generated valid detection queries with a success rate of 77%.

Beyond Joern [7, 72] and CodeQL [13], MoCQ's core workflow of DSL extraction, iterative refinement, and symbolic validation can also extend to other query-based frameworks such as Amazon CodeGuru [58] and Datalog-based systems [56]. For non-query-based approaches that use different pattern representations (*e.g.*, abstract interpretation rules), adapting MoCQ would require defining how vulnerability patterns are represented and how generated patterns can be validated within those frameworks.

Limitations. MoCQ has a few limitations that warrant future research. First, MoCQ's effectiveness relies on the underlying tool's capabilities since MoCQ generates expressive patterns executed by the tools. If a tool lacks essential program representations or capability (*e.g.*, fine-grained pointer analysis), MoCQ cannot guarantee the generation of effective and precise vulnerability queries. Second, MoCQ-generated queries produce false positives, similar to expert-developed queries. A possible way to mitigate this is to leverage LLMs to validate the specific code snippets flagged in the results, as demonstrated in early work [29]. We currently do not include this component as we believe it is orthogonal to the main focus of generating vulnerability patterns. Third, MoCQ currently follows a fixed flow that generates and then executes queries. Some sophisticated vulnerabilities may require more dynamic composition or interplay between LLM and static analysis for complex real-world scenarios. For example, one could leverage LLMs to analyze intermediate results and dynamically adjust query strategies based on partial findings. Future work could combine multiple analysis frameworks to leverage their complementary strengths.

7 Related Work

Pure Learning-based Detection. Transformer-based models have been widely applied to vulnerability detection [21, 70, 71]. To improve performance, researchers have explored domain-specific pretraining [17], instruct-tuning [12, 41], and supervised fine-tuning [69, 77]. However, prior studies [64] show that these methods typically require large labeled datasets and often perform poorly on under-represented vulnerability types and large codebases. Compared to these approaches, MoCQ leverages the scalability of static analysis engines and requires only a small number of vulnerability examples.

LLM-assisted Static Analysis. Recent work combines LLMs with symbolic static analysis components. IRIS [34] uses LLMs to infer project-specific taint specifications for existing Java CodeQL queries, while Artemis [25] extends this idea to PHP applications. These approaches are complementary to MoCQ, as they improve existing queries rather than generating complete queries from scratch. KNighter [73] is a concurrent work that synthesizes Clang static analyzers [38] for C/C++ vulnerability detection. However, as acknowledged in their paper, KNighter does not support temporal or interprocedural analysis, limiting it mainly to simple intraprocedural vulnerabilities. QLCoder [68] is another concurrent work that synthesizes complete CodeQL queries from a CVE's patch, but it is limited to CodeQL and Java. Other works use neural components to assist symbolic validation, such as WAP [43], or decompose analysis into smaller property checks as in NESA [67]. LLift [30] applies LLMs to reason about post-conditions in use-before-initialization bugs in the Linux kernel. LLMxCPG [29] constructs program slices and then leverages an LLM to validate. In contrast, MoCQ focuses on automatically generating vulnerability detection queries from scratch and can complement these systems by substantially reducing manual engineering effort.

Broad Static Analysis. Static vulnerability detection includes a broad range of techniques beyond pattern-based approaches. Methods such as symbolic execution [8, 19], model checking [62, 74], and formal verification [24] analyze program behavior against formal specifications or mathematical models to prove security properties or identify bugs. General dataflow analysis frameworks based on abstract interpretation [16] similarly reason about program behavior using lattice-based abstractions to detect vulnerabilities. Unlike pattern-based tools, these approaches do not rely on manually crafted vulnerability signatures. However, despite their precision and theoretical rigor, such methods often face significant scalability challenges when applied to large or complex codebases. Testing-based approaches such as Statfier [75] and SAST-MT [35] take a different angle by identifying FPs and FNs in existing static analysis tools. MoCQ is complementary as it proactively generates detection queries. As a byproduct, we also found bugs in Joern's data-flow tracking while tracing a query-generation failure, a bug that was subsequently fixed after our report.

8 Conclusion

This work presents MoCQ, a novel approach that combines the complementary strengths of LLMs and static analysis to enable scalable, automated vulnerability detection. By leveraging a DSL harness and trace-driven symbolic validation, MoCQ automatically generates vulnerability detection patterns from examples. MoCQ-generated queries achieved comparable detection performance to SOTA approaches while reducing engineering effort from weeks to hours. Beyond comparable performance, MoCQ discovered 45 new vulnerability patterns missed by experts and detected 25 previously unknown vulnerabilities in real-world applications. By lowering the barrier to developing custom vulnerability detectors, we believe

this approach represents a promising direction to make security analysis more accessible and adaptable.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation under grants CNS-21-54404 and CNS-20-46361, and the Columbia Research Stabilization Fund. Junfeng is also supported DARPA ScAN, NSF CNS 2526620, Google, Samsung, Amazon, and Columbia CAIFI (with Capital One), Enterprise AI (with Infosys), and CDFT centers. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the funding agencies.

References

[1] 2024. CodeQL 2.14.2 Change Log. https://codeql.github.com/docs/codeql-overview/codeql-changelog/codeql-cli-2.14.2/#javascript-typescript.

[2] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. 2022. Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[3] Feras Al-Kassar, Luca Compagna, and Davide Balzarotti. 2023. WHIP: Improving Static Vulnerability Detection in Web Application by Forcing tools to Collaborate. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*. 6079–6096.

[4] Anthropic. 2025. Claude Code: An agentic tool for software engineering. https://claude.com/product/claude-code.

[5] Anthropic. 2025. Introducing Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet.

[6] Anthropic. 2025. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6.

[7] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroSP)*. Paris, France.

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

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyên, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[10] Frederick E. Boland, Jr. and Paul E. Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (Oct. 2012), 88–90. doi:10.1109/MC.2012.345

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.

[12] Wei Chang, Chunyang Ye, and Hui Zhou. 2024. Fine-Tuning Pre-trained Model with Optimizable Prompt Learning for Code Vulnerability Detection. In *Proceedings of the 2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. 108–119.

[13] CodeQL. 2024. CodeQL. https://codeql.github.com/.

[14] CodeQL. 2024. CodeQL Hardware Requirements. https://docs.github.com/en/code-security/code-scanning/creating-an-advanced-setup-for-code-scanning/recommended-hardware-resources-for-running-codeql.

[15] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. 2024. GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes. In *Proceedings of the 33th USENIX Security Symposium (Security)*. Philadelphia, PA, USA.

[16] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[17] Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2023. CONCORD: Clone-Aware Contrastive Learning for Source Code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 26–38. doi:10.1145/3597926.3598035

[18] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability detection with code language models: How far are we?. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. Ottawa, Ontario, Canada.

[19] I. A. Dudina and A. A. Belevantsev. 2017. Using static symbolic execution to detect buffer overflows. *Programming and Computer Software* 43, 5 (2017), 277–288.

[20] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP)*. IEEE, 1125–1142.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Online.

[22] Tobias Fröberg. 2023. *Detection of Prototype Pollution Using Joern: Joern's Detection Capability Compared to CodeQL's.* Master's thesis.

[23] GitHub. 2024. CodeQL: The Libraries and Queries that Power Security Researchers Around the World. https://github.com/github/codeql.

[24] Osman Hasan and Sofiene Tahar. 2015. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global Scientific Publishing, 7162–7170.

[25] Yuchen Ji, Ting Dai, Zhichao Zhou, Yutian Tang, and Jingzhu He. 2025. Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-Procedural Path-Sensitive Taint Analysis. In *Proceedings of the 2025 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Singapore.

[26] Joern. 2024. Joern Hardware Requirements. https://docs.joern.io/installation/#configuring-the-jvm-for-handling-large-codebases.

[27] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[28] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual event.

[29] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware vulnerability detection through code property Graph-Guided large language models. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. 489–507.

[30] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1 (April 2024). doi:10.1145/3649828

[31] Penghui Li and Wei Meng. 2021. LChecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference (WWW)*. Ljubljana, Slovenia.

[32] Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. 2026. Automatically Learning Vulnerability Patterns for Scalable Static Analysis of Web Applications. arXiv:2504.16057 Full version. https://arxiv.org/abs/2504.16057.

[33] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node. js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece.

[34] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. Llm-assisted static analysis for detecting security vulnerabilities. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*. Singapore.

[35] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing* (2024), 4600–4618.

[36] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[37] Zhengyu Liu, Kecheng An, and Yinzhi Cao. 2024. Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node.js Template Engines for Malicious Consequences. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA.

[38] LLVM Project. 2024. Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed: 2024.

[39] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* (2024).

[40] Changhua Luo, Penghui Li, and Wei Meng. 2022. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.

[41] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards Explainable Vulnerability Detection With Large Language Models. *IEEE Transactions on Software Engineering* 51 (2024), 2957–2971. https://api.semanticscholar.org/CorpusID:270521866

[42] Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). In *Encyclopedia of computer science*. 129–131.

[43] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2011. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 21st International World Wide Web Conference (WWW)*. Seoul, Korea.
[44] MITRE Corporation. [n. d.]. Common Vulnerabilities and Exposures (CVE) Program. https://www.cve.org/. Accessed: 2026-04-23.
[45] MITRE Corporation. 2021. CVE-2021-44228: Apache Log4j2 Remote Code Execution Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. Accessed: 2025-06-22.
[46] N/A. 2024. CodeQL 2.16.3 Change Log. https://codeql.github.com/docs/codeql-overview/codeql-changelog/codeql-cli-2.16.3/#javascript-typescript.
[47] N/A. 2025. Joern Documentation: Node-Type Steps. https://docs.joern.io/cpgql/reference-card/.
[48] N/A. 2025. Open-source code analysis platform for C/C++/Java/Binary/Javascript/Python/Kotlin based on code property graphs. https://github.com/joernio/joern.
[49] N/A. 2026. NPM Registry platform. https://www.npmjs.com/.
[50] OpenAI. 2024. GPT-4o Technical Report. https://openai.com/index/gpt-4o. Accessed: 2025-04-07.
[51] OWASP Foundation. 2021. OWASP Top 10: The Ten Most Critical Web Application Security Risks. https://owasp.org/Top10/.
[52] Terence Parr. 2013. The definitive ANTLR 4 reference. (2013).
[53] The Chromium Project. 2025. CodeQL Support in Chromium. https://chromium.googlesource.com/chromium/src/+/refs/tags/126.0.6436.1/tools/codeql/README.md.
[54] Qwen Team, Alibaba Group. 2026. Qwen3.6-35B-A3B: Agentic Coding Power, Now Open to All. https://qwen.ai/blog?id=qwen3.6-35b-a3b.
[55] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.
[56] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.
[57] Apache Logging Services. 2021. Log4Shell: RCE Vulnerability in Log4j (CVE-2021-44228). https://logging.apache.org/log4j/2.x/security.html. Accessed: 2025-06-22.
[58] Amazon Web Services. 2025. Amazon CodeGuru. https://aws.amazon.com/codeguru/.
[59] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node. js. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.
[60] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. 2024. RecurScan: Detecting Recurring Vulnerabilities in PHP Web Applications. In *Proceedings of the Web Conference (WWW)*. Singapore.
[61] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. 2024. A systematic literature review on automated software vulnerability detection using machine learning. *Comput. Surveys* 57, 3 (2024), 1–36.
[62] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. 2021. Model-checking support for file system development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. 103–110.
[63] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[64] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA.
[65] W3Techs. 2026. Usage Statistics and Market Share of Server-side Programming Languages for Websites. https://w3techs.com/technologies/overview/programming_language. Accessed: 2026-04-24.
[66] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS)*. New Orleans, LA, USA.
[67] Chengpeng Wang, Yifei Gao, Wuqi Zhang, Xuwei Liu, Jinyao Guo, Mingwei Zheng, Qingkai Shi, and Xiangyu Zhang. 2026. NESA: Relational Neuro-Symbolic Static Program Analysis. *Proc. ACM Softw. Eng.* 3, FSE (July 2026). doi:10.1145/3808161
[68] Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. 2026. QLCoder: A Query Synthesizer For Static Analysis of Security Vulnerabilities. In *Proceedings of the Fourteenth International Conference on Learning Representations*.
[69] Rongcun Wang, Senlei Xu, Yuan Tian, Xingyu Ji, Xiaobing Sun, and Shujuang Jiang. 2024. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. *Computers & Security* (2024). doi:10.1016/j.cose.2024.103994
[70] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:258685677
[71] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic.
[72] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA.
[73] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. Knighter: Transforming static analysis with llm-synthesized checkers. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*. Seoul, South Korea.
[74] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 393–423.
[75] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated testing of static analyzers via semantic-preserving program transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 237–249.
[76] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. Lisbon, Portugal.
[77] Jin Zhu, Hui Ge, Yun Zhou, Xiao Jin, Rui Luo, and Yanchen Sun. 2024. Detecting Source Code Vulnerabilities Using Fine-Tuned Pre-Trained LLMs. In *Proceedings of the 2024 IEEE 17th International Conference on Signal Processing (ICSP)*. 238–242.

A Open Science

The system implementation of MoCQ and the prompts are provided in https://github.com/columbia/mocq.

B Ethics Considerations

We identify two primary ethical considerations in this work. First, the new vulnerabilities we discovered could affect the software, its developers, and its users. We mitigate this by responsibly disclosing every finding to the respective maintainers before submission and withholding it from the public; for cases not yet acknowledged or patched, we maintain confidentiality and omit identifying details (*e.g.*, project names or reproduction steps) while continuing to work with maintainers toward a fix. Second, using LLMs in security-critical contexts carries inherent risks, which we mitigate through rigorous testing, human review, and coordination with upstream maintainers before any query is deployed. Our evaluation uses only publicly available vulnerability benchmarks and open-source projects and involves no personally identifiable information.

C Generative AI Usage

LLMs were used only editorially, to polish our initial draft for clarity and grammar, with all outputs inspected by the authors. We independently conducted the literature review, developed MoCQ, designed and executed all experiments, and wrote the initial draft; all technical content, experiments, results, and conclusions remain the authors' own work. Separately, MoCQ itself uses LLMs (primarily Claude Sonnet 4.6) to generate detection queries, which are rigorously validated against ground-truth datasets before use; we will release all generated queries in our artifact for independent verification.