ARQ: Agentic CodeQL Query Refinement for C/C++ Vulnerability Detection

Chunyi Wang * **, Yunfei Ke** * **, Junfeng Yang, Yun-Yun Tsai, Penghui Li**

Department of Computer Science, Columbia University, New York, United States
{cw3723, yk3108}@columbia.edu, junfeng@cs.columbia.edu, {yt2781, pl2689}@columbia.edu

arXiv:2608.20637v1 [cs.CR] 21 Aug 2026

Abstract

Static analyzers have been widely adopted for vulnerability detection in C/C++ programs. Query-based static analyzers (e.g., CodeQL) encode vulnerable code patterns in detection queries and match them against source code. However, existing queries still suffer from false positives (FPs, incorrectly flagging benign code as vulnerable) and false negatives (FNs, missing real vulnerabilities). We present ARQ, an agentic framework that automatically refines C/C++ CodeQL queries using execution-grounded evidence from synthesized C/C++ programs. Our key insight is that a synthesized program exposes a query's weakness whenever its execution disagrees with the query's verdict. If the program is genuinely vulnerable but the query stays silent, the query has an FN weakness; if the program is safe but the query fires anyway, it has an FP weakness. ARQ then runs an LLM-based refinement loop that repairs the query using these disagreements as ground truth. Unlike previous query refining methods, ARQ requires no labeled datasets, no commit history, and no vulnerability-specific templates. We demonstrate the effectiveness of ARQ by refining 12 official CodeQL queries using three commercial LLMs (GPT-5.4, Claude-Sonnet-4.6, and Gemini-3.5-flash). We compare both ARQ-refined and original CodeQL queries on the Juliet v1.3 and FormAI v2 datasets and show that ARQ-refined queries detect substantially more true positives, by up to 119.8%, with a Precision of at least 98.0% throughout. ARQ successfully fixed three unresolved GitHub issues raised in the official CodeQL query repository that had remained open for as long as *27 months*. The refined queries also exposed two previously undiscovered bugs in the real-world libraries libpng and zlib.

1 Introduction

Modern software systems such as operating systems, browsers, and embedded systems are usually built with low-level programming languages (e.g., C or C++). Although these languages provide programmers with fine-grained control over system resources, improper resource management can make the programs prone to security vulnerabilities. For instance, buffer-overflows, use-after-free, and double-free (Stroustrup and Sutter 2026; Smirnov 2007) are among the most severe vulnerability classes in deployed software, with over 33,000 reported CVEs from 2020–2025 (NIST Information Technology Laboratory 2026).

To combat these vulnerabilities, static analysis tools such as CodeQL (CodeQL 2026), Semgrep (Semgrep 2026), and Joern (Joern 2026) rely on queries that encode vulnerable code patterns with domain-specific language (DSL) (Avgustinov et al. 2016). Such queries are manually written. Then, they are used to scan and flag any part of vulnerable source code in the target software. The most well-known example is the GitHub Advanced Security (GHAS) platform, which uses CodeQL to automatically detect vulnerabilities across millions of repositories. Major open-source projects such as Chromium have adopted CodeQL as a required part of their security review process (Google 2026a; Microsoft 2026b).

However, two main challenges exist in current static analysis tools. (1) Writing good static analysis queries requires human expertise in software systems, software security, and the DSL used (Chibotaru et al. 2019; Li, Dutta, and Naik 2024; Li et al. 2025; Wang et al. 2026). (2) Even well-designed pre-existing queries in CodeQL still suffer from false positives or missing real vulnerability patterns in practice. Reported query bugs remain open for months to years (6–27 months) without a fix, as we show in § 4.2.

Prior approaches (Shi et al. 2024; Backes et al. 2017; Li et al. 2025; Wang et al. 2026) focus on web applications and cannot be extended to handle system software written in C/C++. IRIS (Li, Dutta, and Naik 2024), for example, targets Java web applications by using an LLM to label taint sources and sinks, then detects injection vulnerabilities by tracking whether tainted data reaches a sink. This works well for vulnerability classes defined by data flow, such as SQL-injection, but C/C++ memory-safety bugs instead hinge on the lifetime and aliasing of memory allocations (Stroustrup and Sutter 2026; Smirnov 2007). For example, a pointer might be used after its target has been freed, or an access might fall outside an allocated buffer's bounds, regardless of whether any tainted data is involved. This kind of reasoning falls outside what taint tracking captures, so IRIS's approach cannot simply be retargeted at C/C++. To the best of our knowledge, *no prior work targets C/C++ CodeQL queries.*

In this work, our goal is to automatically expose and fix false positive and false negative failures in C/C++ CodeQL queries.

We propose ARQ, a novel agentic framework that can

* These authors contributed equally.

Copyright © 2027, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

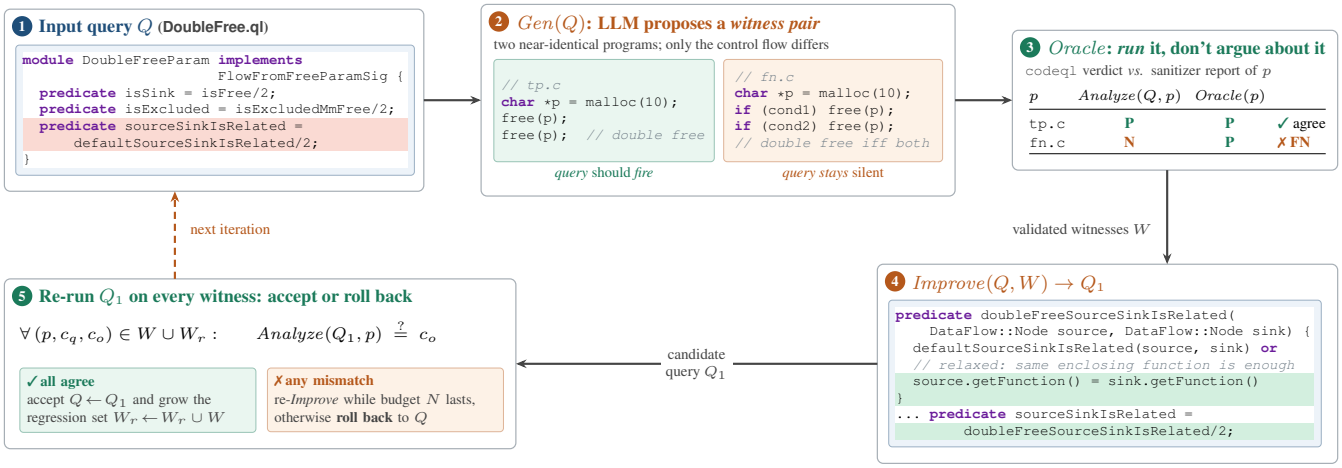

Figure 1: ARQ refining DoubleFree.ql. ① the query under refinement. ② the LLM proposes a pair of near-identical programs, tp.c and fn.c, that should disagree on the query's verdict. ③ an oracle settles whether programs are actually vulnerable through execution. ④ guided by this confirmed witness (fn.c), the LLM proposes a refined query. ⑤ the refined query is re-run against every previously validated witness before being accepted, so a fix cannot silently regress past cases.

automatically synthesize programs and refine queries for C/C++ vulnerability detection. Unlike existing static analysis tools, which require vulnerability patterns, commit history, or labeled datasets, ARQ leverages LLMs to iteratively refine queries based on program synthesis and execution-grounded evidence. In particular, we provide the query as input and let the LLM synthesize C/C++ programs that can expose the query's weaknesses, such as programs that are false positive or false negative. We then execute these programs to validate whether they are genuine witnesses of the exposed weakness. ARQ then takes these validated programs as input and iteratively updates the query, guided by paired vulnerable and non-vulnerable programs to prevent overfitting to individual cases. Figure 1 illustrates the refinement loop end-to-end on DoubleFree.ql, a real query from the official CodeQL repository. The next section explains in detail how ARQ generates confirmed witnesses and uses them to drive concrete refinements without regressing any previously validated case.

We highlight our main contributions:

- ARQ automatically synthesizes witness programs that expose query failures and uses their runtime behavior to confirm the vulnerability is present.
- ARQ uses paired programs to guide the LLM to iteratively refine queries and suppress false positive explosion.
- ARQ substantially improves existing CodeQL queries on the Juliet v1.3 and FormAI v2 datasets when backed by a capable model, achieving up to +119.8% more detected vulnerabilities while keeping Precision at or above 98.0%.
- ARQ demonstrates real-world impact by fixing 3 previously unresolved GitHub issues in the official CodeQL query repository and uncovering 2 previously undiscovered bugs in the widely-used C/C++ libraries zlib and libpng.

2 Preliminaries

2.1 Illustrative Example

We illustrate a buggy query DoubleFree.ql[1] in GitHub's official CodeQL query suite. Figure 1 shows how ARQ improves the query.

The Buggy Query A double-free happens when a program calls free twice on the same pointer. This corrupts the memory allocator's internal bookkeeping and can be exploited to crash the program. Attackers may exploit the crash to mount a denial-of-service attack.

The existing query detects double-free by tracking each memory allocation (the *source*) to its deallocation calls (the *sinks*), then checking whether two such calls could both fire after the same allocation (see step ① in Figure 1).

However, this query further requires the two sink calls to satisfy the rule, defaultSourceSinkIsRelated, which holds only when both calls lie within the same **if** branch. This is where the query breaks. Two free calls in separate, independent **if** statements can still both execute on the same run whenever both conditions hold, but the rule never connects them because they sit in different branches. Our goal is to automatically refine CodeQL queries to reduce such failures at scale, without requiring the deep manual expertise that hand-fixing them demands.

Generating and Validating Witnesses In step ②, ARQ asks the LLM to generate a pair of witnesses, programs whose execution will serve as evidence for or against the query's verdict. The names tp.c and fn.c denote the verdict each program is meant to elicit, a true positive and a false negative respectively, and the two programs differ only in control flow. tp.c is a genuine true positive, since its two free calls are adjacent. fn.c is a false negative. Its two free calls instead

[1] https://github.com/github/codeql/blob/d1fed84daf8f3abc59cf9453692b98632932d60f/cpp/ql/src/Critical/DoubleFree.ql

sit in different **if** branches, so `sourceSinkIsRelated` never holds, even though the same pointer is freed twice whenever both conditions are true. In step ③, ARQ applies `DoubleFree.ql` and the compiler's sanitizer (The LLVM Foundation 2026; Free Software Foundation 2026) (compiled with `-fsanitize=`) to both programs. The sanitizer confirms `tp.c` is vulnerable, while `DoubleFree.ql` does not flag `fn.c` even though the sanitizer does. This mismatch on `fn.c` confirms a real false negative, not a hallucinated one.

Refining the Query Guided by the confirmed witness from step ③, step ④ has the LLM refine `DoubleFree.ql` by relaxing `sourceSinkIsRelated` to also allow two frees within the same enclosing function. In step ⑤, ARQ re-validates the refined query against every previously confirmed witness, including `tp.c` and `fn.c`, before accepting it. This concludes one iteration of refinement. ARQ repeats this loop across multiple iterations, alternating between false-negative and false-positive witnesses, until the refinement budget is exhausted.

2.2 Challenges of Refining Queries

Automating this refinement faces two main challenges. First, most query failures are never filed as bug reports and remain latent, so discovering them requires proactively generating programs that expose disagreements between the query and real vulnerability behavior. A missed report alone does not reveal a failure, since a program not flagged may be a genuine true negative, so confirming a failure needs an independent source of ground truth. Second, improving recall by broadening a query's matching conditions risks a false positive explosion, since loosening the checks that normally rule out a match, or extending what the query looks for, may cause it to flag benign patterns that cannot be ruled out statically. Neither challenge can be resolved by reasoning about the query in isolation. Both require a way to check the query's behavior against what actually happens when a program runs.

2.3 Key Ideas: Execution-driven Reasoning

Unlike purely LLM-based reasoning about a query's correctness, ARQ executes programs and uses their runtime behavior to directly establish whether a vulnerability is present.

This execution-driven approach addresses both prior challenges. ARQ automatically discovers and confirms query failures using ground-truth from program execution. Similarly, ARQ validates every LLM-based refinement through execution, and each updated query is immediately re-run against known programs to check whether the change introduced spurious reports.

3 ARQ Framework

We formalize the problem as follows. Given initial query Q, ARQ should produce a refined query Q_N through multiple iterations of LLM-based refinement, such that Q_f passes all the witness checks. During refinement, ARQ first calls LLM-based generation function $Gen(\cdot)$ to generate programs that try to expose Q's failures, and confirms whether generated programs successfully expose the failures via weakness detection functions $Oracle(\cdot)$ and $Analyze(\cdot)$. ARQ then refines Q with $Improve(\cdot)$ guided by generated programs, then confirms whether the refined query Q_1 is correct, also using $Oracle(\cdot)$ and $Analyze(\cdot)$. In later iterations, previously generated programs are accumulated and used to test for regressions. Algorithm 1 shows the ARQ framework, and we explain and define each component in the sections that follow.

3.1 Generating and Validating Witnesses

A query's weaknesses come in the form of reporting false positives and false negatives. We define a *query failure* as a false positive or false negative that the query commits against a specific program p. Formally, every program p has two classifications, each drawn from $\{P, N\}$, where P (positive) means program p is vulnerable, and N (negative) means it is not. The query's own verdict is $c_q = Analyze(Q, p)$, and the ground-truth verdict is $c_o = Oracle(p)$. A query failure occurs whenever the two disagree, $c_q \neq c_o$. When $c_o = P \wedge c_q = N$, the query commits a *false negative*, missing a real vulnerability; when $c_o = N \wedge c_q = P$, it commits a *false positive*, wrongly flagging safe code. We denote the tuple that documents this outcome as a *witness* $w = (p, c_q, c_o)$. A witness w also documents a query success, in which case $c_q = c_o$. A regression set W_r stores every witnessn validated in earlier rounds, both successes and confirmed failures. Whenever a refined query is proposed, we recheck it against all the witnesses in W_r, so a fix for one weakness cannot silently break a case that was previously validated.

$Gen(Q)$ may generate multiple witnesses. We ask the LLM to generate a pair of witnesses $w_0 = (p_0, c_q^0, c_o^0)$ and $w_1 = (p_1, c_q^1, c_o^1)$, asking it to make p_0 and p_1 as similar as possible. We then verify that $c_o^0 = c_q^0$ (a success, a *true positive* when both equal P or a *true negative* when both equal N) and $c_o^1 \neq c_q^1$ (a failure), so the two together form a matched TN-FP or TP-FN pair. Intuitively, pairing near-identical programs encourages the LLM to reason about the vulnerable control or data flow pattern itself, instead of overfitting by matching exact function names in p_1.

We request $Gen(Q)$ to alternate between TN-FP and TP-FN pairs, since a change that increases Recall may also decrease Precision, and vice versa. No valid W being generated within a reasonably large budget suggests the query is already in good shape, at least with respect to the pair being requested.

We observed that baseline CodeQL queries tend to be extremely conservative. On the first iteration, $Gen(Q)$ was never able to generate (p, P, N), i.e., a false positive, that passed $Oracle$ validation, though the LLM sometimes believed, or hallucinated, $c_o = N \wedge c_q = P$. As a result, the LLM/agent is free to choose and report which pair it generates; we only request a prioritization, not enforce it.

Since $Gen(\cdot)$ is an LLM, each generated tuple $(p, c_q, c_o) \in W$ is only a belief until checked. Algorithm 1 validates it by confirming $c_o = Oracle(p) \wedge c_q = Analyze(Q, p)$ before accepting W. $Oracle(\cdot)$ is a critical component of our algorithm, as it produces ground-truth used to detect hallucinations. We observe that sanitizers provided by major compiler vendors are a convenient, high-quality, real-world instantiation of $Oracle(\cdot)$. For example, Clang provides AddressSanitizer,

MemorySanitizer, UndefinedBehaviorSanitizer, and ThreadSanitizer. In a nutshell, programs instrumented by sanitizers will report vulnerable behaviors at runtime. These can be used to ground a large number of CodeQL queries.

Algorithm 1: Adversarial Refinement Iteration.

Input: Query Q, witness generation budget N, LLM refinement step T, regression tests W_r.
Primitives: $Gen(Q)$ is a generation function that generates witness programs p for query Q. $Analyze(Q, p)$ and $Oracle(p)$ are weakness detection functions. $Improve(Q, W)$ is a query refinement function.
Output: Refined query Q^*

$W_r \leftarrow \emptyset$;
while $T > 0$ **do**
  $T \leftarrow T - 1$;
  `// Phase 1: generate and validate`
  W_r, W_c = FIND_WITNESS(Q, N, W_r)
  `// Phase 2: refine`
  W_r, Q' = REFINE(Q, W_c, W_r)
  $Q \leftarrow Q'$
$Q^* \leftarrow Q'$
return Q^*
Procedure FIND_WITNESS(Q, N, W_r)
  $found \leftarrow$ **false**;
  while $N > 0$ **do**
    $W \leftarrow Gen(Q)$;
    $N \leftarrow N - 1$;
    if $\forall (p, c_q, c_o) \in W : Analyze(Q, p) = c_q \wedge Oracle(p) = c_o$ **then**
      $found \leftarrow$ **true**;
      $W_r \leftarrow W_r \cup W$;
      $W_c \leftarrow W$; `// current witness`
      break;
  if $\neg found$ **then**
    return $(W_r, \emptyset)$; `// no witness found`
  else
    return (W_r, W_c)
Procedure REFINE(Q, W_c, W_r)
  $Q' \leftarrow Improve(Q, W_c)$;
  $\tilde{W} \leftarrow \emptyset$;
  forall $w \in W_r$ **do**
    $p, c_q, c_o \leftarrow w$;
    $c_1 \leftarrow Analyze(Q', p)$;
    if $c_1 \neq c_o$ **then**
      $\tilde{W} \leftarrow \tilde{W} \cup w$; `// store misclassified witness`
  if $\tilde{W} = \emptyset$ **then**
    return (W_r, Q'); `// success`
  $Q' \leftarrow Improve(Q', \tilde{W})$;
  return (W_r, Q'); `// failure`

3.2 Refining the Query

Given validated witnesses, $Improve(Q, W)$ applies targeted edits to address the weaknesses exposed by W. Explicit witnesses are essential because the LLM-based improver may otherwise misidentify the source of failure.

Algorithm 1 implements refinement in two phases. In Phase I, FIND_WITNESS repeatedly samples candidate programs with $Gen(Q)$, up to budget N, until it finds a witness satisfying the validation condition defined by $Analyze$ and $Oracle$. If no valid witness is found, REFINE returns the original query unchanged. In Phase II, the validated witness set W_c is used to produce a candidate query $Q' \leftarrow Improve(Q, W_c)$. The candidate is then checked against the accumulated regression set W_r. Let $\widetilde{W} \subseteq W_r$ denote the witnesses that Q' still misclassifies. If $\widetilde{W} = \emptyset$, the update is accepted; otherwise, Q' is refined again using $\widetilde{W}$ until either all regression witnesses are resolved or the refinement budget is exhausted. On failure, REFINE rolls back to the original query.

The outer loop applies REFINE for up to T iterations. Each iteration searches for new witnesses, adds them to W_r, and attempts a regression-safe update. This iterative process allows ARQ to address multiple weaknesses while preserving previously validated behavior.

Overall, ARQ combines contrastive witness generation, candidate validation, regression testing, and rollback. These mechanisms support incremental improvements in query behavior while preventing updates that introduce new errors.

3.3 Implementation

We implement Gen, $Improve$, $Analyze$, and $Oracle$ by delegating them to LLM-based coding agents, each equipped with task-specific tools.

Likely due to the lack of training data (Wang et al. 2026), even modern LLMs underperform when writing CodeQL queries. Thus, during $Improve$, we provided the LLM/agent with CodeQL syntax Language Server Protocol through Multi-context Protocol (Wang et al. 2026) to help with grammar. Both $Improve$ and $Analyze$ will also report if syntax errors are encountered.

While we described budget N as a fixed number of attempts, nothing stops it from being a time limit. Timeouts are particularly useful if we delegate Algorithm 1 to commercial agents, where a precise limit on the number of attempts is hard to maintain. Without explicit budget control, the task in Algorithm 1 is simple enough that even agents backed by non-reasoning models can complete it well.

4 Evaluation

In this section, we evaluate ARQ to answer the following four research questions:

- **RQ1:** How well do queries refined by ARQ perform?
- **RQ2:** Can ARQ address real-world CodeQL issues and analyze real-world software?
- **RQ3:** Are $Analyze$ and $Oracle$ necessary for effective refinement?
- **RQ4:** How efficient is ARQ in improving queries?

We evaluate ARQ using three mainstream commercial models as its underlying LLMs. Specifically, we use Gemini-3.5-flash in Google Antigravity (Google 2026c), Claude-Sonnet-4.6 in Claude Code (Anthropic 2026), and GPT-5.4 in Codex (OpenAI 2026). We configured each model's thinking level to be low or disabled to reduce cost. Throughout the remaining sections, we may abbreviate Gemini-3.5-flash as simply "Gemini3.5", Claude-Sonnet-4.6 as "Sonnet4.6", and GPT-5.4 as "GPT5.4".

4.1 RQ1: Query Performance

Dataset We evaluate these CodeQL queries on Juliet v1.3 (Black and Black 2018; Jr and Black 2012) and FormAI v2 (Tihanyi et al. 2025). Juliet is a hand-crafted, labeled C/C++ dataset with enough structural complexity to model real-world programs, and it has been widely used to evaluate program analysis tools since its release (Chen et al. 2023; Dubniczky et al. 2025; Black and Black 2018; Tihanyi et al. 2025). FormAI is a large corpus of C programs labeled by bounded model checkers. We do not include other C/C++ datasets, mainly for two reasons. First, several existing datasets fail to provide reliable, project-scale ground truth: PrimeVul (Ding et al. 2024) and DiverseVul (Chen et al. 2023) draw individual functions and files from open-source projects rather than complete, compilable projects, and PrimeVul's labels are LLM-generated rather than manually verified; Castle (Dubniczky et al. 2025) is manually labeled but contains only 250 programs, too small to support the scale of automated refinement and evaluation we require. Second, reliably compiling large numbers of real-world C/C++ projects is difficult given the wide variety of build systems and dependency managers in use. Juliet and FormAI avoid these problems while still exercising realistic code complexity. We complement these results with validation on real open-source projects in § 4.2.

For Juliet v1.3, we only used samples targeting Linux. For FormAI v2, we discarded non-compilable samples and samples without provable classifications.

Setup We evaluate 12 CodeQL queries on the two datasets described above. These queries cover common C/C++ vulnerability types, such as buffer-overflows, use-after-free, and double-free, and are drawn from the official CodeQL repository v2.23.1 (CodeQL 2026), representative of the memory-safety and related CWE categories targeted by CodeQL's C/C++ query suite.

For each dataset, CodeQL is given 12 hours and 32 GiB memory to run each query on Ubuntu 24.04 on an AMD EPYC 7502 CPU.

Baselines We compare only against the original CodeQL queries, not against IRIS (Li, Dutta, and Naik 2024) nor QLCoder (Wang et al. 2026). Both target Java: IRIS infers taint sources and sinks to feed a generic taint-analysis template, and QLCoder synthesizes a new query from CVE metadata; neither has a C/C++ implementation (CodeQL queries are language dependent), and neither refines an existing query the way `ARQ` does. Both also rely fundamentally on taint tracking, which does not capture the memory-lifetime and pointer-aliasing reasoning that our target vulnerability classes require. A fair comparison would require reimplementing both systems for C/C++ from scratch, risking a reimplementation that misrepresents their intended capabilities; we consider this out of scope for this work.

Metric Ideally, as with most vulnerability detection tasks, we should evaluate both precision and recall. However, recall is not feasible to compute here. Juliet and FormAI both label their samples by one CWE type, but CodeQL queries and CWE labels are not in a one-to-one relationship. An incorrect character conversion, for example, may also constitute an invalid pointer dereference (of different type), and each sample carries only one CWE label even though nothing prevents a use-after-free from occurring after a double-free in the same program. This makes it impossible to establish a well-defined count of total negative (or total actual positive) samples for a given query. *This limitation stems from the lack of query-type-labeled benchmarks, not from any limitation of `ARQ` or its refined queries*. We therefore report Precision. Precision $= \frac{TP}{TP+FP}$ Here, a *true positive* (TP) is a query detection on a sample that contains a genuine instance of the targeted vulnerability, and a *false positive* (FP) is a query detection on a sample that does not.

System	Juliet v1.3			FormAI v2		
	TP	FP	Prec. (%)	TP	FP	Prec. (%)
CodeQL	5110	0	100	7009	96	98.6
Gemini3.5-`ARQ`	7957	14	99.8	15406	246	98.4
Sonnet4.6-`ARQ`	5489	1	100.0	12805	266	98.0
GPT5.4-`ARQ`	4946	15	99.7	12457	195	98.5

Table 1: True Positives, False Positives, and Precision aggregated over all included queries. Per-query break down is in technical supplement.

Results Table 1 shows that `ARQ` generally improves query performance, though the exact gains depend on the underlying model. On FormAI, all three models substantially increase true positive detection over CodeQL (**Gemini3.5: +119.8%, Sonnet4.6: +82.7%, GPT5.4: +77.7%**). On Juliet, Gemini3.5 and Sonnet4.6 also improve over CodeQL (+55.7% and +7.4%, respectively), while **GPT5.4 shows a slight regression (-3.2%)**. Across all cases, the drop in Precision remains within 2.0%, benefiting from the incremental nature of Algorithm 1, a reasonable cost given the substantial gains in detected true positives. We further investigate the discrepancies caused by different models in § 4.4.

These results suggest that `ARQ`'s execution-validated refinement loop substantially improves query performance, and, in the worst case that it does not, regressions are kept in check.

Table 1 also illustrates our concerns about the difficulty in evaluating CodeQL queries. GPT5.4-`ARQ` underperformed on Juliet but performed well on FormAI. A sample in the dataset may be labeled CWE-121 (Stack-Based Buffer Overflow), but `BadlyBoundedWrite.ql`, `NoSpaceForZeroTerminator.ql`, and `OverflowCalculated.ql` can all be seen as CWE-121. In addition, these queries may also be CWE-122 (Heap-Based Buffer Overflow). In contrast, each query only detects the specific type of vulnerability mentioned in its name. No existing datasets are labeled according to query type. They are mostly labeled according to CWE types, which is too coarse for CodeQL queries.

4.2 RQ2: Real-World Impact

Beyond the evaluations on Juliet and FormAI in § 4.1, we validate `ARQ` on real, currently unresolved problems in widely-used software, where it matters most.

```
...
void process_buffer(char *buffer) {
    ...
}
void free_buffer(char *buffer) {
    if (buffer != NULL)
        free(buffer);
}
void use_after_free(char *buffer) {
    process_buffer(buffer);
}
...
```

Listing 1: Simplified version of provided code in issue 16542.

```
...
module UseAfterFreeParam implements
    FlowFromFreeParamSig {
  ...
  predicate sourceSinkIsRelated(DataFlow
      ::Node source, DataFlow::Node sink)
       {
    defaultSourceSinkIsRelated(source,
        sink)
    or
    isNonThisParameterUse(sink)
    // Fix: track function arguments.
  }
}
...
```

Listing 2: Simplified version of improved `UseAfterFree.ql`.

Fixing GitHub CodeQL Issues The most direct application of `ARQ` is fixing known, unresolved problems in queries described by open GitHub issues[2]. We selected the 3 most recent C/C++-query-related issues that have a minimally reproducible example, #21187 (opened for 6 months), #20577 (opened for 9 months), and #16542 (opened for 27 months). All three happen to concern `UseAfterFree.ql`. We skipped the generation phase in Algorithm 1 and directly used the examples supplied in each GitHub issue. **`ARQ`, backed by each of the three models, produced a working fix for all three issues**, an improved version of `UseAfterFree.ql` that correctly classified (manually verified) the provided examples.

The minimally reproducible example (Listing 1) incorporates a common code pattern where a pointer is passed through several functions before being used after it is freed. The original `UseAfterFree.ql` misses this because it requires `defaultSourceSinkIsRelated`. `ARQ` fixed this by extending control-flow tracking to function argument usage using `isNonThisParameterUse`. This produced a query that correctly flags the real bug reported in the issue.

Evaluating Queries on Open Source Projects We evaluated CodeQL queries and their refined counterparts on **seven widely-used C/C++ projects** (libpng, zlib, curl, redis, pcre2, Mbed TLS, and TinyXML-2), spanning around 985K lines of code. The refined queries identified 2 previously undiscovered bugs that the original CodeQL queries do not detect.

`InconsistentNullnessTesting.ql` found one previously undiscovered bug each in zlib[3] and libpng[4], two of the most widely used C libraries in the open-source ecosystem. zlib implements the compression routines bundled in nearly every Linux distribution and used by tools such as Git and rsync (**7K** GitHub stars), while libpng is the reference implementation for reading and writing the PNG image format (**1.6K** GitHub stars). The zlib bug has been present since **2017**, while the libpng bug has been present since **October 2025**. Both are hidden in cold code paths that are less frequently exercised and tested, which plausibly explains why neither had previously been caught.

Baseline	Juliet v1.3			FormAI v2		
	TP	FP	Prec. (%)	TP	FP	Prec. (%)
Gemini3.5	4970	0	100	7483	110	98.6
Sonnet4.6	2308	0	100	3146	40	98.7
GPT5.4	546	0	100	849	9	99.0

Table 2: Results for *baseline*, aggregated over the same query subset as Table 1. Per-query break down is in technical supplement.

4.3 RQ3: Ablation Study

`ARQ` is based on the belief that we can reduce hallucinations in LLMs by giving them a source of ground truth. In Retrieval Augmented Generation, the ground truth is the knowledge database. In `ARQ`, the ground truth comes from comparing $Analyze$ and $Oracle$. We investigate whether this belief is true.

Setup Procedures in § 4.1 are mimicked with one change. The agent no longer has access to $Analyze$ and $Oracle$ in Algorithm 1. This means that the agent cannot evaluate CodeQL queries on real programs, nor can it use the sanitizer. The agent still has access to the CodeQL Query Compiler and the LSP, so it can reliably produce syntactically correct queries.

We considered this approach the *baseline*, as it relies purely on the agents' reasoning capabilities.

Results Comparing Table 2 with Table 1, `ARQ` consistently outperforms this pure-LLM baseline, showing the clear advantage of $Analyze$ and $Oracle$. The baseline also regresses below vanilla CodeQL in **5 of 6** model/dataset configurations, compared to just **1 of 6** for `ARQ` (GPT5.4 on Juliet, § 4.1). Manual inspection of agent transcripts shows why. Without execution-grounded evidence, the agent sometimes hallucinates a weakness that does not exist, for example concluding that an already conservative CodeQL query still produces

[2]https://github.com/github/codeql/issues/

[3]https://github.com/madler/zlib/blob/da607da739fa6047df13e66a2af6b8bec7c2a498/adler32.c#L70

[4]https://github.com/pnggroup/libpng/blob/d1d0abeffede1cc898ddc3d0e600839cf026d749/contrib/tools/pngfix.c#L3936

Model	Success	No-Op	Failure	Timeout
Gemini3.5	51	6	0	3
Sonnet4.6	49	0	0	11
GPT5.4	42	2	15	1

Table 3: Outcomes for all iterations per model.

a false positive, then editing the query to "fix" it. Since there was no real weakness, this edit only removes true positives. Grounding LLMs via $Analyze$ and $Oracle$ through actual program execution is therefore not just beneficial—it is necessary.

4.4 RQ4: Efficiency

As mentioned in § 3.3, we delegated Algorithm 1 to coding agents. We assess how efficient this runs in practice.

Each query in § 4.1 was refined over 5 iterations, each given a 25-minute budget. We classify the outcome of each iteration into four types:

Success The agent reached line 31 and reported accordingly.

No-Op The agent reached line 19 and reported accordingly.

Failure The agent reached line 33 and reported accordingly.

Timeout The agent did not produce any valid output within time limit.

Across all iterations, agents ended with Success or No-op 83.3% of the time; the rest ended in a failure or timeout. GPT5.4 accounted for the most failures and, on manual inspection, tends to "give up," reporting that a query cannot be refined even when given validated witnesses; Sonnet4.6 never does this, always ending in success or timeout, while Gemini3.5 falls in between. This is likely one reason behind GPT5.4's regression on Juliet noted in § 4.1. The other likely reason being the lack of regression tests for original CodeQL queries ($W_r = \emptyset$ in first iteration).

On average, an iteration takes 433–690 seconds depending on the model, invokes the sanitizer 89–98% of the time, and produces correct witnesses 92–100% of the time. Because programs used in witnesses are small, `ARQ` is not CPU-bound, so nearly all time is spent waiting on the LLM. A full 5-iteration refinement completes within an hour.

5 Related Work

Pure LLM Vulnerability Detection LLMs have long been used for detecting vulnerabilities in code (Zhou et al. 2024). Standard LLM usage techniques such as prompt engineering (Zhang et al. 2024) and finetuning (Shestov et al. 2024) are both shown to be effective. There are also more specialized vulnerability detection techniques such as Retrieval Augmented Generation (Du et al. 2025). Regardless, these approaches mostly only handle a single function or at most a few files. In contrast, traditional static analysis tools can handle entire projects as they frequently instrument the build process. This is a great incentive to combine LLMs and traditional static analysis tools.

LLM-Assisted Static Analysis Studies about integrating LLMs with traditional static analysis tools for better vulnerability detection are rapidly emerging. For example, IRIS (Li, Dutta, and Naik 2024) and Artemis (Ji et al. 2025) use LLMs to label taint sources and sinks when performing the taint analysis. IRIS, specifically, demonstrated its power on Java web applications. However, taint analysis only applies to limited types of CWEs and does not capture memory-lifetime or pointer-aliasing reasoning, which most C/C++ vulnerability classes require. KNighter (Yang et al. 2025) employs a similar iterative refining process, but relies on commit history and is very project-specific. On the other hand, our approach refines the query using test programs generated independently from the application. MoCQ (Li et al. 2025) is designed for general bug patterns, but is focused on generating a correct query using symbolic query validation. Our approach builds on top of this and uses sanitizers to make sure the query is not only syntactically correct but also semantically effective, and does so with significantly fewer refinement iterations.

LLMs as Agents in Software Engineering Using LLMs as agents to interact with standard SWE tools (Yang et al. 2024) is not a new practice. While we mostly used sanitizers, previous attempts have been using LLMs to interact with other program reasoning tools like fuzzers (Lemieux et al. 2023). Interactions with language servers are used to enhance code generation (Go et al. 2025; Blinn et al. 2024). After vulnerability detection, LLM agents can also be applied to perform automated program repair (Xia, Wei, and Zhang 2023; Joshi et al. 2022).

6 Conclusions

We presented `ARQ`, an agentic framework for automatically refining static analysis queries through execution-validated program synthesis and iterative LLM-guided reasoning. Across 12 CodeQL queries on Juliet and FormAI, `ARQ` substantially improves true positive detection while keeping Precision above 98.0%. Beyond these benchmarks, `ARQ` fixed 3 previously unresolved GitHub issues in the CodeQL repository and found 2 real, previously undiscovered bugs in 7 widely-used C/C++ projects. These results show that execution-guided, example-driven query refinement is a practical and effective strategy for improving static analysis at scale, without requiring manual specifications or labeled vulnerability datasets.

Looking forward, `ARQ`'s core idea is not specific to CodeQL or to C/C++. Grounding an LLM's reasoning in program execution, rather than in static argument alone, applies to any static analysis tool with a runtime signal to check against, and to any language with sanitizer-like tooling (C# provides an overflow detector (Microsoft 2026a); Go provides a data race detector (Google 2026b)). As query suites grow large and expensive to maintain by hand, an automated way to keep queries up-to-date, without waiting for a human to notice and report an issue, could meaningfully reduce the maintenance burden on tools like CodeQL and help them stay effective as codebases and vulnerability patterns evolve.

References

Anthropic. 2026. Claude Code by Anthropic.

Avgustinov, P.; De Moor, O.; Jones, M. P.; and Schäfer, M. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, 2–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Backes, M.; Rieck, K.; Skoruppa, M.; Stock, B.; and Yamaguchi, F. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 334–349.

Black, P. E.; and Black, P. E. 2018. *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology.

Blinn, A.; Li, X.; Kim, J. H.; and Omar, C. 2024. Statically Contextualizing Large Language Models with Typed Holes. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2): 468–498.

Chen, Y.; Ding, Z.; Alowain, L.; Chen, X.; and Wagner, D. 2023. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th international symposium on research in attacks, intrusions and defenses*, 654–668.

Chibotaru, V.; Bichsel, B.; Raychev, V.; and Vechev, M. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 760–774. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367127.

CodeQL. 2026. CodeQL.

Ding, Y.; Fu, Y.; Ibrahim, O.; Sitawarin, C.; Chen, X.; Alomair, B.; Wagner, D.; Ray, B.; and Chen, Y. 2024. Vulnerability Detection with Code Language Models: How Far Are We? arXiv:2403.18624.

Du, X.; Zheng, G.; Wang, K.; Zou, Y.; Wang, Y.; Deng, W.; Feng, J.; Liu, M.; Chen, B.; Peng, X.; Ma, T.; and Lou, Y. 2025. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. arXiv:2406.11147.

Dubniczky, R. A.; Horvát, K. Z.; Bisztray, T.; Ferrag, M. A.; Cordeiro, L. C.; and Tihanyi, N. 2025. CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs towards CWE Detection. arXiv:2503.09433.

Free Software Foundation. 2026. Program Instrumentation Options.

Go, G.; Zhou, C.; Zhang, Q.; Jiang, Y.; and Wei, Z. 2025. LSPAI: An IDE Plugin for LLM-Powered Multi-Language Unit Test Generation with Language Server Protocol. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, 144–149. New York, NY, USA: Association for Computing Machinery. ISBN 9798400712760.

Google. 2026a. CodeQL & Chromium.

Google. 2026b. Data Race Detector.

Google. 2026c. Google Antigravity.

Ji, Y.; Dai, T.; Zhou, Z.; Tang, Y.; and He, J. 2025. Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-procedural Path-Sensitive Taint Analysis. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1): 1349–1377.

Joern. 2026. Joern.

Joshi, H.; Cambronero, J.; Gulwani, S.; Le, V.; Radicek, I.; and Verbruggen, G. 2022. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. arXiv:2208.11640.

Jr, F. E. B.; and Black, P. E. 2012. The Juliet 1.1 C/C++ and Java Test Suite. *NIST*, 45(10): 88–90. Last Modified: 2021-10-12T11:10-04:00 Publisher: Frederick E. Boland Jr., Paul E. Black.

Lemieux, C.; Inala, J. P.; Lahiri, S. K.; and Sen, S. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, 919–931. IEEE Press. ISBN 9781665457019.

Li, P.; Yao, S.; Korich, J. S.; Luo, C.; Yu, J.; Cao, Y.; and Yang, J. 2025. Automated Static Vulnerability Detection via a Holistic Neuro-symbolic Approach. arXiv:2504.16057.

Li, Z.; Dutta, S.; and Naik, M. 2024. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*.

Microsoft. 2026a. C# Compiler Options for language feature rules.

Microsoft. 2026b. CodeQL extension for Visual Studio Code.

NIST Information Technology Laboratory. 2026. National Vulnerability Database.

OpenAI. 2026. Codex in ChatGPT.

Semgrep. 2026. Semgrep.

Shestov, A.; Levichev, R.; Mussabayev, R.; Maslov, E.; Cheshkov, A.; and Zadorozhny, P. 2024. Finetuning Large Language Models for Vulnerability Detection. arXiv:2401.17010.

Shi, Y.; Zhang, Y.; Bai, T.; Zhang, L.; Tan, X.; and Yang, M. 2024. RecurScan: Detecting Recurring Vulnerabilities in PHP Web Applications. In *Proceedings of the ACM Web Conference 2024*, WWW '24, 1746–1755. New York, NY, USA: Association for Computing Machinery. ISBN 9798400701719.

Smirnov, I. B. 2007. Raw pointers in application classes of C++ considered harmful. *SIGPLAN Not.*, 42(4): 23–31.

Stroustrup, B.; and Sutter, H. 2026. C++ Core Guidelines.

The LLVM Foundation. 2026. Clang Compiler User's Manual.

Tihanyi, N.; Bisztray, T.; Ferrag, M. A.; Jain, R.; and Cordeiro, L. C. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(47).

Wang, C.; Li, Z.; Dutta, S.; and Naik, M. 2026. QLCoder: A Query Synthesizer For Static Analysis of Security Vulnerabilities. arXiv:2511.08462.

Xia, C. S.; Wei, Y.; and Zhang, L. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1482–1494.

Yang, C.; Zhao, Z.; Xie, Z.; Li, H.; and Zhang, L. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, 655–669. ACM.

Yang, J.; Jimenez, C. E.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.; and Press, O. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793.

Zhang, C.; Liu, H.; Zeng, J.; Yang, K.; Li, Y.; and Li, H. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. arXiv:2308.12697.

Zhou, X.; Cao, S.; Sun, X.; and Lo, D. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. arXiv:2404.02525.