HapCheck: DSL-Based Static Bug Detection Framework for OpenHarmony

Xitong Zhong[1], Chang Liu[1], Runlin Liu[1], Zifu Xu[1], Zhengyao Liu[1], Juqi Zhou[1], Gang Fan[2], Mingyi Zhou[1], Xiang Gao[1, *], Li Li[1, *]

[1]Beihang University, Hangzhou Innovation Institute of Beihang University {xitongzhong, serfend, runlin22, zifu2022, zhengyaoliu, berry0303, zhoumingyi, xiang_gao}@buaa.edu.cn, lilicoding@ieee.org, China

[2]Independent Researcher, fan.gang.cn@gmail.com, China

Abstract

Defect detection in HarmonyOS applications, which are primarily developed using ArkTS (a TypeScript-based superset that incorporates unique features and UI semantics), remains under-supported by existing static analysis tools. Most tools rely on rigid, low-level APIs, making it difficult for developers to define complex or high-level semantic rules efficiently. To address this gap, we propose HapCheck, a novel framework that introduces an embedded domain-specific language (DSL) tailored for ArkTS. This DSL enables concise and expressive rule definitions, abstracting low-level analysis details through a fluent API. HapCheck supports both native and engine-agnostic rule specifications and integrates with heterogeneous analysis backends via a modular DSL Adapter. To enhance detection precision, we further develop a semantic rule execution engine based on Code Property Graphs (CPG), equipped with lifecycle-aware control flow modeling for HarmonyOS applications. Our evaluation of 258 real-world ArkTS projects reveals the effectiveness of HapCheck. The bug detector based on our HapCheck uncovers more than 4,000 defects, including 197 critical security vulnerabilities, with minimal false positives. Moreover, HapCheck reduces the number of rule definition lines by 65% on average and improves rule readability and expressiveness, as confirmed by user studies. HapCheck is integrating into Huawei's IDE as part of the official Harmony application development toolkit.

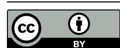
Keywords

Static Analysis, Defect Detection, HarmonyOS, Domain-Specific Language, Code Property Graph

ACM Reference Format:

Xitong Zhong[1], Chang Liu[1], Runlin Liu[1], Zifu Xu[1], Zhengyao Liu[1], Juqi Zhou[1], Gang Fan[2], Mingyi Zhou[1], Xiang Gao[1, *], Li Li[1, *] . 2026. HapCheck: DSL-Based Static Bug Detection Framework for OpenHarmony. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786583.3786895

* Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ICSE-SEIP '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786895

1 INTRODUCTION

OpenHarmony is an open-source operating system, while HarmonyOS is its commercial counterpart developed by Huawei. As of now, HarmonyOS has 10 billion connected devices [14, 21]. HarmonyOS Apps are mainly developed by ArkTS, a superset of TypeScript enhanced with system-level capabilities and a declarative UI framework called ArkUI. Like other programming languages (e.g., Typescripts) constantly suffering from code defects or vulnerabilities, the ArtTS will likely encounter the same problems.

Automated defect detection plays a crucial role in modern software development, enabling developers to identify potential issues before deployment and thus mitigate costly failures [5, 8, 15, 19, 20, 25, 27]. Among existing approaches, rule-based static analysis tools are most widely adopted due to their flexibility, usability and interpretability [6, 7, 18, 32]. These tools typically rely on user-defined defect rules to scan the source code and identify problematic patterns. For instance, CodeQL [1] has attracted attention for its success in program analysis and defect detection [10, 28, 40]. Developed and maintained by GitHub, CodeQL saves source code into graph database and enables defect detection through dataset query. CodeQL simplifies complex rule definition by defining a declarative, SQL-like domain-specific language (DSL) [24]. However, existing static defect detectors are not compatible with ArkTS programs due to the unique features of ArkTS, such as the declarative UI framework and syntactic sugar [5, 16].

HomeCheck[2] is the state-of-the-art tool for detecting defects in ArkTS. By defining syntactic patterns, HomeCheck can identify security, performance, or quality issues, and accurately pinpoints their locations. However, HomeCheck lack supports to detect complex defects involving control or data flow analysis. Moreover, defining defect rules is notably complicated for HomeCheck. As illustrated in Figure 1, which demonstrates a rule that flags the usage of `for-in` loops, much of the code (Lines 3–8 and Line 13) is devoted to traversing the abstract syntax tree (AST), while only a small portion (Lines 10–11) implements the actual rule logic. This inefficiency stems from HomeCheck's reliance on low-level AST manipulation without providing a developer-friendly interface. Therefore, rule definitions tend to be verbose and difficult to maintain.

To address these challenges, we propose HapCheck, a framework specifically designed for defect detection in ArkTS. HapCheck introduces a high-level DSL that encapsulates commonly used analysis patterns, reducing the boilerplate code required for rule development. To align with the habits of ArkTS developers, the DSL is

[1]https://github.com/github/codeql
[2]https://gitcode.com/openharmony-sig/homecheck

```
export class NoForInArrayCheck {
    public check = (arkFile: ArkFile) => {
        const astRoot = AstTreeUtils.
            getSourceFileFromArkFile(arkFile);
        for (let child of astRoot.statements) {
            this.isForInCheck(child);
        };
    };
    isForInCheck = (node: ts.Node): void => {
        if (ts.isForInStatement(node)) {
            this.handleForInStatement(node);
        };
        ts.forEachChild(node, this.isForInCheck);
    };
    < ...omitted code... >
}
```

Figure 1: Rule Implemented Using Native HomeCheck

embedded directly in TypeScript and offers a Fluent API for intuitive rule composition. Listing 1 shows a rule implemented using our DSL, which is semantically equivalent to the rule in Listing 1 but significantly more concise and readable. HapCheck is designed to be compatible with existing static analysis tools such as HomeCheck. To this end, we introduce a DSL Adapter that decouples rule definition from underlying execution engines, enabling seamless integration with multiple backend engines, including HomeCheck. Moreover, inspired by CodeQL, Joern [29], QVoG [8], HapCheck features a semantic rule execution backend engine built on code property graphs (CPG), tailored to ArkTS [4, 22, 37, 41]. Developing a CPG-based engine for ArkTS presents additional challenges due to language-specific features. In particular, the declarative nature of ArkUI introduces non-traditional control flow behaviors triggered by lifecycle events and UI callbacks. To address these problem, we design a lifecycle-aware control flow model that captures event-driven execution paths and component lifecycle transitions, enabling more precise static analysis [2, 9, 36, 39].

```
export default native<HomeCheckHint>()()
    .match(ForInStatement)
    < ...omitted code... >
```

Listing 1: Rule Implemented Using our DSL

On top of the HapCheck framework, we develop a defect detector including 22 rules, spanning performance, security, and quality issues. We evaluate our approach on 258 real-world ArkTS projects. The results demonstrate that our tool effectively detects a wide range of practical defects, identifying 4,278 issues across 177 projects. Among the 45 security defects that required inter-statement analysis and subjected to manual validation, only 5 were determined to be false positives, indicating a low false positive rate of 11.4% in this subset. HomeCheck achieves a low analysis latsency (*i.e.*, no more than 20 seconds) even on 100K-LOC projects, while the CPG engine enables deeper inter-statement reasoning. Furthermore, our DSL reduces the number of code lines required for rule definitions by 65% on average and is consistently rated higher in expressiveness, clarity, and conciseness by practitioners, significantly lowering the barrier for custom rule development. In summary, our contributions are as follows.

- We design an ArkTS defect detection framework, including an embedded DSL to simplify rule definition, and a DSL adapter to support multiple analysis engines through a modular architecture.
- We design a code property graph-based rule execution engine tailored to ArkTS, supporting semantic analysis in the presence of ArkUI-specific language features.
- We discovered over 4,000 code defects in real-world projects, including 197 security vulnerabilities.
- We open-sourced the DSL[3], CPG engine[4], and Ark2Graph[5]. We also adapted the DSL support into HomeCheck and released this integration publicly[6].

2 METHODOLOGY

In this section, we provide a detailed introduction to HapCheck.

2.1 Framework Design

The overall architecture of HapCheck is illustrated in Figure 2, which comprises three primary components: DSL Rule Definition, DSL Rule Translation, and DSL Rule Execution.

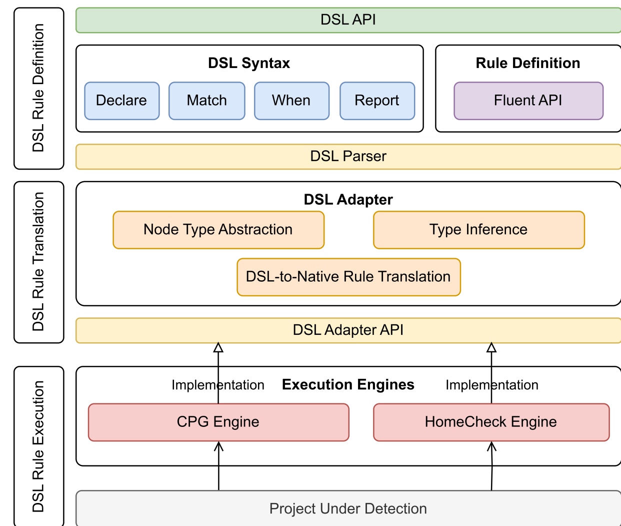


Figure 2: Overall Design of the HapCheck Framework

DSL Rule Definition. To facilitate rule definition, HapCheck introduces a user-friendly embedded DSL that abstracts away low-level execution details. The DSL is designed to be engine-agnostic, allowing rules to be reused across multiple backend engines. HapCheck is embedded into TypeScript, enabling ArkTS developers to define rules using their familiar language TypeScript and hence reduce learning curve of DSL syntax. Additionally, HapCheck leverages TypeScript's type inference mechanisms to streamline rule specification, allowing developers to omit explicit type annotations while preserving type correctness across multiple clauses.

DSL Rule Translation. The DSL Adapter bridges high-level DSL rules and various DSL execution engines. Since each engine operates with distinct semantics, HapCheck captures contextual information through closures during translation. The translation rule is then loaded using a callback mechanism to preserve semantic correctness and ensure compatibility with the target engine.

[3] https://gitcode.com/QVoG-BUAA/qvog-dsl
[4] https://gitcode.com/QVoG-BUAA/qvog-engine
[5] https://gitcode.com/QVoG-BUAA/Ark2Graph
[6] https://gitcode.com/QVoG-BUAA/homecheck

$r \in$ DslRule	$:=$	$d\ m\ [w]\ p$
$d \in$ DeclareClause	$:=$	$\mathrm{r}_{\mathrm{generic}}$
	$\mid$	$\mathrm{r}_{\mathrm{native}}\langle e\rangle([options])$
$m \in$ MatchClause	$:=$	match(TypePredicate $\mid a$)
$a \in$ MatchAction	$:=$	$(nodes,\ predicate) \Rightarrow nodes$
$w \in$ WhenClause	$:=$	when(h)
$h \in$ WhenAction	$:=$	$(nodes,\ predicate) \Rightarrow nodes$
$p \in$ ReportClause	$:=$	report(IssueTemplate $\mid q$)
$q \in$ ReportAction	$:=$	$(nodes,\ report) \Rightarrow$ void
$e \in$ Engine	=	DSL execution engine
options $\in$ OptionList	=	engine configuration
node $\in$ Node	=	AST or ArkIR node
pred, *report* $\in$ Function	=	user-defined utilities
$i \in$ Issue	=	issue descriptor

Figure 3: Schematic Diagram of DSL Syntax Definition

DSL Rule Execution. Rule execution refers to the process of applying translated DSL rules to source code or intermediate representations to detect defects. By default, HapCheck supports two execution engines: the HomeCheck engine and the CPG engine. The HomeCheck engine directly executes translated rules on the AST of ArkTS source code or ArkIR. In contrast, the CPG engine first converts ArkTS code into a CPG, embedding language-specific semantics. It then transforms the DSL rule into a set of graph queries, turning defect detection into graph query tasks.

2.2 Embedded DSL

The primary objective of the DSL design is to provide a concise and expressive mechanism for specifying ArkTS defect detection rule, while abstracting away the complexities inherent in native rule implementations. Specifically tailored for ArkTS, this design allows developers to formulate defect detection rules in a high-level, declarative style. By separating rule definitions from low-level implementation details, the DSL promotes reusability and portability across heterogeneous static analysis engines.

2.2.1 DSL Syntax. The formal grammar of the DSL, specified in abstract syntax definition, is presented in Figure 3, followed by a detailed explanation of each clause.

Declare Clause. The DSL is designed to support multiple execution engines, addressing the diverse requirements of defect detection in ArkTS. Simple defects, such as syntax errors or basic rule violations, can be efficiently detected using lightweight syntactic pattern matching, offering fast but limited analysis capability. In contrast, identifying complex defects—such as those requiring data-flow or control-flow analysis—demands more sophisticated and computationally intensive techniques. By supporting different execution engines, the DSL is enabled to handle varying defect types and be adapted to different workflows, such as real-time IDE feedback or large-scale CI/CD pipelines.

To decouple DSL rule specifications from engine-specific implementations, we propose a unified abstraction layer that standardizes execution engine's functionalities. This abstraction encapsulates essential analysis services provided by each engine, such as node matching and diagnostic reporting. By isolating engine-specific capabilities behind a common interface, the framework allows rule developers to select execution backends based on functional requirements rather than implementation details, thereby enhancing the flexibility and usability of the rule definition process.

To support diverse definition scenarios, we define two declarative variants of rule declaration:

- Generic Form ($r_{generic}$): An engine-agnostic form ensures cross-backend portability but is limited to the feature intersection of all backends.
- Native Form (r_{native}): Targets a specific engine, granting access to engine-specific capabilities and optimizations that are not universally supported.

Rule r_{native} takes inputs as two parameters: *Engine*: *e* and engine configuration: $options$, where $options$ specifies engine-specific configuration parameters. For example, in the case of the `HomeCheck` engine, users can specify via the $options$ whether the analysis should traverse source or Ark intermediate representation (ArkIR). The available configuration options are determined by the target engine.

Match Clause. defect detection begins by identifying AST nodes that may be relevant to potential issues, enabling targeted analysis. The Match clause serves as an efficient pre-filter, selecting candidate nodes during scanning to optimize performance in static analysis workflows. It accepts either a TypePredicate for basic node type filtering, or a MatchAction that evaluates nodes against user-defined predicate. This dual mechanism supports both lightweight type checks and complex pattern matching, offering an optimal balance between speed and flexibility. This clause outputs the set of all nodes satisfying the specified matching criteria.

When Clause. The When clause performs in-depth analysis on the nodes selected by the Match clause, applying user-defined validation predicates. These predicates can examine either: patterns within individual nodes, or semantic relationships between multiple nodes. For example, when detecting sensitive data leaks, the clause might analyze control and data flow relationships between source (sensitive data origin) and sink (potential leak point) nodes. The When clause ultimately returns the set of problematic nodes that violate the specified conditions. This clause is optional, allowing for rules where Match alone suffices for issue detection.

Report Clause. Upon identifying problematic nodes via the When clause, the Report clause aggregates relevant details, such as file paths, locations, and descriptions, and reports the issue. It supports both predefined IssueTemplate templates and user-defined reporting logic. The latter allows users to compose customized reports using the filtered *nodes* and APIs through the *ecap*. Completion of the Report clause finalizes the DSL rule, allowing a complete, executable artifact for integration into code analysis pipelines.

Moreover, as discussed in Section 1, conventional approaches to defining code defect detection rules frequently employ domain-specific syntax that substantially deviates from mainstream programming. This disparity imposes considerable overhead for developers to learn how to write rules. Given that ArkTS is a superset of TypeScript, developers proficient in ArkTS are inherently familiar with the TypeScript programming paradigm. To address this gap and make the tool more accessible, HapCheck is embedded into TypeScript to enable clear, expressive DSL rule definitions.

2.2.2 Fluent Interface. In the design of embedded DSLs, Fluent Interface pattern [33] is adopted for constructing expressive and readable pattern. By enabling method chaining through the return of the current or contextually related objects, this pattern facilitates a natural syntax that closely resembles human language. Furthermore, the fluent design aligns well with language tooling ecosystems, such as the TypeScript Language Server, enabling syntax-aware auto-completion and interactive feedback during DSL rule definition. These affordances are particularly valuable in scenarios that require iterative definition and debugging of DSL rules, where quick validation and feedback loops are critical.

```
1  export default native<HomeCheckHint>()()
2      .match(ForInStatement)
3      .when((nodes: ts.ForInStatement[]) => {
4          return nodes.filter(node =>
5              !is(node.expression,
                   ObjectLiteralExpression)
6          );
7      })
8      .report({
9          severity: SEVERITY,
10         description: DESCRIPTION,
11         docPath: DOC_PATH,
12     });
```

Listing 2: Rule Definition of NoForInArrayCheck

Listing 2 presents a complete rule definition written in our DSL. This example demonstrates a rule designed to detect potentially unsafe usages of the for-in loop. While for-in is a valid construct in ArkTS, it may skip array holes or unintentionally traverse properties inherited through the prototype chain[7]. However, this behavior is safe when the for-in loop iterates over object literals. Specifically, this rule consists of four core components that map to the DSL clauses. Line 1 corresponds to the Declare clause, which specifies HomeCheck as the target execution engine. The Match clause at Line 2 identifies all for-in loop statements and passing the matched nodes to subsequent clauses. Lines 3–7 represent the When clause, where additional filtering logic is applied to exclude cases where the loop iterates over an object literal, which is considered safe. The remaining lines form the Report clause. Here, the rule utilizes the Issue template provided by the DSL, which incorporates both user-defined properties—such as severity and description—and built-in functionality for pinpointing the defect location in the source code.

2.3 DSL Adapter

The DSL Adapter serves as a bridge between high-level DSL rule specifications and the heterogeneous execution environments that interpret and apply these rules. Its design focuses on two core aspects: (1) the unified abstraction of node types across diverse intermediate representations, and (2) the translation of DSL rules into native rules compatible with specific analysis engines.

2.3.1 Unified Node Type Abstraction. A central challenge in DSL-based static analysis is the reconciliation of varying type systems across different representations, such as AST of source code and ArkIR. To address this problem, the DSL Adapter introduces a unified abstraction mechanism to encapsulate native node types under a standardized interface.

While primitive type checks suffice for simple patterns, more complex program structures often require richer semantics. To accommodate such cases, the DSL framework allows users to define custom node types by composing existing ones. Listing 3 demonstrates a user-defined type *ConditionalOrBinary*, which captures both conditional and binary expressions. This abstraction enables more flexible and semantically meaningful rule definitions.

```
1  type ConditionalOrBinaryNode = ts.
       ConditionalExpression | ts.BinaryExpression;
2  const ConditionalOrBinary: TypePredicate<
       ConditionalOrBinaryNode> = {
3      is(node: ts.Node): node is ConditionalOrBinaryNode
           {
4          return ts.isConditionalExpression(node) || ts.
               isBinaryExpression(node);
5      },
6  };
```

Listing 3: Custom Node Type Definition

Beyond manual declarations, the DSL framework also supports compositional abstractions using logical operators. As shown in Listing 4, the same node type can be defined via composition using the *or* operator. Similar operators (*and*, *not*) support intersection and negation, respectively. This capability enables users to build expressive, reusable predicates for complex type hierarchies, boosting rule definition modularity and scalability.

```
1  const ConditionalOrBinary = or(ConditionalExpression,
       BinaryExpression);
```

Listing 4: Composition of Node Types

2.3.2 Type Inference. To enhance usability and safety in rule definition, HapCheck leverages TypeScript's type system to enable automated and modular type inference across clauses. Unlike prior DSL implementations that rely on manual annotations, we design our rule language to be type-aware by construction, minimizing boilerplate and preventing type mismatches at compile time.

Our DSL lies in the mechanism for propagating the type between the clauses. For instance, the node type deduced in the Match clause is automatically inferred—via TypeScript's type predicates—and seamlessly propagated to downstream clauses When and Report. This eliminates the need for users to redundantly re-specify types, and ensures that all operations within a rule remain type-safe and contextually valid. In addition, we leverage advanced features of TypeScript generics, including `extends` and `infer`, to enable flexible yet sound type derivations in reusable rule components. These generics allow us to model parameterized node structures and predicates, making the DSL both expressive and strongly typed.

[7] https://developer.huawei.com/consumer/en/doc/harmonyos-guides/ide_no-for-in-array

2.3.3 DSL-to-Native Rule Translation. A central challenge in DSL-based static analysis frameworks is translating high-level declarative rules into engine-specific code suitable for execution. Since analysis engines often differ significantly in how rules are defined and registered—ranging from declarative specs to imperative class-based APIs—this translation must be generalizable and adaptable.

To address this problem, we introduce a DSL Adapter providing a unified, extensible rule translation layer. Instead of immediate translation, the adapter defers native rule construction to execution time, enabling dynamic integration of engine-specific capabilities. This mechanism preserves the semantic intent of the original DSL rule while ensuring compatibility with diverse engine architectures. By decoupling rule declaration from execution, the adapter supports flexible, context-aware rule development and promotes reusability across different execution engines.

2.4 Execution Engine

To support DSL execution, we implemented two engines: HomeCheck engine and CPG Engine. Since HomeCheck engine is directly implemented using existing HomeCheck, we only present the details of CPG engine. The CPG engine transforms the target project into a CPG and enables DSL execution by traversing the CPG.

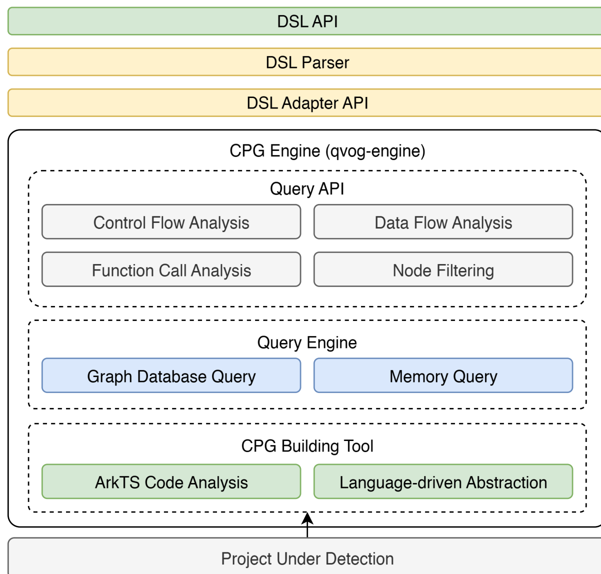


Figure 4: Architecture of CPG engine

The architecture of CPG engine is illustrated in Figure 4. The engine first performs static code analysis on the target ArkTS project, accounting for language-specific characteristics. While mature CPG tools exist for mainstream languages such as C and Java, these tools are not directly applicable to ArkTS, due to its unique language features. To address this, we developed Ark2Graph, a dedicated CPG construction tool for ArkTS. ArkGraph adapts well-established program analysis techniques from other languages and enhances them to handle ArkTS-specific challenges such as event-driven control flow and UI lifecycle semantics.

Ark2Graph is built on top of ArkAnalyzer [5], a widely adopted static analysis framework for ArkTS. ArkAnalyzer converts source programs into three-address intermediate representation (ArkIR), simplifying control and data flow extraction. Building the CPG at the statement level of ArkIR, rather than the AST, strikes a balance between preserving semantic fidelity and improving analysis efficiency. Using ArkAnalyzer, we extract both control flow graphs (CFGs) and data flow graphs (DFGs) from the ArkIR representation. Specifically, basic blocks identified in ArkIR provide the structural basis for constructing CFG, capturing the potential execution paths within a function or program module. In parallel, HapCheck builds DFG by leveraging ArkAnalyzer's built-in def-use analysis, which identifies the propagation of data values across program statements. Compared to source-level analysis, operating on ArkIR reduces complexity by abstracting away syntactic irregularities inherent to ArkTS, such as lexical scoping and object-oriented constructs, while preserving essential semantic information.

While this approach provides a robust foundation for analyzing core language constructs, it is insufficient for modeling higher-level event-driven behaviors, particularly those introduced by ArkUI, ArkTS's declarative user interface framework. In contrast to traditional applications with well-defined entry points and linear control flow, ArkUI applications rely heavily on user interactions to trigger asynchronous callback functions, which dynamically alter program state and execution flow. This paradigm shift introduces a significant challenge in static control-flow modeling, as the invocation order and execution paths are not explicitly defined in the source code. For example, consider the following code snippet:

```
1. Button('Button1')
2.     .onClick(() => { this.state = 1; }
3. Button('Button2')
4.     .onClick(() => { this.state = 2; }
```

The execution order and possibility of Line 2 and Line 4 are non-deterministic and state-dependent. Traditional static control flow analyses fail to capture such behavioral uncertainty, potentially leading to false negatives in downstream analyses.

To overcome this, we designed a lifecycle-aware control flow model inspired by Android's event-driven analysis techniques [11]. This model treats ArkUI-registered callback functions as potential invocation sites and abstracts the entire ArkUI component as an infinite loop with a non-deterministic switch-case structure, where any callback may be invoked at each iteration.

While this abstraction maximizes coverage, it introduces scalability issues due to an explosion in possible control paths. To mitigate this, we incorporate ArkUI lifecycle constraints, such as `aboutToAppear`, `build`, `onDidBuild`, and `aboutToDisappear`. We further account for parent-child rendering orders to accurately prune infeasible paths and reduce query space.

Figure 5 presents an example of our abstraction process. In this case, an ArkUI implementation is transformed into a standard ArkTS representation, significantly reducing the complexity of subsequent code analysis. In the original code, the variable *title* is accessed by both the *aboutToAppear* lifecycle hook and the *Text* component. The variable is defined during component initialization and within the callback triggered by user interaction with the *Text* element. Due to lifecycle semantics, the value of *title* accessed by *aboutToAppear* can only originate from the component initialization phase, while the value accessed by *Text* may originate from

```
@Component
struct Hello {
  @State title: string = 'Hello';
  aboutToAppear() {
    console.info(this.title);
  }
  build() {
    Text(this.title)
      .onClick(() => {
        this.title = 'ArkUI';
      })
  }
}
```

```
class Hello {
  title: string = 'Hello';
  constructor() {
    this.aboutToAppear();
    this.build();
  }
  build() {
    while(true) {
      Text(this.title);
      if (case1)
        this.title = 'ArkUI';
      else break; } }
}
```

Figure 5: An example of abstracting ArkUI code. The left snippet illustrates the original ArkUI implementation. The component displays a text bound to *title*, which updates its value upon user interaction. The right snippet presents the abstracted but semantically equivalent representation.

the initial assignment or the callback function invoked on a click event. Crucially, both the control-flow and data-flow semantics are preserved throughout the abstraction, thereby demonstrating the correctness and fidelity of our abstraction.

After constructing the code property graph, the CPG engine abstracts over the underlying graph database traversal and exposes a suite of streaming query APIs. These APIs enables users to express more sophisticated and precise query logic during rule definition.

3 IMPLEMENTATION

The HomeCheck engine supports defect detection at both the source code and intermediate representation (IR) levels. For source code analysis, it provides direct AST inspection. For IR-level analysis, the engine leverages the comprehensive APIs offered by ArkAnalyzer. These two analysis modes operate independently, offering flexible and decoupled solutions for different use cases.

The CPG engine follows a three-stage pipeline: construction, storage, and querying. During construction, ArkAnalyzer generates the CPG from ArkIR. The resulting CPG is then persisted in Neo4j[26], a robust graph database, using Gremlin Server[23] for efficient dataset access. In the querying phase, the engine retrieves relevant nodes and edges through Gremlin Server to perform rule-based pattern matching and vulnerability detection. To optimize performance for repetitive executions, we implemented a *CPG-MemCache* mode. This enhancement pre-builds and caches the CPG in memory, bypassing both repeated graph construction and Neo4j storage operations. This optimization is particularly valuable in batch processing and continuous integration scenarios, where the same codebase undergoes multiple rule executions.

To simplify DSL rule development, we adopted a modular approach inspired by CodeQL's query library (.qll) system. Our DSL library encapsulates a comprehensive set of reusable functions covering essential code elements including methods, literals, and expressions. This design promotes logic reuse, improving both rule-writing efficiency and user convenience.

4 EVALUATION

To investigate the effectiveness of our tool, we propose the following research questions and conduct experiments accordingly:

- **RQ1:** Can HapCheck identify defects in real-world projects?
- **RQ2:** How does the execution time of HapCheck vary with different code scales?
- **RQ3:** Can DSL simplify the writing of defect rules?
- **RQ4:** What is the effectiveness of lifecycle handling?

All experiments were conducted on a machine running Windows, equipped with a 12-core CPU and 16 GB of RAM.

Table 1: Statistics of Representative Projects

Application Name	Category	Source	LOC
harmonyVideoPlayer	Video player	Gitcode	789
ohos_apng	Animation	Gitcode	3,462
cqeditor	Editor	Gitee	3,575
DimensionalPocket	Tool kit	Gitee	4,949
uctooapp	Database	Gitee	456,711

4.1 Dataset

To build the ArkTS project dataset, we searched for the Harmony project on GitHub, GitCode, and Gitee platforms using keywords "ArkTS" and "Harmony". The final dataset[8] comprises 258 projects, including 181 actively maintained ones that have at least one commit in the last six months (post-April 2025). The dataset contains 963,589 LOC (Lines of Code) of ArkTS, 237,384 LOC of JavaScript, and 119,415 LOC of TypeScript, totaling 1.32 million lines of source code. The selected projects span multiple domains, including animation, database management tools, and video players. Table 1 lists some representative projects with lines of code ranging from a few hundred to tens of thousands.

4.2 RQ1: Real-world Defect Detection

To validate whether HapCheck can effectively detect real-world vulnerabilities, we developed a set of detection rules and performed a complete detection on the dataset to verify the efficacy of HapCheck.

4.2.1 Rule Development. We developed a total of 22 rules for our evaluation, which are defined based on the OpenHarmony/HarmonyOS programming specifications and the ArkTS best practices [13]. These rules are listed in Table 2 and categorized into three major dimensions: performance(perf.), quality and security. Performance rules focus on detecting issues that may decrease system efficiency and responsiveness. The quality rules emphasize code maintainability, readability, and reliability, while security rules are intended to detect potential vulnerabilities and prevent malicious attacks.

In addition, our rules can be categorized into two types: intra-statement rules and inter-statement rules. For the former, the rule logic is typically straightforward and is caused by a single statement, which can be implemented by both the HomeCheck engine and CPG engine. In contrast, inter-statement rules check the relation between multiple statements, such as the data-flow relation between a sensitive information source and sink. Inter-statement rules can only be implemented using the CPG engine. Together, these rules provide a comprehensive framework for detecting potential defects in real-world projects.

[8] https://gitcode.com/QVoG-BUAA/DataSet

Table 2: Rules and Detection Results

Rule Name	Description	Intra/Inter Statement	Category	#Detected Defects
no-key-generator-in-foreach	Missing key generator degrades perf by triggering full component rebuild on new array items.	Intra	perf	1,273
console-in-high-freq-func	Using console.log in high-frequency funcs can degrade perf due to sync I/O blocking the main thread.	Intra	perf	6
about-to-appear-never-dispose	Objects created in the aboutToAppear lifecycle method may cause memory leaks and performance degradation if not properly released or destroyed later.	Intra	perf	28
avoid-frequent-state-update	Avoid frequent state updates in high-frequency event handlers to enhance performance.	Inter	perf	43
ban-types	Using String, Boolean, Number, Symbol, Object, Function, and BigInt as types can lead to confusion with TypeScript's native types, potentially causing unexpected behavior.	Intra	quality	1,649
no-non-null-assertion	Use ! operator may cause runtime TypeErrors if it is used on null or undefined values.	Intra	quality	585
no-cond-assign	Assignment and comparison operations look very similar, making them prone to logical errors.	Intra	quality	4
no-useless-catch	May cause potential stack trace pollution.	Intra	quality	0
no-for-in-array	For-in loops over arrays skips holes and may visit the prototype chain or other enumerable properties.	Intra	quality	3
use-isnan	Direct comparison (x === NaN) is always false in JavaScript, which causes the intended conditional check to fail.	Intra	quality	3
no-case-declarations	Variables in a case scope span the entire switch block, not just the case, risking hoisting and logic errors.	Intra	quality	50
default-case-last	Default clause should be the last clause to avoid fall-through.	Intra	quality	0
no-confusing-non-null-assert	Confusing combinations of non-null assertion and equal test like "a! == b", which looks very similar to not equal "a !== b".	Intra	quality	0
insecure-http-request	Using the HTTP protocol to send requests transmits data in plain text, making it susceptible to leakage and tampering.	Inter	security	33
log-sensitive-info	Logging sensitive information may cause privacy disclosure	Inter	security	12
hardcoded-password	Embedding sensitive data such as passwords or API keys directly in source code to avoid information disclosure.	Intra	security	1
weak-pseudorandom-number	Disallow the use of insecure random number generation functions such as `Math.random()` to avoid being predicated.	Intra	security	226
unsafe-mac-hash	Disallow the use of insecure hash algorithms in MAC (Message Authentication Code) algorithms to avoid being attacked.	Intra	security	5
field-init-with-sensitive-info	Initializing sensitive information in code may lead to information leakage.	Intra	security	77
comment-with-sensitive-info	Finding comments with sensitive information	Intra	security	122
no-http-no-scheme	HTTP URLs without a scheme in setRequestMethod calls may cause protocol downgrade or SSRF.	Intra	security	67
unsafe-references-in-deps	Using insecure references poses security risks.	Intra	security	91

4.2.2 Detection Results and False Positive Analysis. The detection results are summarized in Table 2. Totally, HapCheck identifies 4,278 defects distributed across 177 projects including 1,350 performance issues, 634 security issues and 2,294 quality issues. The project with the highest number of defects contains 496 issues, including 492 instances of `ban-types`. Note that the quality and performance issues are usually code smells and unlikely cause seriously issues. Security issue may require further attentions from developers.

We further evaluate the potential false positives of detection results. The intra-statement rules are unlikely introduce false positive as such rules usually check simple syntactic patterns. For instance, the `no-key-generator-in-foreach` rule check the existence of third parameter, which is deterministic and will not introduce any false positive. For defects requiring inter-statement analysis, the detection result may introduce false positives because of the potentially imprecise data/control flow construction. Hence, we implemented a dual-reviewer cross-verification of the results detected by inter-statement security rules. Specifically, two researchers independently examined the detection results, followed by a consensus meeting to resolve discrepancies. Among the 88 defects detected by inter-statement rules, 10 of them are determined as false positives. Among these 10 false positives, five were reported by the `insecure-http-request` rule: four of them stemmed from the presence of security check measures in the code that mitigated

the risks associated with HTTP requests, while one was classified as a false positive because the request target address was a local loopback interface (resulting in low actual risk). The five remaining false positives were triggered by the `log-sensitive-info` rule, all stemming from the DSL rules misclassifying low-risk information as high-risk. This reflects the limitations of hapcheck in identifying protective barriers, as well as its inability to flexibly adjust the definition of sensitive information according to practical scenarios.

To further validate our findings, we selected 23 issues across 20 projects and informed developers about these issues. The selected projects either had updates since July 1, 2025, or contained critical security vulnerabilities (such as unsafe-mac-hash, insecure-http-request, and hardcoded-password). Each issue documented multiple defects of a specific category in the respective project. As of September 30, 2025, we received responses for 9 issues:

- 7 confirmed the presence of reported defects
- 2 was verified as a false positive

The two false-positive cases both involved a project using weak pseudo-random numbers for page animation generation. Although attackers could theoretically predict random values, the impact was limited to predictable animation effects without security implications. This case highlights a key limitation of static analysis - its inability to effectively evaluate usage context and application-specific risk scenarios.

Furthermore, we compare the defect detection capabilities of HapCheck with native HomeCheck. For the rules executed by HomeCheck's engine, HapCheck and HomeCheck produce identical detection results. However, since HomeCheck currently does not support flow-sensitive analysis, it fails to detect the issues identified by the CPG engine.

4.2.3 Result Analysis. **Performance**: Among performance-related issues, 96.3% of defects are attributed to the omission of the third parameter in the `ForEach` method. This defect is distributed across 115 projects, exhibiting a wide distribution and occurring in applications of various types. This widespread prevalence reflects that the awareness of performance optimization guidelines among current ArkTS developers still needs to be strengthened.

Quality: In terms of code quality, the `ban-types` rule is violated in 96 projects, which may be attributed to the fact that relying solely on capitalization to distinguish types can easily lead to misuse. Additionally, the `no-non-null-assertion` rule violation present in 68 projects. This prevalence may be attributed to the fact that the `!` non-null assertion operator simplifies code by circumventing extensive type checks. Consequently, it has become a favored practice among developers seeking to enhance coding efficiency, despite potentially compromising type safety and long-term maintainability.

Security: From a security perspective, the vulnerabilities we detect span multiple CWEs, such as CWE-798 (Use of Hard-coded Credentials) and CWE-319 (Cleartext Transmission of Sensitive Information), which may cause serious security issues. The `weak-pseudo-random-number` rule identifies the highest number of violations, amounting to 226 instances. This prevalence may be attributed to the lack of awareness of developers about the inherent security risks associated with the widespread use of the `Math.random()` function, which is not cryptographically secure and the random value can be predicted by malicious users. Furthermore, when examining the inter-statement `insecure-http-request` rule, the CPG engine detects 33 defects involving multiple lines of code. This demonstrates the CPG engine's robust capability in detecting complex interprocedural vulnerabilities.

4.3 RQ2: Execution Time

This section conducts a performance evaluation using two core execution engines (HomeCheck and CPG). Additionally, we experimented with replacing Neo4J with in-memory storage in the CPG engine (CPG-Mem) to achieve faster execution speed. We measure HapCheck's performance on projects with varying sizes (1-999 LOC, 1k-99k LOC, 10k-100k LOC and >100k LOC), representing common application scales. We measure HomeCheck and CPG engines by executing 1, 5, and 10 rules per run.

The results (Table 3) reveal a clear performance distinction of three different engines. HomeCheck with DSL, CPG-Mem, and CPG-Neo4J—exhibit markedly different performance profiles across varying project sizes. HomeCheck demonstrates a consistent and relatively low execution time for rule execution, with only a moderate increase as the number of rules scales from one to ten. This comes at the cost of a mandatory preprocessing phase, the duration of which grows significantly with the project's LOC. In contrast, the CPG-Mem engine features a remarkably low rule execution time, often an order of magnitude faster than HomeCheck, and maintains a relatively stable and low CPG build time, owing to our targeted optimizations for the CPG engine. CPG-Neo4J engine, while also achieving very fast query times for rules, is characterized by an exceptionally high and often prohibitive CPG build time. The huge build time is mainly caused by the data transition with Neo4J dataset, which becomes the dominant factor in its overall performance, especially for larger projects.

Both HomeCheck and CPG-Mem are well-suited for integration within an Integrated Development Environment (IDE) to provide real-time feedback to developers. HomeCheck can offer rapid, incremental rule checks after its initial preprocessing, while CPG-Mem's consistently low build and query times make it excellent for providing near-instantaneous analysis during code edits. In contrast, the CPG-Neo4j engine is not ideal for real-time interaction but better for a Continuous Integration/Continuous Deployment (CI/CD) pipeline. In this scenario, the substantial cost of building the graph is amortized over a single, scheduled run, and its fast query capabilities can then efficiently execute a large battery of rules against the persisted codebase to ensure quality and security before deployment.

4.4 RQ3: Simplify the Writing of Detection Rules

To investigate whether DSL can simplify the writing of rules, we conducted experiments based on the HomeCheck engine. First, we compared the lines of code required to implement identical functionalities using both DSL and native HomeCheck approaches. Subjectively, we administered a questionnaire survey to 13 professionals with defect detection expertise, presenting them with equivalent code samples written in both DSL and native HomeCheck syntax, and collected their readability assessments.

Table 3: Execution Time Comparison (in milliseconds) for Different Projects and Rules Using HomeCheck, CPG-Mem, and CPG-Neo4J Engines

LOC Range	HomeCheck+DSL				CPG-Mem				CPG-Neo4J
	Pre*	1 Rule	5 Rules	10 Rules	Build CPG	1 Rule	5 Rules	10 Rules	Build CPG
[1, 999]	230	30	40	50	804	8	48	100	41,487
[1,000, 9,999]	680	130	160	200	1,194	12	76	153	67,514
[10,000, 99,999]	3,650	700	860	1,030	2,538	28	210	408	> 1.7e5
[100,000, ∞)	12,860	3,500	4,950	7,060	6,365	71	647	1,048	> 5e5

* The preprocessing of HomeCheck, including collecting and filtering files to be analyzed, constructing scenes and establishing mappings between rules and files.

4.4.1 Code Volume Comparison. To assess the syntactic conciseness and expressive efficiency of the DSL in rule creation, we compared the amount of code required to implement the same rules using the original approach and the DSL. Specifically, we compare 11 rules with the native implementation of HomeCheck. The results are shown in Table 4.

Table 4: Lines of code of DSL and Homecheck rules

Rule Name	DSL	HC	Comparison
ban-types	67	387	-320
no-cond-assign	90	215	-125
no-for-in-array	18	75	-57
use-isnan	120	232	-112
no-key-generator-in-foreach	39	109	-70
console-in-high-freq-func	79	139	-60
no-useless-catch	48	100	-52
no-non-null-assertion	15	75	-60
default-case-last	21	104	-83
no-case-declarations	31	95	-64
no-control-regex	45	141	-96
no-confusing-non-null-assert	68	169	-101
Average	53.42	153.42	-100.00

The results show that, on average, each rule saved approximately 100 lines of code after being rewritten using the DSL, representing a reduction of approximately 65%. This reduction can be attributed to two key design aspects of the DSL:

(1) The DSL provides high-level abstractions that encapsulate common conditional and matching logic in rule definitions, thereby reducing syntactic verbosity and improving expressiveness.
(2) The DSL abstracts away the low-level, boilerplate node traversal mechanisms inherent in the native implementation approach. Although this abstraction entails a trade-off in terms of customization flexibility, it substantially reduces the implementation burden and code complexity for common rule patterns.

4.4.2 Readability of DSL. A total of 16 participants were recruited for this study, categorized based on their familiarity with TypeScript (TS). Among them, 3 participants reported being highly familiar with TS, 8 were moderately familiar, and 5 were considered to have low familiarity. In the questionnaire, we provided two defect rules (`no-for-in-array` and `default-case-last`), with each implemented in both native Homecheck syntax and DSL, and asked participants to review the code of these rule implementations and rate them. The readability assessment dimensions were defined as follows:

- **Expressiveness**: The capability of rules to comprehensively cover various defect scenarios.
- **Clarity**: The ease of understanding the rule descriptions.
- **Conciseness**: The compactness of rules and the absence of redundant content.

The rating scale ranges from 1 (lowest) to 5 (highest). The experimental results are presented in Table 5, where each data point represents the average of evaluations from two rules.

Table 5: Readability of DSL and Homecheck Rules under Different Levels of TypeScript Familiarity

Metric	High		Moderate		Low	
	DSL	HC	DSL	HC	DSL	HC
Expressiveness	4.50	3.83	3.75	3.31	4.00	3.50
Clarity	4.83	3.17	3.50	3.19	4.00	3.50
Conciseness	5.00	2.17	3.69	2.56	4.00	3.60

First, comparing the DSL with Homecheck, it is evident that the DSL consistently outperforms Homecheck on all levels of familiarity and all dimensions of evaluation. The maximum performance gap reaches 5.00 versus 2.17, highlighting the substantial advantage of DSL in terms of readability.

Second, examining the impact of familiarity on readability, we observe that Homecheck's scores exhibit relatively little variation across different familiarity levels. This suggests that even developers highly proficient in TypeScript (TS) may struggle to quickly comprehend Homecheck rules. In contrast, developers who are "very familiar" with TS achieve significantly higher scores than those who are "moderately" or "lowly" familiar, indicating that a foundational proficiency in TS enables developers to better understand DSL rules.

4.5 RQ4: Ablation Study for ArkUI Lifecycle

We conducted a ablation study to assess the impact of supporting ArkTS's lifecycle features. Specifically, we disabled the lifecycle support module in our framework and evaluated its ability to detect the `aboutToAppear-object-never-dispose` rule. This rule refers to a scenario in which objects instantiated within the aboutToAppear lifecycle method are never properly disposed of, potentially leading to memory leaks and performance degradation. The results revealed that without the lifecycle feature support, the framework failed to identify this problem, as it relies on understanding ArkTS's lifecycle hooks to track object creation and disposal. This finding underscores the essential role of lifecycle support in enabling effective static detection for ArkTS-specific performance pitfalls.

5 RELATED WORKS

Generic Static Analysis Engines and DSLs: The design of domain-specific languages for static analysis has been extensively explored. CodeQL, a prominent industrial-strength tool, employs a variant of Datalog (a declarative logic programming language) to express complex program queries [10, 28, 40]. While powerful for vulnerability discovery in diverse languages, its SQL-like syntax imposes a significant cognitive barrier for developers unfamiliar with logic programming. Crucially, CodeQL struggles with seamless integration of complex external computations—invoking third-party libraries within predicates is cumbersome or often requires engine-level extensions, limiting expressiveness for domain-specific checks like those needed in OpenHarmony.

Similarly, SonarQube [3] relies on ast traversal APIs (typically in Java and TypeScript) for rule creation, which can be verbose and low-level. However, these often operate at a syntactic or limited semantic level, requiring rule authors to manually handle complex control/data flows and lacking built-in abstractions for domain-specific patterns (e.g., UI component lifecycles, reactive state management) [38]. Frameworks like Infer [9] (based on separation logic) and Checkmarx [10] offer sophisticated analysis but lack dedicated, accessible DSLs tailored for application developers, focusing instead on security analysts. Semgrep [1, 12, 17] provides a simpler pattern-matching DSL but lacks the expressiveness for complex, inter-procedural, or stateful analyses required for deep ArkTS semantics (e.g., tracking state changes across asynchronous boundaries or decorator-induced behaviors), and suffers from false positives due to its shallow representation [34, 35, 39].

Joern[29] is an open-source security analysis platform based on the CPG, which integrates multiple program representations such as AST, CFG, DFG, and CG into a unified intermediate form, enabling inter-procedural and context-sensitive vulnerability detection. However, its query language is relatively low-level and has a steep learning curve, making it less accessible to rule developers. Moreover, Joern lacks high-level abstractions for domain-specific language features (e.g., reactive programming models or lifecycle semantics in ArkTS), failing to support ArkTS.

Static analyzer for OpenHarmony: ArkAnalyzer is a sophisticated static analysis framework meticulously engineered for the OpenHarmony ecosystem. It provides a comprehensive suite of foundational static analysis capabilities, including control flow graph construction and call graph generation, addressing the critical gap in existing tools that lack support for ArkTS-specific syntactic and structural features. This enables accurate parsing and semantic understanding of ArkTS code, which is essential for static analysis.

HomeCheck is a dedicated static analysis tool designed for OpenHarmony software. It performs in-depth static code analysis on application projects, assessing code quality with respect to security, performance, and maintainability. Furthermore, HomeCheck demonstrates high precision in identifying potential issues and localization. As a result, it facilitates efficient debugging and code improvement, contributing significantly to the robustness and reliability of OpenHarmony-based applications.

Embedded DSLs for Program Analysis: Embedded DSLs (eDSLs) leverage the syntax and tooling of a host language to lower adoption barriers. For instance, Roslyn analyzers for .NET [11] allow writing diagnostic rules in C#, benefiting from IDE support. RascalMPL [12] is a meta-programming language built atop Java for language analysis and transformation. While demonstrating the productivity benefits of eDSLs, these approaches are either tied to specific platforms (.NET, JVM) or lack deep integration with the unique semantics of modern UI frameworks and asynchronous runtimes like ArkTS/OpenHarmony.

ESLint, a TypeScript-based analysis leverages the TypeScript Compiler API [30, 31] for linting. However, these often operate at a syntactic or limited semantic level, requiring rule authors to manually handle complex control/data flows and lacking built-in abstractions for domain-specific patterns (e.g., UI component lifecycles, reactive state management).

In contrast, our work fundamentally differs by embedding its DSL directly within TypeScript, eliminating the paradigm shift, and enabling unrestricted use of TS libraries and complex logic within rules, advances this paradigm by being the first eDSL deeply specialized for OpenHarmony's ArkTS, providing native constructs for decorators, event handling, and concurrency models out-of-the-box.

6 CONCLUSION

In this paper, we present HapCheck, a novel framework for detecting defects in ArkTS. It features: (1) An embedded DSL for concise, readable rule specification; (2) Compatibility with existing tools like HomeCheck. (3) A CPG-based engine optimized for ArkTS, including lifecycle-aware control flow modeling for precise static analysis; Our evaluation shows HapCheck effectively detects defects, uncovering 4,000+ issues in real-world projects. Future work will expand its capabilities for large-scale rules and ArkTS projects.

Acknowledgments

This work was supported by the National Key R&D Program of China No 2024YFB4506300 and the Fundamental Research Funds for the Central Universities.

[9] https://fbinfer.com/
[10] https://checkmarx.com/
[11] https://github.com/dotnet/roslyn
[12] https://www.rascal-mpl.org/

References

[1] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. 2024. Semgrep*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 614–623.

[2] Eric Bodden. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 5–14.

[3] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[4] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*. 1456–1468.

[5] Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. ArkAnalyzer: The Static Analysis Framework for OpenHarmony. *arXiv preprint arXiv:2501.05798* (2025).

[6] Roland Croft, Dominic Newlands, Ziyu Chen, and M Ali Babar. 2021. An empirical study of rule-based and learning-based approaches for static application security testing. In *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*. 1–12.

[7] Ahmed M. Darwish and Anil K. Jain. 1988. A rule based approach for visual pattern inspection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10, 1 (1988), 56–68.

[8] Xingjing Deng, Zhengyao Liu, Xitong Zhong, Shuo Hong, Yixin Yang, Xiang Gao, Xuhui Yan, and Hailong Sun. 2025. Code Property Graph Meets Typestate: A Scalable Framework to Behavioral Bug Detection. In *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, to appear.

[9] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–34.

[10] Matías F Gobbi and Johannes Kinder. 2023. Poster: Using codeql to detect malware in npm. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3519–3521.

[11] Yongjian Hu and Iulian Neamtiu. 2018. Static detection of event-based races in android apps. *ACM SIGPLAN Notices* 53, 2 (2018), 257–270.

[12] Lukas Kree, René Helmke, and Eugen Winter. 2024. Using semgrep oss to find owasp top 10 weaknesses in php applications: A case study. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 64–83.

[13] Fangming Li, Jian Feng, Huaguang Zhang, Jinhai Liu, Senxiang Lu, and Dazhong Ma. 2018. Quick reconstruction of arbitrary pipeline defect profiles from MFL measurements employing modified harmony search algorithm. *IEEE Transactions on Instrumentation and Measurement* 67, 9 (2018), 2200–2213.

[14] Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F. Bissyandé, Jacques Klein, John Grundy, Tao Xie, Haibo Chen, and Huaimin Wang. 2023. Software Engineering for OpenHarmony: A Research Roadmap. arXiv:2311.01311 [cs.SE] https://arxiv.org/abs/2311.01311

[15] Farong Liu, Mingyi Zhou, Yakun Zhang, Ting Su, Bo Sun, Jacques Klein, Xiang Gao, and Li Li. 2025. HapTest: The Dynamic Analysis Framework for OpenHarmony. (2025).

[16] Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. 2024. LLM-Based Java Concurrent Program to ArkTS Converter. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2403–2406. doi:10.1145/3691620.3695362

[17] Charles Lohest and Axel Legay. 2024. Improving security analysis rule set by relationship identification.. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 297–300.

[18] Yunlong Ma, Wentong Tian, Xiang Gao, Hailong Sun, and Li Li. 2024. API Misuse Detection via Probabilistic Graphical Model. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 88–99. doi:10.1145/3650212.3652112

[19] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.

[20] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. 2006. Automatic generation of detection algorithms for design defects. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 297–300.

[21] OpenHarmony. 2025. status. Online. Available: https://www.harmony-developers.com/p/harmonyos-next-applications-has-reached.

[22] Fangcheng Qiu, Zhongxin Liu, Xing Hu, Xin Xia, Gang Chen, and Xinyu Wang. 2024. Vulnerability detection via multiple-graph-based code representation. *IEEE Transactions on Software Engineering* (2024).

[23] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[24] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using domain-specific languages for analytic graph databases. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1257–1268.

[25] Zhidong Shen and Si Chen. 2020. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks* 2020, 1 (2020), 8858010.

[26] Rahmatian Jayanty Sholichah, Mahmud Imrona, and Andry Alamsyah. 2020. Performance Analysis of Neo4j and MySQL Databases using Public Policies Decision Making Data. In *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*. IEEE. doi:10.1109/icitacee50144.2020.9239206

[27] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 119–130. doi:10.1145/3468264.3468620

[28] Tamás Szabó. 2023. Incrementalizing production codeql analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1716–1726.

[29] T. J. Team. 2024. Joern. Online. Available: https://joern.io.

[30] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. 2017. Why and how JavaScript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 578–589.

[31] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. 2018. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering* 46, 8 (2018), 863–891.

[32] Octavian Udrea, Cristian Lumezanu, and Jeffrey S. Foster. 2008. Rule-based static analysis of network protocol implementations. *Information and Computation* 206, 2 (2008), 130–157. doi:10.1016/j.ic.2007.05.007 Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA '06).

[33] Milica Vuković, Vladimir Vujović, Zorana Štaka, and Snježana Milinković. 2023. Domain-Specific Language for Modeling Fluent API. In *2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE, 1–6.

[34] 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* 145 (2024), 103994.

[35] Xin-Cheng Wen, Cuiyun Gao, Shuzheng Gao, Yang Xiao, and Michael R Lyu. 2024. Scale: Constructing structured natural language comment trees for software vulnerability detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 235–247.

[36] Peng Wu, Liangze Yin, Xiang Du, Liyuan Jia, and Wei Dong. 2020. Graph-based vulnerability detection via extracting features from sliced code. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 38–45.

[37] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*. IEEE, 590–604.

[38] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM. doi:10.1145/2420950.2421003

[39] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Hailong Sun. 2024. Reducing false positives of static bug detectors through code representation learning. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 681–692.

[40] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495.

[41] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 457–467.