VulPA: Detecting Semantically Recurring Vulnerabilities with Multi-object Typestate Analysis

LIQING CAO, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
HAOFENG LI*, Institute of Computing Technology, CAS, China
CHENGHANG SHI, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
JIE LU, Institute of Computing Technology, CAS, China
HAINING MENG, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
LIAN LI*, Institute of Computing Technology, CAS, China, University of Chinese Academy of Sciences, China, and Zhongguancun Laboratory, China
JINGLING XUE, University of New South Wales, Australia

Detecting semantically recurring vulnerabilities with similar root causes remains a challenge due to the complex interactions between multiple variables. This paper introduces VulPA, a novel approach for precisely identifying such vulnerabilities through complex inter-procedural data and control flows across multiple objects. VulPA tackles this challenge in two steps: 1) Defining root causes with a Vulnerability Pattern Description Language (VPDL) that specifies variable relations and bug-triggering operations, and 2) Detecting these patterns using an inter-procedural multi-object analysis that tracks dataflows and variable interactions. Built on the Heros IFDS framework, VulPA was evaluated on 26 Java applications using rules from 34 CVEs. It identified 90 new vulnerabilities (23.7% false positive rate), outperforming existing tools (ReDeBug, VUDDY, SourcererCC, PHunter, PPT4J, FlowDroid, and IDE^{al}), which collectively found only 13. VulPA effectively uncovers complex vulnerabilities missed by state-of-the-art tools.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Recurring vulnerability, Temporal logic, IFDS

ACM Reference Format:
Liqing Cao, Haofeng Li, Chenghang Shi, Jie Lu, Haining Meng, Lian Li, and Jingling Xue. 2025. VulPA: Detecting Semantically Recurring Vulnerabilities with Multi-object Typestate Analysis. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE108 (July 2025), 24 pages. https://doi.org/10.1145/3729378

*Corresponding author

Authors' Contact Information: Liqing Cao, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, caoliqing19s@ict.ac.cn; Haofeng Li, SKLP, Institute of Computing Technology, CAS, Beijing, China, lihaofeng@ict.ac.cn; Chenghang Shi, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, shichenghang21s@ict.ac.cn; Jie Lu, SKLP, Institute of Computing Technology, CAS, Beijing, China, lujie@ict.ac.cn; Haining Meng, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, menghaining@ict.ac.cn; Lian Li, SKLP, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China and Zhongguancun Laboratory, Beijing, China, lianli@ict.ac.cn; Jingling Xue, University of New South Wales, Sydney, Australia, j.xue@unsw.edu.au.

ACM 2994-970X/2025/7-ARTFSE108
https://doi.org/10.1145/3729378

1 Introduction

Recurring vulnerabilities are prevalent in real-world systems [Kim and Lee 2018; Liu et al. 2020]. A recent Google report revealed that 6 out of 24 0-day vulnerabilities in 2020 were variants of existing ones [Zero 2021]. These vulnerabilities can recur either syntactically, due to code reuse, or semantically, from common misconceptions in implementing similar code logic [Pham et al. 2010], such as misuse of common APIs. While syntactic recurring vulnerabilities are often detectable through code similarity analysis [Jang et al. 2012; Kim et al. 2017], detecting semantically recurring vulnerabilities is more challenging and requires precise root cause analysis. Despite recent advancements [Jang et al. 2012; Kang et al. 2022; Kim et al. 2017; Shi et al. 2024; Xiao et al. 2020], many recurring vulnerabilities remain undetected.

Problem Statement. This paper tackles the challenge of identifying recurring vulnerabilities with similar root causes and detecting them precisely while maintaining a low false positive rate.

To illustrate the challenge of detecting semantically recurring vulnerabilities, consider CVE-2022-41936 from Xwiki, shown in Figure 1. This vulnerability allows unauthorized users to access sensitive information (such as hidden pages) through the public REST API `getModifications()`. The vulnerability stems from the lack of an access control check before executing the sensitive `createHistorySummary` operation on `wiki`, `spaces`, and `page`. The remediation (lines 3-5 and 7) shows how these variables are linked to `docRef` through the `getPageId` and `resolve` methods (lines 3-4). The `docRef` variable is then validated using the `hasAccess` method (line 5) to ensure the necessary permissions before performing the sensitive operation (line 6).

To detect CVE-2022-41936 and similar vulnerabilities, it is essential to capture the relationships between these variables and track their data flows. The vulnerability occurs if `wiki` can reach the `createHistorySummary` operation without the `docRef` validation, which is derived from `wiki`, `spaces`, and `page` via the `getPageId` and `resolve` methods. Failure to track these relations precisely may lead to missing the vulnerability or generating numerous false positives. The core challenge is: *how can we accurately capture these variable relationships to detect the vulnerability?*

```
1  public History getModifications(String wiki) {
2      . . .
3  + String pageId = Utils.getPageId(wiki, spaces, page);
4  + DocRef docRef = resolver.resolve(pageId);
5  + if (authManager.hasAccess(docRef)) {
6      Factory.createHistorySummary(wiki, spaces, page);
7  + }
```

Fig. 1. A simplified code snippet for CVE-2022-41936.

Prior Work. Existing approaches struggle to detect semantically recurring vulnerabilities like CVE-2022-41936 (Figure 1). Signature-based methods [Jang et al. 2012; Kim et al. 2017; Xiao et al. 2020] rely on syntactic or pattern-based similarities but fail to track data flows, missing vulnerabilities that are structurally different yet semantically similar. Static analysis tools, including taint analysis [Arzt et al. 2014; He et al. 2019; Li et al. 2015; Wang et al. 2023] and typestate analysis [Fink et al. 2008; Hallem et al. 2002; Li et al. 2022; Späth et al. 2017], primarily focus on individual objects, overlooking interactions between multiple variables—such as calling `hasAccess(docRef)` before invoking `createHistorySummary(wiki,spaces,page)`. Although multi-object typestate analysis has been proposed in [Naeem and Lhotak 2008], it relies on a complex automaton as input and is not publicly available. As a result, two vulnerabilities similar to CVE-2022-41936 remained undetected for 630 days until our discovery.

In practice, such vulnerabilities often involve complex inter-procedural control and data flows across variables, complicating detection. This raises a critical question: *how can we effectively identify the nuanced characteristics of semantically recurring vulnerability patterns?*

This Work. We address this challenge with a novel two-step approach:

- **Specifying Vulnerability Patterns.** We have developed a domain-specific language, *Vulnerability Pattern Description Language* (VPDL), to describe vulnerability root causes. VPDL defines a vulnerable trace, outlining variable relations and key bug-triggering operations. With a straightforward syntax, VPDL uses variables and code-like statements, making it accessible for application developers without requiring program analysis expertise.
- **Detecting VPDL Patterns.** A vulnerability is detected when a program path matches a VPDL pattern. This is achieved through an IFDS (Inter-procedural, Finite, Distributive, Subset)-based multi-object typestate analysis [Naeem and Lhotak 2008], which tracks dataflows and interactions across multiple variables. The VPDL specification is first translated into linear temporal logic (LTL) formulas with free VPDL variables. These formulas are then converted into Büchi automata and solved using an IFDS-based algorithm, which processes automaton state transitions alongside variable bindings. A vulnerability is flagged when a program path reaches a vulnerable state with valid bindings.

We have developed VulPA, a recurring vulnerability detection tool, and evaluated it on 34 CVEs (covering 63 vulnerabilities) across 26 Java applications. VulPA has successfully detected all 63 known vulnerabilities and identified 90 new ones, achieving a 23.7% false positive rate. In contrast, existing static analysis tools (ReDeBug, VUDDY, SourcererCC, PHunter, PPT4J, FlowDroid, and IDE^{al}) collectively detected only 13 new vulnerabilities.

Contributions. This paper makes the following key contributions:

- We design VPDL, a domain-specific language for describing semantically recurring vulnerability patterns with complex control and data dependencies across multiple objects.
- We develop a novel IFDS-based algorithm that simultaneously handles automaton state transitions and multi-variable data dependencies, enabling precise detection of VPDL-specified patterns.
- We implement VulPA and evaluate it on 63 real-world vulnerabilities across 26 Java applications, identifying 90 new vulnerabilities, with majority of them undetectable by existing tools.

The rest of the paper is structured as follows. Section 2 presents an overview of VulPA. Section 3 formally describes the VPDL language, and Section 4 elaborates on our IFDS-based vulnerability detection algorithm. Section 5 evaluates the effectiveness of our approach on real-world vulnerability cases. Section 6 discusses related work, and Section 7 concludes the paper.

2 The VulPA Approach

Figure 2 shows the high-level architecture of VulPA, which detects recurring vulnerabilities in two steps. First, the root cause of an existing vulnerability is specified as a VPDL pattern. Second, new recurring vulnerabilities are reported by matching program paths to the VPDL pattern in the analyzed programs.

2.1 Specifying Vulnerability Patterns in VPDL

We design the *Vulnerability Pattern Description Language* (VPDL) to precisely express vulnerability root causes. Inspired by languages like SmPL [Padioleau et al. 2007] and Metal [Hallem et al. 2002], VPDL features a Java-like syntax. Figure 3 shows the VPDL specification for the CVE-2022-41936

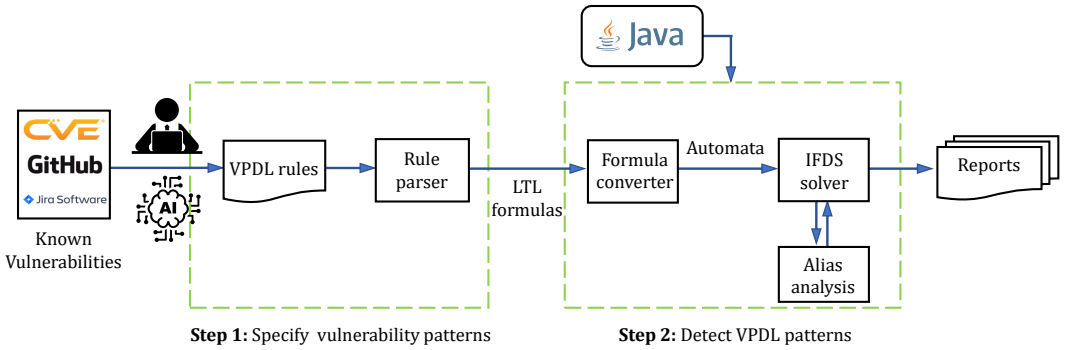

Fig. 2. The high level architecture of VulPA.

```
1  @CVE-2022-41936@
2  Variable wiki, spaces, page, pageId, docRef;
3  @@
4  not {
5    pageId = Utils.getPageId(wiki, spaces, page);
6    docRef = Resolver.resolve(pageId);
7    AuthManager.hasAccess(docRef);
8  }
9  Factory.createHistorySummary(wiki, spaces, page);
```

Fig. 3. VPDL for CVE-2022-41936.

```
1  public History getPageHistory(String wikiName,
     String spaceName, String pageName) {
2    List<String> spaces = parseSpaceSegments(spaceName);
3    . . .
4    for (Object object : queryResult) {
5      . . .
6      HistorySummary historySummary = Factory
         .createHistorySummary(wikiName, spaces, pageName);
7      . . .
```

Fig. 4. Simplified code snippet containing a new vulnerability in PageHistoryResourceImpl.java.

example in Figure 1. The description begins with a header enclosed by "@" symbols, annotating its name. The body is divided into two sections by the "@@" delimiter. The first section declares free variables to be mapped to program variables during detection. The second section describes a vulnerability-triggering trace, including statements that capture variable relations (lines 5–6) and bug-triggering operations (line 9). The "`not{...}`" clause excludes the enclosed statement sequence from the trace. Thus, Figure 3 precisely captures the root cause of CVE-2022-41936: the invocation of `createHistorySummary()` (line 9) occurs without the required access control checks (lines 5–7).

VPDL specifications are parsed by the VPDL parser and translated into LTL formulas with free variables, which are checked in the next step. Manually deriving VPDL specifications involves understanding vulnerability root causes and patches, akin to defining sources, sinks, and sanitizers in classic taint analysis. In our experience, writing a VPDL specification for an existing vulnerability takes less than 10 minutes, assuming familiarity with vulnerability patterns.

This paper does not explore automatic generation of VPDL specifications. Automatically learning VPDL specifications remains a promising direction for future work.

2.2 Detecting VPDL Patterns

We report vulnerabilities by identifying program paths that match a VPDL pattern, specifically paths satisfying the LTL formulas derived from the pattern. Figure 4 shows a new vulnerability in Xwiki, where the path from line 1 to line 6 matches the VPDL pattern in Figure 3. This path propagates `wikiName` to `createHistorySummary()` without the required access control checks

(matching the `not` clause in Figure 3). Here, the VPDL variables *wiki*, *spaces*, and *page* map to program variables `wikiName`, `spaces`, and `pageName`, respectively.

Revisiting the code snippet in Figure 1, the patch fixes CVE-2022-41936, as it no longer satisfies the `not` clause in Figure 3 and thus does not match the VPDL pattern. However, changing `wiki` in line 3 to `wiki1` restores satisfaction of the `not` clause, as the VPDL variable *wiki* cannot bind to two distinct, non-aliasing variables. This will result in a vulnerability being reported.

As noted in previous work [Beyer et al. 2018; Schmidt 1998], model checking of LTL formulas can be viewed as a dataflow analysis problem. In VulPA, LTL formulas are solved using a dataflow analysis implemented in the IFDS framework [Heros 2023; Reps et al. 1995]. First, the LTL formulas of a VPDL pattern are converted into Büchi automata [Duret-Lutz et al. 2022b; Vardi and Wolper 1986]. An IFDS-based algorithm, proposed in this paper, then computes automaton states while mapping VPDL variables to program variables at each program point. The mapping is updated during dataflow analysis when assigning one mapped program variable to another, with aliases managed by the alias analysis module. A vulnerability is reported if an accepted automaton state is reached with valid VPDL variable mappings.

3 The VPDL Language

We introduce the syntax and semantics of VPDL, followed by an illustrative example.

3.1 Syntax

The simplified syntax of VPDL is formally defined as follows:

$$
\begin{array}{lll}
P & ::= @\ name\ @\ \overline{V}\ @@\ \overline{S} & \\
V & ::= T\ x & \\
T & ::= t \mid \texttt{Variable} \mid \texttt{Method} \mid \ldots & (1) \\
S & ::= p(\overline{x}) \mid \{\overline{S}\}\ \texttt{and}\ \{\overline{S}\} \mid \{\overline{S}\}\ \texttt{or}\ \{\overline{S}\} \mid \texttt{not}\ \{\overline{S}\} & \\
\text{Identifiers} & name, t, x, p &
\end{array}
$$

A vulnerability pattern description P starts with the "`@` *name* `@`" construct, denoting its name, followed by a set of variables $\overline{V}$ and a sequence of statements $\overline{S}$. A variable x is qualified by type T, which can be an identifier t representing a Java type (e.g., `String`), indicating that x can be any variable of type t in the Java programs under analysis. Alternatively, it can be a keyword (`Variable`, `Method`, ...) referring to a specific program element. For example, the declaration "`Variable` *wiki*" (line 2 in Figure 3) indicates that *wiki* can be any variable in the target Java program.

A VPDL statement S can be either a predicate $p(\overline{x})$ or a logical operation (`and`, `or`, or `not`) on sequences of VPDL statements $\overline{S}$. Specifically, the predicate $p(\overline{x})$ defines the code patterns to be matched, with $\overline{x}$ representing a set of declared VPDL variables.

3.2 Semantics

A VPDL specification defines a sequence of statements $\overline{S}$ to be matched in order along a program path. Each atomic proposition $p(\overline{x}) \in \overline{S}$ matches a program statement $p(\overline{v})$, where the VPDL variable x is bound to the program variable v. A VPDL variable cannot be bound to distinct non-aliasing variables, and paths with invalid bindings are discarded. The logical operators "$\overline{S}$ `or` $\overline{S}$", "$\overline{S}$ `and` $\overline{S}$", and "`not` $\overline{S}$" create complex path constraints: "$\overline{S}$ `or` $\overline{S}$" requires at least one of two sequences to match, "$\overline{S}$ `and` $\overline{S}$" requires both to match, and "`not` $\overline{S}$" ensures the sequence does not match.

3.2.1 Linear Temporal Logic. We formally define the semantics of VPDL in linear temporal logic (LTL), where time is modeled as a sequence of states and connectives refer to the future [Huth and Ryan 2004]. To enhance LTL's expressive power, we adopt existing approaches [Bustan et al. 2005;

$$
\begin{aligned}
\mathrm{T}\,(\overline{V}\ @@\ \overline{S}) &= start \wedge X\ F\ C[\![\overline{S}]\!]\ true \\
C[\![\overline{S}\ S]\!]\ a &= C[\![\overline{S}]\!](C[\![S]\!]\ a) \\
C[\![p(\overline{x})]\!]\ a &= p(\overline{x})\ \wedge\ X\ F\ a \\
C[\![\{\overline{S}_1\}\ \texttt{or}\ \{\overline{S}_2\}]\!]\ a &= (C[\![\overline{S}_1]\!]\ a) \vee (C[\![\overline{S}_2]\!]\ a) \\
C[\![\{\overline{S}_1\}\ \texttt{and}\ \{\overline{S}_2\}]\!]\ a &= (C[\![\overline{S}_1]\!]\ a) \wedge (C[\![\overline{S}_2]\!]\ a) \\
C[\![\texttt{not}\{\overline{S}\}]\!]\ a &= N[\![\overline{S}]\!]\ a \\
N[\![\overline{S}\ S]\!]\ a &= N[\![\overline{S}]\!]\ (N[\![S]\!]\ a) \\
N[\![p(\overline{x})]\!]\ a &= (\neg p(\overline{x})\ U\ a) \\
N[\![\{\overline{S}_1\}\ \texttt{or}\ \{\overline{S}_2\}]\!]\ a &= (N[\![\overline{S}_1]\!]\ a)\ \wedge\ (N[\![\overline{S}_2]\!]\ a) \\
N[\![\{\overline{S}_1\}\ \texttt{and}\ \{\overline{S}_2\}]\!]\ a &= (N[\![\overline{S}_1]\!]\ a)\ \vee\ (N[\![\overline{S}_2]\!]\ a)
\end{aligned}
$$

Fig. 5. A simplified translation of VPDL to LTL.

De Giacomo et al. 2013; Demri and d'Souza 2007; Wolper 1983] to extend LTL with free variables. In this extension, atomic propositions are denoted as $p(\overline{x})$, with $\overline{x} = \{x_1, \ldots, x_n\}$ representing free variables. This extension allows us to express properties involving different entities as VPDL variables, which is crucial for vulnerability analysis and detection.

The syntax of our extended LTL is given as follows:

$$\phi ::= true \mid false \mid p(\overline{x}) \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (X\phi) \mid (F\phi) \mid (\phi U \phi)$$

Here, $p(\overline{x})$ is a predicate that holds true at matching statements labeled with $\mathsf{p}(\overline{\mathsf{v}})$. To support the semantics of predicates over free variables, we introduce an environment $E = \{x \mapsto \overline{\mathsf{v}}\}$, mapping free variables to an arbitrary set of program variables v. Consequently, all judgments carry an environment. The logical operators include negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). The temporal operators are denoted as X, F, and U. The *next* operator (X) indicates that a proposition must hold in the next state, the *eventually* operator (F) specifies that a proposition must hold at some point in the future, and the *until* operator (U) asserts that a proposition must hold until another proposition is satisfied. These connectives are sufficient to express the complete semantics of LTL, as previously concluded [Huth and Ryan 2004].

3.2.2 VPDL in LTL. Figure 5 shows the translation of a VPDL description into an LTL formula. $T(\overline{V}@@\overline{S})$ is the entry point, where $\overline{V}$ represents the declared VPDL variables and $\overline{S}$ is the sequence of statements in a VPDL description P. The translation yields an LTL formula with free variables, characterizing the program path starting from the program's entry point, $start$.

The translation process involves two sets of functions: 1) The C functions, which translate S sequences excluding those within "$\texttt{not}\ \{\overline{S}\}$", and 2) The N functions, which handle S sequences within "$\texttt{not}\ \{\overline{S}\}$". Both C and N functions take $\overline{S}$ and a as arguments, where $\overline{S}$ is the sequence of statements (defined in Equation (1)) and a is the formula generated to encode $\overline{S}$.

Both sets of translation rules are straightforward. Let us examine the C functions. $C[\![\overline{S}\ S]\!]\ a$ allows a sequence of statements $\overline{S}\ S$ to be handled inductively as $C[\![\overline{S}]\!](C[\![S]\!]\ a)$.

For $C[\![p(\overline{x})]\!]\ a$, the predicate $p(\overline{x})$ represents a statement matched in the target program. Since the sequence of statements reflects temporal ordering, $C[\![p(\overline{x})]\!]a$ indicates that $p(\overline{x})$ is true, and a must eventually hold true in the future. Therefore, it translates to $p(\overline{x}) \wedge X\ F\ a$.

Next, $C[\![\{\overline{S}_1\}\ \texttt{or}\ \{\overline{S}_2\}]\!]\ a$ and $C[\![\{\overline{S}_1\}\ \texttt{and}\ \{\overline{S}_2\}]\!]\ a$ follow standard rules. For $C[\![\texttt{not}\ \{\overline{S}\}]\!]\ a$, we use $N[\![S]\!]\ a$, as no matching sequence $\overline{S}$ exists. Note that multiple nested "$\texttt{not}\ \{\overline{S}\}$" constructs are not supported, as they can lead to confusion and are not practically useful.

The rules corresponding the N functions can be understood similarly.

$$
\begin{aligned}
& T(\overline{V} \texttt{ @@ not}\{p_5, p_6, p_7\}, p_9) && , T(\overline{V}@@\overline{S}) \\
= \ & start \wedge X\ F\ C[\![\texttt{not}\{p_5, p_6, p_7\}, p_9]\!] true && , C[\![\overline{S}\ S]\!]a \\
= \ & start \wedge X\ F\ C[\![\texttt{not}\{p_5, p_6, p_7\}]\!](C[\![p_9]\!] true) && , C[\![p(\overline{x})]\!]a \\
= \ & start \wedge X\ F\ C[\![\texttt{not}\{p_5, p_6, p_7\}]\!](p_9 \wedge XF\ true) && , C[\![\texttt{not}\{\overline{S}\}]\!]a \\
= \ & start \wedge X\ F\ N[\![p_5, p_6, p_7]\!](p_9 \wedge XF\ true) && , N[\![\overline{S}\ S]\!]a \\
= \ & start \wedge X\ F\ N[\![p_5, p_6]\!](N[\![p_7]\!](p_9 \wedge XF\ true)) && , N[\![p(\overline{x})]\!]a \\
= \ & start \wedge X\ F\ N[\![p_5, p_6]\!](\neg\ p_7\ U(p_9 \wedge XF\ true)) && , N[\![p(\overline{x})]\!]\ a \\
= \ & start \wedge X\ F\ N[\![p_5]\!](N[\![p_6]\!](\neg\ p_7\ U(p_9 \wedge XF\ true))) && , N[\![p(\overline{x})]\!]\ a \\
= \ & start \wedge X\ F\ N[\![p_5]\!](\neg\ p_6 U(\neg\ p_7\ U(p_9 \wedge XF\ true))) && , N[\![p(\overline{x})]\!]\ a \\
= \ & start \wedge X\ F\ (\neg p_5\ U(\neg\ p_6 U(\neg\ p_7\ U(p_9 \wedge XF\ true)))) \\
& \text{\# Formula refinement transformation} \\
\Rightarrow \ & start \wedge X\ (\neg p_5\ U(\neg\ p_6 U(\neg\ p_7\ U(p_9 \wedge XF\ true))))
\end{aligned}
$$

Fig. 6. Derivation for Figure 3.

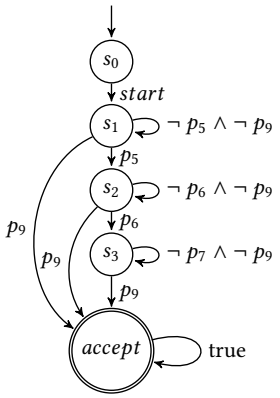

Fig. 7. The Büchi automaton for Figure 3.

After the translation process, we apply a formula refinement transformation that simplifies the LTL formula while preserving its semantics. This targets specific sub-expressions, eliminating the unnecessary F operator: $F(\phi_1\ U\ \phi_2) \Rightarrow \phi_1\ U\ \phi_2$.

3.3 Example

Figure 3 presents the VPDL description for vulnerability CVE-2022-41936, with the CVE ID annotated in Line 1. Line 2 declares the set of variables $\overline{V}$: *wiki*, *spaces*, *page*, *pageId*, and *docRef*. These VPDL variables are annotated with the type `Variable`, meaning they can match any type of variable in the target Java program under analysis.

Lines 4-9 form a sequence of VPDL statements $\overline{S}$, starting with a `not` $\{\overline{S}\}$ clause (lines 5-7) and followed by a single statement (line 9). A predicate $p(\overline{x})$ defines the code pattern to match in the target program and the entities to be bound by VPDL variables. For example, the predicate in line 5 matches any call statement `v1 = Utils.getPageId(...)` in the target program, resulting in $pageId \mapsto \{\texttt{v1}\}$, with *wiki*, *spaces*, and *page* bound to the respective arguments. The predicate in line 6 matches `Resolver.resolve(v2)` only if `v2` is aliased to `v1`, as *pageId* would otherwise be bound to two distinct entities, failing to satisfy the VPDL clause specified.

For simplicity, we refer to all predicates as p_i, where i denotes the line number. For instance, p_5 represents the predicate $pageId = Utils.getPageId(wiki, spaces, page)$ (line 5 in Figure 3).

Figure 6 illustrates the derivation of the VPDL specification in Figure 3 into an LTL formula, based on the rule applications in Figure 5. The subformulas being transformed are highlighted in red. For instance, applying $C[\![p(\overline{x})]\!]a$ results in $C[\![p_9]\!] true = p_9 \wedge XF\ true$. After translation, the formula refinement transformation yields the final LTL formula.

4 The Vulnerability Detection Algorithm

To detect a VPDL pattern, we first convert its LTL formula into a Büchi automaton [Duret-Lutz et al. 2022b; Vardi and Wolper 1986], then use an IFDS-based analysis [Heros 2023; Reps et al. 1995] introduced here to identify program paths satisfying the formula. Both the Büchi automaton and IFDS analysis require extensions to handle free variables in the LTL formula.

4.1 Translate LTL to Automaton

A Büchi automaton [Duret-Lutz et al. 2022b; Vardi and Wolper 1986] is a type of ω-automaton that operates on infinite inputs, extending traditional finite automata. The automaton accepts an infinite input sequence if and only if there exists a run that visits at least one accepting state infinitely

often. An LTL formula ϕ can be converted to a Büchi automaton $A_\phi = (\Sigma, S, s_0, \rho, F)$ [Vardi and Wolper 1986] in the standard manner, using tools like Spot [Duret-Lutz et al. 2022a]. To handle free variables, the automaton is extended by incorporating predicates over free variables in its alphabet:

- $\Sigma = \{p(\overline{x})\}$ is the finite set of *alphabet*, representing predicates declared in VPDL.
- S is the finite set of *states*.
- $s_0 \in S$ is the *initial* state.
- $\rho : S \times \Sigma \to 2^S$ is the transition function. Given $s \in S$ and $p(\overline{x}) \in \Sigma$, $\rho(s, p(\overline{x}))$ is the set of states the automaton moves to when in state s and reading a program statement $\mathsf{p}(\overline{\mathsf{v}})$ matching $p(\overline{x})$.
- $F \subseteq S$ is the set of *accepting* states.

The LTL formula in Figure 6 can be transformed into the Büchi automaton shown in Figure 7, where unsatisfiable conditions are discarded. The automaton transitions from one state to another if the current input program statement being analyzed satisfies the edge's transition condition.

4.2 Multi-object Typestate Analysis

Building on the IFDS algorithm [Reps et al. 1995], we developed a novel algorithm to detect recurring vulnerability patterns specified in VPDL rules. The algorithm tracks automaton state transitions and data dependencies across multiple variables, guided by the Büchi automaton A_ϕ.

At each program point, the analysis computes the set of possible automaton states, with each state linked to an environment that maps a VPDL variable x to a set of program variables $\overline{\mathsf{v}}$. As the program is explored statement by statement, the analysis updates the automaton state and its corresponding environment based on the transition function ρ and the matched code elements. Simultaneously, the analysis tracks data flow propagations, such as for the variable `docRef`, by updating the set of program variables a VPDL variable maps to. This combined tracking of automaton states and environments ensures precise capture of the temporal sequencing and variable relationships specified by the VPDL rule. A program path is considered a *match* for a VPDL pattern if it leads the automaton to an accepting state with non-empty environments.

4.2.1 Dataflow Domain. The fact d is a set of tuples $\{\overline{\langle s,\ e\rangle}\}$, where $s \in S$ is a state in the Büchi automaton A_ϕ, and e is an environment consisting of bindings for the VPDL variables. A VPDL variable x can have either positive bindings, representing the set of program variables to which x is mapped, or negative bindings, indicating the set of program variables from which x is excluded. By convention, we use $e[x]$ for the binding of x in e, and $e[x \mapsto \overline{\mathsf{v}}]$ to update the binding of x to $\overline{\mathsf{v}}$.

To accurately represent both positive and negative bindings, we define our bindings similarly to [Naeem and Lhoták 2008; Naeem and Lhotak 2008]. There are four distinct types of bindings:

- **Top ($x \mapsto \top$)**: Indicates that x can be any program variable.
- **Bottom ($x \mapsto \bot$)**: Represents an invalid binding, indicating an unreachable state.
- **Positive Binding ($x \mapsto \overline{\mathsf{v}}$)**: Represents the set of variables $\overline{\mathsf{v}}$ to which x is mapped.
- **Negative Binding ($x \mapsto \neg\overline{\mathsf{v}}$)**: Represents the set of variables $\overline{\mathsf{v}}$ that x cannot be mapped to.

The meet operation $\sqcap$ is performed by set union, which combines two sets of tuples while discarding identical tuples. Specifically, $\{\overline{\langle s_1,\ e_1\rangle}\} \sqcap \{\overline{\langle s_2,\ e_2\rangle}\} = \{\overline{\langle s_1,\ e_1\rangle}, \overline{\langle s_2,\ e_2\rangle}\}$. This approach ensures high precision, as distinct analysis results for different program paths are recorded in separate tuples, rather than being merged when the paths converge at their confluence point.

At the program's entry, the initial fact is $d_0 = \langle s_0, \{\overline{x \to \top}\}\rangle$, where s_0 is the initial state of the Büchi automaton, and each VPDL variable x is bound to $\top$, allowing it to bind to any program variable during analysis.

$$
\begin{aligned}
F_{\mathsf{v}_1=\mathsf{v}_2}(e[x]) &= \begin{cases} x \mapsto \top, & x \mapsto \top \\ x \mapsto \overline{\mathsf{v}} \cup \{\mathsf{v}_1\}, & x \mapsto \overline{\mathsf{v}} \;\&\; \mathsf{v}_2 \in \overline{\mathsf{v}} \\ x \mapsto \overline{\mathsf{v}}, & x \mapsto \overline{\mathsf{v}} \;\&\; \mathsf{v}_2 \notin \overline{\mathsf{v}} \\ x \mapsto \neg\{\overline{\mathsf{v}} \cup \{\mathsf{v}_1\}\}, & x \mapsto \neg\overline{\mathsf{v}} \;\&\; \mathsf{v}_2 \in \overline{\mathsf{v}} \\ x \mapsto \neg\overline{\mathsf{v}}, & x \mapsto \neg\overline{\mathsf{v}} \;\&\; \mathsf{v}_2 \notin \overline{\mathsf{v}} \end{cases} \\
F_{\mathsf{v}_1=\mathsf{v}_2}(e) &= \bigcup_{e[x] \in e} F_{\mathsf{v}_1=\mathsf{v}_2}(e[x]) \\
F_{\mathsf{v}_1=\mathsf{v}_2}(\langle s, e\rangle) &= \langle s, F_{\mathsf{v}_1=\mathsf{v}_2}(e)\rangle \\
F_{\mathsf{v}_1=\mathsf{v}_2}(\overline{\langle s, e\rangle}) &= \bigcup_{\langle s,e\rangle \in \overline{\langle s,e\rangle}} F_{\mathsf{v}_1=\mathsf{v}_2}(\langle s, e\rangle)
\end{aligned}
$$

Fig. 8. Flow function for assignment statement v_1=v_2.

Meet $\sqcap$ over bindings:

$$
\begin{aligned}
x \mapsto \top \sqcap x \mapsto any &= x \mapsto any \\
x \mapsto \overline{\mathsf{v}} \sqcap x \mapsto \overline{\mathsf{v}}' &= \begin{cases} x \mapsto \overline{\mathsf{v}} \cap \overline{\mathsf{v}}', & \overline{\mathsf{v}} \cap \overline{\mathsf{v}}' \neq \emptyset \\ x \mapsto \bot, & \overline{\mathsf{v}} \cap \overline{\mathsf{v}}' = \emptyset \end{cases} \\
x \mapsto \overline{\mathsf{v}} \sqcap x \mapsto \neg\overline{\mathsf{v}}' &= \begin{cases} x \mapsto \bot, & \overline{\mathsf{v}} \cap \overline{\mathsf{v}}' \neq \emptyset \\ x \mapsto \overline{\mathsf{v}}, & \overline{\mathsf{v}} \cap \overline{\mathsf{v}}' = \emptyset \end{cases} \\
x \mapsto \neg\overline{\mathsf{v}} \sqcap x \mapsto \neg\overline{\mathsf{v}}' &= x \mapsto \neg\{\overline{\mathsf{v}} \cup \overline{\mathsf{v}}'\} \\
e \sqcap x \mapsto \overline{\mathsf{v}}' &= e[e[x] \sqcap x \mapsto \overline{\mathsf{v}}'] \\
e \sqcap x \mapsto \neg\overline{\mathsf{v}}' &= e[e[x] \sqcap x \mapsto \neg\overline{\mathsf{v}}']
\end{aligned}
$$

Flow function $F_{\mathsf{p}(\mathsf{v})}$:

$$
\begin{aligned}
F_{\mathsf{p}(\mathsf{v})}(\langle s,e\rangle) &= \begin{cases} \langle s', e \sqcap x \mapsto \{\mathsf{v}\}\rangle, & s \xrightarrow{p(x)} s' \\ \langle s', e \sqcap x \mapsto \neg\{\mathsf{v}\}\rangle, & s \xrightarrow{\neg p(x)} s' \\ \langle s', e\rangle, & s \xrightarrow{\neg q(x)} s' \end{cases} \\
F_{\mathsf{p}(\mathsf{v})}(\overline{\langle s,e\rangle}) &= \bigcup_{\langle s,e\rangle \in \overline{\langle s,e\rangle}} F_{\mathsf{p}(\mathsf{v})}(\langle s, e\rangle)
\end{aligned}
$$

Fig. 9. Flow function for the predicate-matching statement $\mathsf{p}(\mathsf{v})$.

4.2.2 Flow Functions. Non-trivial flow (transfer) functions are required to handle two types of statements: one for assignment statements ($\mathsf{v}_1 = \mathsf{v}_2$), which updates the dataflow facts for v_1 by incorporating the dataflow facts of v_2, and one for predicate-matching statements ($\mathsf{p}(\mathsf{v})$), which updates both the automaton state and its corresponding environment. All other statements do not modify any input dataflow facts, and their transfer functions are modeled as identity functions.

Figure 8 gives the flow function for an assignment statement $\mathsf{v}_1 = \mathsf{v}_2$. In this case, we bind x to v_1 if x is bound to v_2. The environment e is updated accordingly, and the flow function updates the input fact d by modifying the environment e of each tuple $\langle s, e\rangle$, while preserving the state s.

$$\frac{l : x = y}{\textbf{alias}(x, y)} \quad \frac{\textbf{alias(x,y)} \quad l : z = y}{\textbf{alias}(x, z)} \ [\textsc{Assign}] \qquad \frac{l_1 : x = y.f \quad l_2 : x' = z.f \quad \textbf{alias}(y, z)}{\textbf{alias}(x, x')} \ [\textsc{Load}]$$

$$\frac{l_1 : y.f = x \quad l_2 : x' = z.f \quad \textbf{alias}(y, z)}{\textbf{alias}(x, x')} \ [\textsc{Store}] \qquad \frac{l : x = a_0.f(a_1) \quad \{m'\} = \textbf{calleeOf}(l)}{\textbf{alias}(a_0, this^{m'}) \quad \textbf{alias}(a_1, p_1^{m'}) \quad \textbf{alias}(x, ret^{m'})} \ [\textsc{Call}]$$

Fig. 10. Rules for demand-driven alias analysis.

Figure 9 presents the flow function for a predicate-matching statement `p(v)`. For simplicity, we assume that predicates take a single argument, though predicates with multiple arguments can be handled by translating them into a list of single-argument statements. The meet operation for two positive bindings maps a free variable to the intersection of its values in the bindings, with a similar definition for negative bindings. The meet operation for an environment and a binding generalizes naturally. The flow function $F_{\texttt{p(v)}}$ updates the automaton state and environment e with new bindings for each tuple $\langle s, e\rangle$ in the input dataflow fact.

4.2.3 Aliases. Aliases are handled via an on-demand use-def analysis based on the inference rules in Figure 10. Two variables may alias if they can point to a common object through assignments ([Assign]) or loads/stores ([Load] and [Store]). Our alias analysis is performed inter-procedurally at a call statement only when it is a mono-call (with a unique callee function); otherwise, it is ignored ([Call]). When a statement `p(v)` matches a predicate $p(x)$, we compute the set of aliased variables for `v` in the same scope and include it in the binding of x.

Our experience with existing alias analysis tools, such as Boomerang [Späth et al. 2016], has highlighted frequent aborts due to implementation bugs and missed aliases caused by framework-injected objects. As a result, we have developed a customized alias analysis for detecting semantically recurring vulnerabilities. Inspired by IDE^{al} [Späth et al. 2017] and PATA [Li et al. 2022], this analysis prioritizes efficiency and scalability over soundness, computing aliases based on direct def-use relations. More complex aliasing rules can be explicitly specified using VPDL rules.

4.2.4 Example. Let us revisit our motivating example in Figure 1. Assume the patch is not applied. The dataflow fact before line 6 is $\langle s_1, \{\overline{x \mapsto \top}\}\rangle$. By matching the statement `Factory.createHistorySummary(wiki, spaces, page)` to p_9, the fact is updated to $\langle accept, \{wiki \mapsto \{\texttt{wiki}\}, \ldots, docRef \mapsto \top\}\rangle$, signifying a vulnerable path with the accepting state $accept$.

If the patch is applied, the predicate-matching statement at line 3 results in two tuples: $\langle s_2, \{wiki \mapsto \{\texttt{wiki}\}, \ldots, pageId \mapsto \{\texttt{pageId}\}, docRef \mapsto \top\}\rangle$ and $\langle s_1, \{wiki \mapsto \neg\{\texttt{wiki}\}, \ldots\}\rangle$. Consider the corresponding Büchi automaton in Figure 7. From state s_2, the automaton transitions to s_2 itself and s_3, but the accepting state cannot be reached. From state s_1, the automaton may transition to the accepting state $accept$ if p_9 is matched. However, the binding $wiki \mapsto \neg\{\texttt{wiki}\}$ prevents the matching of line 6 to p_9, so no vulnerability is reported.

4.3 IFDS-Based Implementation

Based on its architecture (Figure 2), VulPA's implementation primarily involves automaton generation, an IFDS-based analysis algorithm, and a demand-driven alias analysis. The Spot [Duret-Lutz et al. 2022a] library is used to convert LTL formulas into Büchi automata. We implemented a demand-driven use-def analysis to compute alias results for the IFDS solver within the current scope. Below, we discuss the implementation of our IFDS-based analysis algorithm.

4.3.1 IFDS. An IFDS problem IP is defined as a five-tuple $IP = (G^*, D, F, M, \sqcap)$ [Reps et al. 1995]. $G^* = (N^*, E^*)$ represents the supergraph, or the inter-procedural control flow graph (ICFG) of the program. The domain D denotes a finite set of dataflow facts. F, a subset of $2^D \to 2^D$, contains a set of flow functions that are distributive over the meet operator $\sqcap$ (either set union or intersection).

Finally, M maps edges in E^* to their corresponding flow functions in F. The IFDS problem is solved by computing graph reachability over an exploded graph (over G^*), where each element in D is represented as a node at every program point, and flow functions are transformed into edges.

In Section 4.2, we formulated the multi-object typestate analysis as an IFDS problem, where the flow functions in Figures 8 and 9 are distributive. We implemented this analysis using Heros [Heros 2023], a widely-used IFDS framework, and the Soot framework [Soot 2024] for tasks like parsing Java bytecode and building the inter-procedural control flow graphs.

4.3.2 Optimizations. We introduce three optimization techniques to improve the efficiency of our multi-object typestate analysis by reducing unnecessary variable bindings and, consequently, the number of facts being propagated during the analysis: (1) Bindings are maintained only for program variables accessible in the current scope. At procedure boundaries, only bindings to access paths with base variables as actual parameters are propagated to the callee. (2) If a VPDL variable x is not referenced in subsequent state transitions, its binding is simplified to $x \mapsto \top$. (3) Identical tuples resulting from these optimizations are merged.

5 Evaluation

We evaluate VulPA based on its ability to identify existing and new recurring vulnerabilities, as well as its analysis performance. Tested on 26 Java applications using rules from 34 CVEs, VulPA identified 90 new vulnerabilities with a 23.7% false positive rate, outperforming existing tools (ReDeBug, VUDDY, SourcererCC, PHunter, PPT4J, FlowDroid, and IDE^{al}), which collectively found only 13. VulPA effectively uncovers complex vulnerabilities that were missed by state-of-the-art tools. Our evaluation seeks to answer the following research questions:

- **RQ1:** Can VulPA detect known vulnerabilities?
- **RQ2:** Can VulPA detect more new recurring vulnerabilities than state-of-the-art tools by leveraging the expressive power of VPDL?
- **RQ3:** How time-efficient is VulPA's analysis?

5.1 Experiment Setup

Dataset. We evaluated VulPA using a set of real-world applications with known CVEs, listed in Table 1, to assess its ability to detect recurring vulnerabilities. To compile this dataset, we reviewed all 34,694 CVE [CVE 2024] records from March 2022 to February 2023. Of these, 119 Java-related CVEs with known patches were identified. After manual examination, we excluded 63 entries with root causes outside of Java files and removed 22 CVEs where the corresponding projects failed to compile. This left us with a refined dataset of 34 CVEs, representing 63 vulnerabilities (with some CVEs assigned to multiple vulnerabilities) across 26 applications.

Baselines. We compared VulPA with seven well-known tools, as summarized in Table 2. These tools represent a diverse set of recurring vulnerability detection approaches: ReDeBug [Jang et al. 2012] and VUDDY [Kim et al. 2017] are signature-based recurring vulnerability detection tools; SourcererCC [Sajnani et al. 2016] is a clone detection tool; PHunter [Xie et al. 2023] and PPT4J [Pan et al. 2024] identify the presence of patches to report unpatched vulnerable software versions; FlowDroid [Arzt et al. 2014] and IDE^{al} [Späth et al. 2017] are well-established tools for taint and typestate analysis, respectively.

To facilitate comprehensive program analysis, we adopted the experimental configuration proposed by IDE^{al} [Späth et al. 2017]: distinct sources are separately analyzed by the underlying IFDS solver and a time limit of 30 seconds per seed is imposed for VulPA, FlowDroid, and IDE^{al}. This constraint allows for a thorough examination while effectively managing computational resources. The three tools mentioned above were configured to start analysis from common entry points such

Table 1. The real-word dataset.

Project	#Stars	#Files	#Lines	CVEs
AntiSamy	174	37	10,235	CVE-2022-29577
Appointmentscheduling	14	166	17,824	CVE-2022-4727
Cloudsync	177	36	6,950	CVE-2022-4773
Dragonfly	7	25	1,976	CVE-2022-41967
DSpace	843	3,379	572,475	CVE-2022-31193, CVE-2022-31194, CVE-2022-31195
eionet.contreg	1	660	99,805	CVE-2022-4513
JATOS	78	200	35,756	CVE-2022-4878
jgit-cookbook	1.8k	76	5,556	CVE-2022-4817
jLEMS	8	618	56,208	CVE-2022-4583
Joget	499	615	134,711	CVE-2022-4859, CVE-2022-4560
MCPMappingViewer	215	27	4,708	CVE-2022-4494
Netty	33.2k	2,842	501,856	CVE-2022-41915
OneDev	12.8k	3,410	371,228	CVE-2022-38301, CVE-2023-24828
Portofino	180	657	72,446	CVE-2022-3952
qtiworks	66	1,056	136,733	CVE-2022-39367
RuoYi	5.4k	271	36,027	CVE-2022-32065
spring-boot-admin	12.3k	333	30,238	CVE-2022-46166
SCIFIO	86	336	79,721	CVE-2022-4493
SuperXray	1.2k	21	4,683	CVE-2022-41958
SurveyKing	2.9k	355	22,304	CVE-2022-26249
TJWS2	65	90	49,521	CVE-2022-4594
UnsafeAccessor	22	50	10,454	CVE-2022-31139
Venice	29	826	194,741	CVE-2022-36007
Vert.x-Web	1.1k	764	124,682	CVE-2023-24815
Widoco	266	34	12,063	CVE-2022-4772
Xwiki	932	6,124	871,033	CVE-2022-29253, CVE-2022-41929, CVE-2022-41936, CVE-2023-26471, CVE-2023-26475

Table 2. Overview of VulPA and state-of-the-art tools in detecting recurring vulnerabilities.

Tool	Description	Target	Vulnerability Pattern Input
ReDeBug [Jang et al. 2012]	Signature-based Recurring Vulnerability Detection (Line-Level)	Source Code	Patch Files
VUDDY [Kim et al. 2017]	Signature-based Recurring Vulnerability Detection (Method-Level)	Source Code	Pre-Collected Vulnerable Method Signatures (Online Database)
SourcererCC [Sajnani et al. 2016]	Clone Detection (Token-Level Similarity)	Source Code	Patch-Related Vulnerable Files
PHunter [Xie et al. 2023]	Patch Presence Testing (Candidate Method Localization via Similarity)	Bytecode	Patch File and Both Pre-Patch and Post-Patch Bytecode
PPT4J [Pan et al. 2024]	Patch Presence Testing (Precise Method Localization via Signature Matching)	Bytecode	Patch File, Both Pre-Patch and Post-Patch Source Code and Bytecode
FlowDroid [Arzt et al. 2014]	Taint Analysis	Bytecode	Taint Sources, Sinks, and Sanitizers
IDE^{al} [Späth et al. 2017]	Typestate Analysis	Bytecode	State Transition Rules
VulPA	Multi-Object Typestate Analysis	Bytecode	VPDL Rules

as main methods, RESTful endpoints, and web framework handlers. The rest five tools–ReDeBug, VUDDY, SourcererCC, PHunter, and PPT4J–used their default parameter settings, which are considered to yield overall best performance in practice. All detection results were independently verified by two researchers to ensure the correct classification of true and false positives.

Computing Environment. All experiments were conducted on a 64-core Intel(R) Xeon(R) CPU E7-4809 v3 @ 2.00 GHz machine with 1 TB of memory, running Clear Linux OS 41370. For the three tools, FlowDroid, IDE^{al}, and VulPA, we allocated a maximum of 128 GB JVM heap space for each run, which is sufficient for these tools to analyze all projects in Table 1 with their respective detection rules.

5.2 RQ1: Ability to Detect Existing Vulnerabilities

We ran VulPA alongside the seven baseline tools to assess their ability to detect the 34 known CVEs in Table 1. To support the analysis of each CVE listed in Table 1, three researchers developed and validated the detection rules for FlowDroid, IDE^{al}, and VulPA. To ensure a fair comparison and minimize bias, these rules were derived from the same underlying automata generated by VPDL, with each rule being cross-reviewed by other researchers for accuracy. In total, we manually created 34 VPDL rules for VulPA (one for each CVE), 8 rules for FlowDroid, and 11 rules for IDE^{al} (Table 3). Additionally, 26 CVEs could not be expressed as taint rules for FlowDroid, and 23 CVEs could not be expressed as typestate rules for IDE^{al}.

For the remaining five tools, we provided the relevant source code, bytecode, and patch files for each CVE as detection inputs: ReDeBug utilizes patch files as inputs; VUDDY relies on an online database, and our attempts to construct a local database using open-source tools were unsuccessful, as they did not generate any reports. This makes it nearly impossible to manually configure or enhance this tool effectively. SourcererCC takes pre-patched vulnerable source files as inputs for file-level clone detection, but its method-level detection is less reliable due to limitations in its slicing technique, which often results in incomplete methods missing critical code segments. PHunter and PPT4J require patch files along with the corresponding pre- and post-patch source or bytecode files to identify the existence of patches. When these tools determine that the target vulnerable project has not been patched, we consider it a successful identification of all corresponding vulnerabilities.

Table 3 reports our results, showing that these 34 CVEs correspond to a total of 63 known vulnerabilities. In analyzing the detection results, we focus on vulnerabilities reported in the source locations of these known vulnerabilities, excluding bugs found in other methods or files. The three tools—VulPA, SourcererCC, and PPT4J—successfully detected all 63 vulnerabilities. SourcererCC identified all vulnerable file clones, as its input pre-patched files were exact copies in the analyzed target programs. PPT4J correctly reported missing patches for all 34 CVEs, though it could not extract patch features for 4 CVEs, which were also considered as missing patches. PHunter identified 49 vulnerabilities across 29 CVEs. FlowDroid detected 16 vulnerabilities across 7 CVEs, while IDE^{al} reported 2 vulnerabilities. ReDeBug detected 14 vulnerabilities across 8 CVEs, and VUDDY identified 9 vulnerabilities across 6 CVEs.

Limited Expressiveness of FlowDroid and IDE^{al}. Out of the 34 CVEs, 11 can be abstracted as taint analysis problems solvable by FlowDroid. Additionally, 14 CVEs can be expressed in typestate rules for IDE^{al}, which are more general than standard taint analysis. However, both IDE^{al} and FlowDroid are limited to specifying method invocations, making them unsuitable for rules involving field accesses without code modification. As a result, we were able to specify only 11 rules for IDE^{al} and 8 rules for FlowDroid.

False Negatives of Other Existing Tools. ReDeBug detects recurring vulnerabilities by matching code hunks from patch files, which can lead to false negatives and false positives if the surrounding context is irrelevant to the vulnerability or if the number of lines in a hunk falls below the sliding window size (4 lines by default), due to whitespaces or comments. We experimented with various parameter settings beyond the default configuration, but these adjustments did not detect any additional vulnerabilities. VUDDY requires exact matching of patches in its online database and fails to detect methods with code modifications irrelevant to the targeted vulnerabilities.

Table 3. Detection of 63 known vulnerabilities across 34 CVEs by VulPA and state-of-the-art tools: ReDeBug, VUDDY, SourcererCC, PHunter, PPT4J, FlowDroid, and IDE^{al}.

CVE	# Vul	ReDeBug	VUDDY	SourcererCC	PHunter	PPT4J	FlowDroid	IDE^{al}	VulPA
CVE-2022-29577	1		1	1	1	1			1 ✓
CVE-2022-4727	1			1	1	1			1 ✓
CVE-2022-4773	1			1		1	1 ✓	✓	1 ✓
CVE-2022-41967	1			1	1	1			1 ✓
CVE-2022-31193	1		1	1		1	1 ✓	✓	1 ✓
CVE-2022-31194	4		4	4	4	4			4 ✓
CVE-2022-31195	1		1	1	1	1			1 ✓
CVE-2022-4513	2	2		2	2	2			2 ✓
CVE-2022-4878	1			1	1	1			1 ✓
CVE-2022-4817	1			1	1	1	1 ✓	1 ✓	1 ✓
CVE-2022-4583	1			1	1	1			1 ✓
CVE-2022-4859	2	2		2		2	✓	✓	2 ✓
CVE-2022-4560	6			6	6	6			6 ✓
CVE-2022-4494	1			1	1	1			1 ✓
CVE-2022-41915	2			2	2	2			2 ✓
CVE-2022-38301	1			1	1	1			1 ✓
CVE-2023-24828	9	1		9		9		✓	9 ✓
CVE-2022-3952	1			1	1	1	1 ✓	1 ✓	1 ✓
CVE-2022-39367	1			1	1	1			1 ✓
CVE-2022-32065	1	1	1	1	1	1			1 ✓
CVE-2022-46166	9	5		9	9	9	9 ✓	✓	9 ✓
CVE-2022-4493	1			1	1	1			1 ✓
CVE-2022-41958	3			3	3	3	2 ✓	✓	3 ✓
CVE-2022-26249	1			1	1	1			1 ✓
CVE-2022-4594	1	1		1	1	1			1 ✓
CVE-2022-31139	1			1	1	1		✓	1 ✓
CVE-2022-36007	1		1	1	1	1	1 ✓	✓	1 ✓
CVE-2023-24815	1	1		1	1	1			1 ✓
CVE-2022-4772	1	1		1	1	1			1 ✓
CVE-2022-29253	1		0/1	1	1	1			1 ✓
CVE-2022-41929	1			1	1	1			1 ✓
CVE-2022-41936	1			1	1	1			1 ✓
CVE-2023-26471	1			1		1		✓	1 ✓
CVE-2023-26475	1			1	1	1			1 ✓
# CVE$_{reported}$	34	8	6	34	29	34	7	2	34
# Vul$_{reported}$	63	14	9	63	49	63	16	2	63

Note: ✓ indicates that the detection rule has been implemented manually.

PHunter missed 5 CVEs: 3 due to missing patch features and 2 because of exceptions during patch condition evaluation in its similarity analysis. FlowDroid failed to detect 26 CVEs that could not be configured with taint analysis rules, and 3 additional vulnerabilities due to exceptions in Soot, resulting in 47 undetected vulnerabilities across 28 CVEs. Surprisingly, IDE^{al} only reported 2 vulnerabilities, despite its theoretical advantage over FlowDroid. We suspect this may be due to bugs in its implementation, which warrants further investigation.

VPDL demonstrates superior expressiveness, as it allows specifying all 34 CVEs in our dataset and enables VulPA to detect all 63 known vulnerabilities across them.

5.3 RQ2: Ability to Detect New Vulnerabilities

In this second RQ, we ran VulPA and the seven baseline tools on the set of projects in Table 1 to detect new recurring vulnerabilities, where all known CVEs in Table 1 have been patched. For each project, we ran each static analysis tool with all available detection rules enabled, i.e., each tool attempted to detect the recurrence of all 34 studied CVEs in each analyzed project.

Table 4. Detection of new vulnerabilities by VulPA and state-of-the-art tools: ReDeBug, VUDDY, SourcererCC, PHunter, PPT4J, FlowDroid, and IDE^{al}. The **#Real Result** represents the total number of true positives (excluding duplicate reports) reported by each tool for each project.

Project	#Real Result	ReDeBug TP/FP	VUDDY TP/FP	SourcererCC TP/FP	PHunter TP/FP	PPT4J TP/FP	FlowDroid TP/FP	IDE^{al} TP/FP	VulPA TP/FP
AntiSamy	0	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Appointmentscheduling	0	0 / 3	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Cloudsync	5	0 / 3	0 / 0	0 / 1	0 / 0	0 / 0	1 / 0	0 / 0	5 / 0
Dragonfly	0	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
DSpace	22	0 / 8	0 / 0	0 / 6	0 / 2	0 / 1	7 / 0	0 / 0	22 / 4
eionet.contreg	3	0 / 49	0 / 0	0 / 2	0 / 2	0 / 2	0 / 0	0 / 0	3 / 0
JATOS	4	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	4 / 0
jgit-cookbook	0	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
jLEMS	1	0 / 4	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	1 / 0
Joget	8	0 / 19	0 / 0	0 / 8	0 / 3	0 / 0	0 / 0	0 / 0	8 / 5
MCPMappingViewer	4	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	4 / 3
Netty	0	0 / 23	0 / 0	0 / 2	0 / 0	0 / 2	0 / 0	0 / 0	0 / 0
OneDev	0	0 / 108	0 / 0	0 / 10	0 / 3	0 / 1	0 / 0	0 / 0	0 / 1
Portofino	0	0 / 8	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 1
qtiworks	3	0 / 0	0 / 0	0 / 1	0 / 2	0 / 1	0 / 0	0 / 0	3 / 2
RuoYi	0	0 / 6	0 / 0	0 / 1	0 / 3	0 / 0	0 / 0	0 / 0	0 / 1
spring-boot-admin	0	0 / 0	0 / 0	0 / 9	0 / 2	0 / 0	0 / 0	0 / 0	0 / 0
SCIFIO	0	0 / 12	0 / 0	0 / 1	0 / 2	0 / 1	0 / 0	0 / 0	0 / 2
SuperXray	4	0 / 0	0 / 0	0 / 3	0 / 3	0 / 0	0 / 0	0 / 1	4 / 0
SurveyKing	4	0 / 0	0 / 0	0 / 1	0 / 2	0 / 0	0 / 0	0 / 0	4 / 0
TJWS2	7	0 / 46	0 / 0	0 / 1	0 / 3	0 / 1	0 / 0	0 / 0	7 / 3
UnsafeAccessor	0	0 / 3	0 / 0	0 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Venice	6	0 / 0	0 / 0	0 / 1	0 / 4	0 / 0	2 / 0	0 / 0	6 / 2
Vert.x-Web	9	0 / 23	0 / 0	0 / 1	0 / 2	0 / 0	2 / 0	0 / 0	9 / 2
Widoco	1	0 / 5	0 / 0	0 / 1	0 / 1	0 / 0	0 / 0	0 / 0	1 / 1
Xwiki	9	0 / 8	0 / 0	0 / 5	0 / 2	0 / 2	1 / 0	0 / 0	9 / 1
Total	90	0 / 328	0 / 0	0 / 63	0 / 36	0 / 11	13 / 0	0 / 1	90 / 28

Table 4 summarizes our results, including the number of true positives (TP) and false positives (FP) reported by each tool. VulPA reported a total of 118 vulnerabilities, comprising 90 real recurring vulnerabilities and 28 false positives, resulting in a false positive rate of 23.72%. In contrast, the seven baseline tools were significantly less effective.

ReDeBug generated 328 false warnings, primarily due to the loss of precision when abbreviating code hunks from source code patches. These abbreviated code lines often included irrelevant details unrelated to the vulnerability root causes. VUDDY reported 5 CVEs whose signatures matched patch signatures in its online database, but these were unrelated to the CVEs we studied. Consequently, both signature-based tools failed to identify any new recurring vulnerabilities.

SourcererCC, PHunter, and PPT4J did not detect any new vulnerabilities. SourcererCC produced 63 false positives, as it was unable to distinguish between pre- and post-patch files. PHunter incorrectly reported missing patches for 3 CVEs across 16 projects, leading to 36 false positives. For instance, the patch for CVE-2022-4513 modified two methods, `setUri()` and `setSearchTag()`, each containing only a single store statement. This simplicity in the patch caused incorrect method matching and false positives. PPT4J reported 11 false positives: 6 due to errors in similar patch presence testing, and 5 due to its inability to extract patch-related features. While some of these false positives could be mitigated with alternative configuration settings, such adjustments might introduce potential false negatives.

FlowDroid reported 13 true positives with no false positives, all of which were also identified by VulPA. IDE^{al} produced only 1 report, which was a false positive.

The eight tools together reported a total of 90 real vulnerabilities, with VulPA specifically identifying all of them. Additionally, the other seven tools collectively reported only 13 new vulnerabilities. This demonstrates the superiority of VulPA over the state of the art in detecting new recurring vulnerabilities through its novel multi-object typestate analysis.

False Positives of VulPA. Among the 28 false positives reported by VulPA, 26 were due to alternative fixes that differed from our VPDL description. For example, CVE-2022-31193 (an open redirect vulnerability) was patched by adding a `String.startsWith()` method invocation to sanitize the tainted string before reaching the sink `sendRedirect()`. However, an alternative fix involves converting the tainted string to a `URI` object and checking it with regular expressions. Our VPDL rule, based on the patch for CVE-2022-31193, resulted in false positives for such cases. These false positives can be potentially avoided by refining the VPDL rules to account for these alternative sanitizing mechanisms. The remaining 2 false positives reported by VulPA were caused by imprecise alias analysis, which can be improved in future work.

False Negatives of Existing Tools. The five tools—ReDeBug, VUDDY, SourcererCC, PHunter, and PPT4J—failed to identify any new vulnerabilities. ReDeBug, VUDDY, and SourcererCC detect recurring vulnerabilities or code clones through token-based similarity matching, which is inherently limited in detecting semantically recurring vulnerabilities. PPT4J tests for patch presence in methods with identical signatures to those of the patched methods, making it incapable of detecting new vulnerabilities in distinct methods. PHunter's method similarity matching approach, although potentially capable of finding new vulnerabilities, proved insufficient for detecting new recurring vulnerabilities in our experiments. FlowDroid missed 77 real bugs, with 76 missed due to its taint analysis rules not covering the relevant cases. The remaining false negative was caused by timeouts triggered by the underlying IFDS solver. IDE^{al} missed all 90 new vulnerabilities, due to limitations in expressing vulnerability root causes and implementation bugs.

VulPA has proven effective in detecting real-world vulnerabilities, identifying issues that existing tools miss with high precision. Its precision can be further enhanced with refined VPDL descriptions.

5.4 RQ3: Time Efficiency in Vulnerability Detection

We measured the analysis time for each tool in detecting new vulnerabilities during the experiment conducted for RQ2, as described in Section 5.3. SourcererCC required an average of only 6 seconds per project. The two signature-based tools also completed within seconds, averaging 11 seconds for ReDeBug and 22 seconds for VUDDY. The two patch presence testing tools exhibited distinct characteristics: PPT4J tested for patch existence only in methods with identical signatures to those of patched methods, taking an average of 24.36 seconds per project. In contrast, PHunter performed similarity matching for all methods, averaging 8,079.31 seconds per project.

Figure 11 compares the analysis times of VulPA, FlowDroid, and IDE^{al}, running with 34 VPDL rules, 8 taint rules, and 11 typestate rules, respectively. VulPA successfully analyzed 19 projects within 10 minutes, and 24 projects were completed within 1 hour. The most time-consuming project, Vert.x-Web, took 5.9 hours to analyze. This project required significantly more time than others due to its large number of third-party libraries: it was compiled into 49 JAR files and, compared to other benchmarks, consumed notably more memory to load all these JAR files.

FlowDroid completed its analysis within 1 hour for 25 out of the total 26 projects. However, its analysis time was significantly longer than VulPA's: the average analysis time (6,206 seconds) was 3.16 × longer than VulPA's (1,491 seconds), and for the Vert.x-Web benchmark, FlowDroid took 6.68 × longer (141,604 seconds for FlowDroid vs. 21,199 seconds for VulPA). IDE^{al} also analyzed 25 projects within 1 hour but was faster than VulPA for only four benchmarks: DSpace, Joget, Vert.x-Web, and Xwiki. On average, IDE^{al}'s analysis time was 1.67 × faster than VulPA's

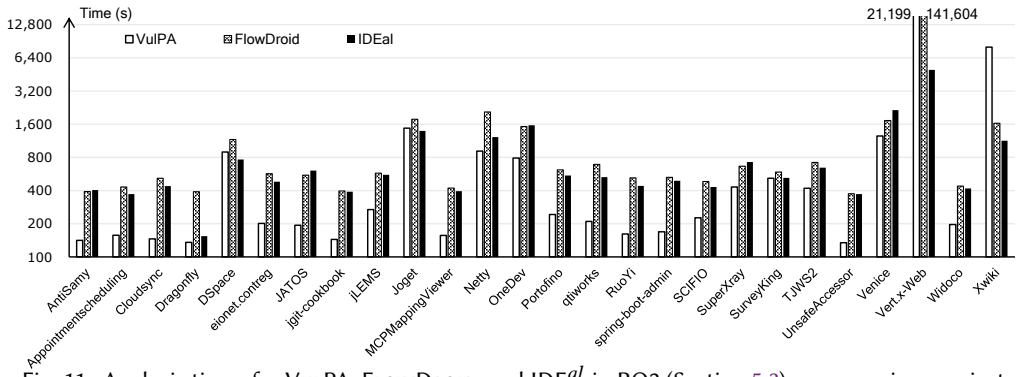

Fig. 11. Analysis times for VulPA, FlowDroid, and IDE^{al} in RQ2 (Section 5.3) across various projects.

(851 seconds compared to 1,419 seconds). Among these three static analysis tools evaluated, IDE^{al} was the fastest, though its efficiency is questionable due to implementation bugs. VulPA, on the other hand, is significantly faster than FlowDroid, thanks to its more efficient handling of aliases, designed to trade soundness for scalability (Section 4.2.3).

Compared to non-static-analysis tools—ReDeBug and VUDDY (signature-based), SourcererCC (clone-based), and PHunter and PPT4J (patch-presence-testing-based)—static analysis tools require much longer analysis times but, as demonstrated here with VulPA, can be designed to offer significantly better detection capabilities.

In our experiments, we imposed a 30-second time limit per seed for VulPA, FlowDroid, and IDE^{al} to ensure thorough detection. VulPA had 3.45% (226 out of 6,560) of seeds *timed out*, FlowDroid had 10.97% (62 out of 565), and IDE^{al} had 2.57% (13 out of 505). A timeout forces the IFDS solver to terminate early, potentially missing some vulnerabilities. We conducted additional experiments by gradually increasing the time limit from 30 seconds to 10 minutes per seed, but this extension did not yield any new vulnerability reports or reduce false positives.

VulPA successfully analyzed 24 out of 26 projects within 1 hour, demonstrating significantly higher efficiency compared to the classic taint analysis tool, FlowDroid. This performance advantage is particularly notable given that VulPA applied more detection rules and identified a substantially greater number of new vulnerabilities.

5.5 Case Studies

We present several case studies to illustrate the process of deriving VPDL rules from existing patches, how these rules precisely capture the root causes of vulnerabilities, and how refining VPDL rules can improve detection capabilities.

Writing VPDL Rules. The intuitive syntax of VPDL enables developers to quickly translate vulnerability patches into actionable detection rules. As illustrated in Figure 12, the path traversal vulnerability CVE-2022-4494 in the Widoco project (Figure 12(a)) arises when a file path is constructed using `ze.getName()`, which could allow directory traversal outside `destDir`, including unauthorized directories. The fix (lines 3 and 4) ensures the path is validated as a subdirectory of `destDir`. Figure 12(b) shows the corresponding VPDL rule, which differs from Figure 12(a) by representing program variables (e.g., `fileName` and `newfile`) as VPDL meta variables (e.g., *name* and *file*) and encoding the patch logic using the "`not`" operator. The simplicity of VPDL's syntax allows developers to easily derive detection rules from patched code examples, potentially enabling future advancements in automated rule extraction techniques.

Describing Vulnerability Root Causes. Figure 13 illustrates the remote code execution vulnerability CVE-2023-26476 (from the Xwiki project) and its corresponding VPDL rules. In Figure 13(a),

```
1   String fileName = ze.getName();
2   File newFile = new File(destDir, fileName);
3 + if (!newFile.toPath().normalize().startsWith(
        destDir.toPath().normalize())) {
4 +   throw new IOException("Bad zip entry"); }
5   newFile.mkdirs();
```

(a) Code snippets in the Widoco project

```
1  @CVE-2022-4494@
2  Variable name, file, path1, path2;
3  @@
4  name = ZipEntry.getName();
5  file = new File(*, name);
6  not {
7    path = file.toPath();
8    path2 = path.normalize();
9    path2.startsWith(*); }
10 file.mkdirs();
```

(b) The VPDL rule

Fig. 12. CVE-2022-4494.

```
1   xclass.addTextAreaField("highlight", "Highlighted", 40, 2);
2   xclass.addNumberField("replyto", "Reply To", 5, "integer");
3   String name = "comment";
4 - xclass.addTextAreaField(name, "Comment", 40, 5);
5 + xclass.addTextAreaField(name, "Comment", 40, 5, true);
6   TextAreaClass comment = xclass.getField(name);
7   comment.setEditor((String) null);
```

(a) Code snippet in the Xwiki project

```
1  @CVE-2023-26475@
2  BaseClass xclass;
3  String name;
4  Variable comment;
5  @@
6  xclass.addTextAreaField(name, *, *, *);
7  comment = xclass.getField(name);
8  comment.setEditor(*);
```

(b) The VPDL rule

Fig. 13. CVE-2023-26475.

line 4 adds an unrestricted area field by calling `xclass.addTextAreaField()` without setting the `restrict` attribute. Lines 6 and 7 make this field editable via `xclass.getField().setEditor()`, enabling unrestricted comment editing. The fix replaces line 4 with line 5, correctly restricting the field. This vulnerability involves implicit references across variables (`xclass`, `name`, and `comment`) and APIs, making it challenging for dataflow-based analyses to track such relationships.

The VPDL rules in Figure 13(b) capture these relationships (lines 6–8), precisely describing the root cause. Notably, ignoring these relationships and disallowing unrestricted area fields (i.e., removing lines 7–8) would cause VulPA to report 63 false positives.

Refining VPDL Rules. As shown in Figure 3, we applied a single VPDL rule to specify CVE-2022-41936 for the Xwiki application. This approach led to the discovery of two previously unknown vulnerabilities, which were promptly acknowledged and validated by the developers. Building on this success, we refined the rule incrementally. By making minor adjustments to bug-triggering sensitive operations—such as replacing the call to `createHistorySummary()` (line 9) with the `createWiki()` method, which returns the content of the queried page—we uncovered an additional 24 new vulnerabilities (not included in Table 3). These newly detected cases are currently being examined by the developers to assess the potential for sensitive information leakage.

5.6 Disscussion

5.6.1 Threats to Validity. The effectiveness of VulPA relies on manually written VPDL rules, which may vary based on differing interpretations of CVEs. To address this, we conducted peer reviews to ensure consistent rule derivation. To mitigate the risk of incorrectly validated bug reports, two researchers independently cross-checked each report. Furthermore, we submitted all valid reports[1] to developers via issues, pull requests, and JIRA tickets.

To date, we have received 42 responses from developers across nine projects: Xwiki, Widoco, Qtiworks, SurveyKing, SuperXray, JATOS, jLEMS, DSpace, and CloudSync. Among these, 22 reports pertain to projects or outdated versions no longer maintained (e.g., CloudSync, DSpace, Qtiworks, and SuperXray). Twenty reports were fixed and merged (e.g., Xwiki, Widoco, SurveyKing, jLEMS,

[1]The list of reports can be viewed at https://caoliqingstudio.github.io/FSE25-VulPA/FSE25-VulPA-Report.html

JATOS, and DSpace), with four confirmed as CVEs: Xwiki's two reports were assigned CVE-2024-45591, while SurveyKing's two reports received CVE-2024-31567 and CVE-2024-31568.

5.6.2 Limitations and Future Work. VulPA currently supports only Java applications and cannot detect cross-language vulnerabilities. Additionally, VPDL is unable to express vulnerabilities originating from configuration files or annotations, which limits its detection scope. To address these limitations, future work could focus on extending VPDL to support multiple programming languages and a broader range of vulnerability patterns. We also plan to explore the use of large language models (LLMs) to automatically generate detection rules from vulnerability databases and security reports, reducing the manual effort required for rule construction.

6 Related Work

In this section, we review prior work closely related to this research.

6.1 Recurring Vulnerability Detection

Many recurring vulnerability detection techniques focus on identifying clones of vulnerable code [Jang et al. 2012; Kang et al. 2022; Kim et al. 2017; Pham et al. 2010; Shi et al. 2024; Xiao et al. 2020]. These methods generate signatures from vulnerability fixes and match them to the analyzed code. ReDeBug [Jang et al. 2012] and VUDDY [Kim et al. 2017] use token sequences from adjacent code lines, but cannot detect vulnerabilities with the same root cause but different structures, as shown in this work. In contrast, methods using dependence graph signatures [Kang et al. 2022; Pham et al. 2010; Shi et al. 2024; Xiao et al. 2020; Zhan et al. 2024; Zhang et al. 2022] better preserve semantic relationships and handle structural variations. However, extracted signatures may inadvertently capture unrelated code patterns due to varying patch quality [Xu et al. 2022].

Static analysis methods detect recurring vulnerabilities using dedicated algorithms [Huang et al. 2024; Krüger et al. 2017; Lu et al. 2022; Rahaman et al. 2019; Singleton et al. 2020; Son et al. 2011; Yan et al. 2018], often requiring domain expertise to address specific vulnerabilities like missing permission checks or improper cryptography use. Some tools, such as FlowDroid [Arzt et al. 2014], RAPID [Emmi et al. 2021], and others [CodeQL 2024; Fink et al. 2008; Hallem et al. 2002; Jaspan 2008; Pradel et al. 2012; Topl 2024], utilize configurable specifications but still demand significant expertise, particularly for domain-specific vulnerabilities [Habib and Pradel 2018; Liu et al. 2020]. In contrast, VulPA introduces a new vulnerability specification language resembling patch representations, facilitating the detection of vulnerability variants and domain-specific issues, and lowering the barrier for domain experts with limited program analysis expertise.

6.2 Model Checking

Model checking systematically verifies whether a system model satisfies properties specified in temporal logic, typically using Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). CTL allows reasoning over branching paths and has been applied to compiler optimization [Lacey et al. 2004], API evolution [Brunel et al. 2009; INRIA 2024], and program verification [Clarke et al. 1986; Song and Touili 2014]. LTL, suited for linear execution sequences, is effective for analyzing specific traces [Armando et al. 2009; Khoury et al. 2016; Morse et al. 2015].

Büchi automata are equivalent to LTL formulas, a key result in model checking that enables the automated verification of LTL properties over system models. Many model checking tools, such as Spin [Holzmann 2004], NuSMV [Cimatti et al. 1999], and Spot [Duret-Lutz et al. 2022a], can perform the translation from LTL formulas to Büchi automata.

6.3 IFDS/IDE

The Inter-procedural Finite Distributive Subset (IFDS) algorithm, introduced by Reps et al. [Reps et al. 1995], solves inter-procedural data-flow problems over finite domains with distributive flow functions. It has since been extended and optimized [Gui et al. 2023; He et al. 2023; Li et al. 2024a, 2021, 2024b; Naeem et al. 2010], and incorporated into major analysis frameworks such as Soot, Heros, and WALA [Heros 2023; Soot 2024; WALA 2024].

IFDS underpins many static analysis tools [Arzt et al. 2014; He et al. 2019, 2018; Li et al. 2015] used for taint tracking and vulnerability detection.

The IDE algorithm [Sagiv et al. 1996] extends IFDS to more precisely model value transformations across procedures. Both IFDS and IDE have been widely used in typestate analysis [Emmi et al. 2021; Naeem and Lhoták 2008; Späth et al. 2017], which tracks object states throughout execution.

6.4 Typestate Analysis

Typestate analysis tracks object state transitions across method calls to detect illegal behaviors. xgcc [Hallem et al. 2002] uses domain-specific languages like Metal to specify transitions, while IDE^{al} [Späth et al. 2017] improves precision with demand-driven alias analysis and strong updates. Fusion [Jaspan 2008] specifies temporal constraints between objects using intraprocedural "relations." Naeem and Lhoták [2008]; Naeem and Lhotak [2008] adopt the IFDS/IDE framework with context- and flow-sensitive analysis, using tracematches [Allan et al. 2005] for multi-object properties. Tac [Yan et al. 2017] boosts use-after-free detection precision by combining typestate analysis with machine learning. Accumulation analysis [Kellogg et al. 2022] soundly checks typestate properties without alias information for a special class of typestate problems.

While existing approaches aim to balance precision and scalability across diverse application scenarios, VulPA introduces an innovative method for detecting recurring vulnerabilities. It features a new vulnerability specification language that mirrors patch representations, enabling the identification of both vulnerability variants and domain-specific vulnerabilities. This approach lowers the barrier for domain experts with minimal program analysis expertise to define and implement vulnerability patterns. Additionally, VulPA leverages the well-established IFDS algorithm to analyze program paths that correspond to these vulnerability patterns.

7 Conclusion

In this paper, we introduce a VPDL language to accurately describe vulnerability root causes and develop an IFDS-based multi-object typestate analysis algorithm to detect vulnerabilities specified in VPDL rules. We have implemented a tool, VulPA, that can precisely detect recurring vulnerabilities involving complex data and control dependencies across multiple objects. Our evaluation on real-world cases demonstrates VulPA's effectiveness in detecting both known and previously unknown vulnerabilities, surpassing state-of-the-art tools.

8 Data-Availability Statement

The artifact is publicly at available [Cao 2025].

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (62132020, 62402474, and 62202452), the China Postdoctoral Science Foundation (2024M753295), and Zhongguancun Laboratory.

References

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 345–364. https://doi.org/10.1145/1094811.1094839

Alessandro Armando, Roberto Carbone, and Luca Compagna and. 2009. LTL model checking for security protocols. *Journal of Applied Non-Classical Logics* 19, 4 (2009), 403–429. https://doi.org/10.3166/jancl.19.403-429 arXiv:https://doi.org/10.3166/jancl.19.403-429

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

Dirk Beyer, Sumit Gulwani, and David A Schmidt. 2018. Combining model checking and data-flow analysis. *Handbook of Model Checking* (2018), 493–540.

Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2009. A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA, 114–126.

Doron Bustan, Alon Flaisher, Orna Grumberg, Orna Kupferman, and Moshe Y Vardi. 2005. Regular vacuity. In *Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005. Proceedings 13*. Springer, 191–206.

Liqing Cao. 2025. VulPA: Detecting Semantically Recurring Vulnerabilities with Multi-object Typestate Analysis (Artifact). (4 2025). https://doi.org/10.6084/m9.figshare.27002680.v2

Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 1999. NuSMV: A new symbolic model verifier. In *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*. Springer, 495–499.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (April 1986), 244–263. https://doi.org/10.1145/5397.5399

CodeQL. 2024. *github/codeql: CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security*. Retrieved April 3, 2024 from https://github.com/github/codeql

CVE. 2024. *CVE Website*. Retrieved March 23, 2024 from https://www.cve.org/

Giuseppe De Giacomo, Moshe Y Vardi, et al. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces.. In *Ijcai*, Vol. 13. 854–860.

Stéphane Demri and Deepak d'Souza. 2007. An automata-theoretic approach to constraint LTL. *Information and Computation* 205, 3 (2007), 380–415.

Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022a. From Spot 2.0 to Spot 2.10: What's New?. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22) (Lecture Notes in Computer Science, Vol. 13372)*. Springer, 174–187. https://doi.org/10.1007/978-3-031-13188-2_9

Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, et al. 2022b. From spot 2.0 to spot 2.10: what's new?. In *International Conference on Computer Aided Verification*. Springer, 174–187.

Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. RAPID: checking API usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1416–1426.

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.

Yujiang Gui, Dongjie He, and Jingling Xue. 2023. Merge-Replay: Efficient IFDS-Based Taint Analysis by Consolidating Equivalent Value Flows. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 319–331. https://doi.org/10.1109/ASE56229.2023.00027

Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/3238147.3238213

Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 69–82. https://doi.org/10.1145/512529.512539

Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. 2023. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3597926.3598041

Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 267–279.

Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 167–177. https://doi.org/10.1145/3238147.3238185

Heros. 2023. *soot-oss/heros: IFDS/IDE Solver for Soot and other frameworks*. Retrieved March 30, 2024 from https://github.com/soot-oss/heros

Gerard J Holzmann. 2004. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading.

Yongheng Huang, Chenghang Shi, Jie Lu, Haofeng Li, Haining Meng, and Lian Li. 2024. Detecting Broken Object-Level Authorization Vulnerabilities in Database-Backed Applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2934–2948. https://doi.org/10.1145/3658644.3690227

Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.

INRIA. 2024. *Coccinelle: A Program Matching and Transformation Tool for Systems Code*. Retrieved March 22, 2024 from https://coccinelle.gitlabpages.inria.fr/website/

Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

Ciera Jaspan. 2008. Checking framework interactions with relationships. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA Companion '08)*. Association for Computing Machinery, New York, NY, USA, 901–902. https://doi.org/10.1145/1449814.1449899

Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1695–1708. https://doi.org/10.1145/3548606.3560664

Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D Ernst. 2022. Accumulation analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*.

Raphaël Khoury, Sylvain Hallé, and Omar Waldmann. 2016. Execution trace analysis using ltl-fo. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 356–362.

Seulbae Kim and Heejo Lee. 2018. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Computers & Security* 77 (2018), 720–736. https://doi.org/10.1016/j.cose.2018.02.007

Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–936.

David Lacey, Neil D Jones, Eric Van Wyk, and Carl Christian Frederiksen. 2004. Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation* 17 (2004), 173–206.

Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Lian Li, and Lin Gao. 2024a. Boosting the Performance of Multi-Solver IFDS Algorithms with Flow-Sensitivity Optimizations. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 296–307. https://doi.org/10.1109/CGO57630.2024.10444884

Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 236–247. https://doi.org/10.1109/CGO51591.2021.9370311

Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Jingling Xue. 2024b. Boosting the Performance of Alias-Aware IFDS Analysis with CFL-Based Environment Transformers. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 364 (Oct. 2024), 29 pages. https://doi.org/10.1145/3689804

Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 859–872. https://doi.org/10.1145/3503222.3507770

Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1547–1559. https://doi.org/10.1145/3377811.3380923

Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2145–2158. https://doi.org/10.1145/3548606.3560589

Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. 2015. Model checking LTL properties over ANSI-C programs with bounded traces. *Software & Systems Modeling* 14 (2015), 65–81.

Nomair A Naeem and Ondrej Lhoták. 2008. Extending typestate analysis to multiple interacting objects. *OOPSLA'08: Proceedings of Object-Oriented Programming, Systems, Languages and Applications* (2008).

Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like analysis of multiple interacting objects. *SIGPLAN Not.* 43, 10 (Oct. 2008), 347–366. https://doi.org/10.1145/1449955.1449792

Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical extensions to the IFDS algorithm. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 124–144.

Yoann Padioleau, Julia L. Lawall, and Gilles Muller. 2007. SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers. *Electronic Notes in Theoretical Computer Science* 166 (2007), 47–62. https://doi.org/10.1016/j.entcs.2006.07.022 Proceedings of the ERCIM Working Group on Software Evolution (2006).

Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 447–456. https://doi.org/10.1145/1858996.1859089

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. 925–935. https://doi.org/10.1109/ICSE.2012.6227127

Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.

David A. Schmidt. 1998. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 38–48. https://doi.org/10.1145/268946.268950

Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. 2024. RecurScan: Detecting Recurring Vulnerabilities in PHP Web Applications. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1746–1755. https://doi.org/10.1145/3589334.3645530

Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. 2020. Cryptotutor: Teaching secure coding practices through misuse pattern detection. In *Proceedings of the 21st Annual Conference on Information Technology Education*. 403–408.

Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.

Fu Song and Tayssir Touili. 2014. Pushdown model checking for malware detection. *International Journal on Software Tools for Technology Transfer* 16, 2 (2014), 147–173.

Soot. 2024. *soot-oss/soot: Soot - A Java optimization framewor.* Retrieved March 30, 2024 from https://github.com/soot-oss/soot

Johannes Späth, Karim Ali, and Eric Bodden. 2017. Ide al: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Topl. 2024. *Topl | Infer.* Retrieved February 8, 2025 from https://fbinfer.com/docs/1.1.0/checker-topl

Moshe Y Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society.

WALA. 2024. *wala/WALA: T.J. Watson Libraries for Analysis, with frontends for Java, Android, and JavaScript, and may common static program analyses.* Retrieved May 28, 2024 from https://github.com/wala/WALA

Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 932–944.

Pierre Wolper. 1983. Temporal logic can be more expressive. *Information and control* 56, 1-2 (1983), 72–99.

Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. 1165–1182.

Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and efficient patch presence test for android applications against code obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 347–359.

Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 860–871.

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC '17)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/3134600.3134620

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 327–337. https://doi.org/10.1145/3180155.3180178

Project Zero. 2021. *Déjà vu-lnerability.* https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html

Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. 2024. PS3: Precise Patch Presence Test based on Semantic Symbolic Signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. 2022. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 190–201.

Received 2024-09-12; accepted 2025-04-01