Recurring Vulnerability Detection: How Far Are We?

YIHENG CAO*, Fudan University, China
SUSHENG WU*, Fudan University, China
RUISI WANG*, Fudan University, China
BIHUAN CHEN*†, Fudan University, China
YIHENG HUANG*, Fudan University, China
CHENHAO LU*, Fudan University, China
ZHUOTONG ZHOU*, Fudan University, China
XIN PENG*, Fudan University, China

With the rapid development of open-source software, code reuse has become a common practice to accelerate development. However, it leads to inheritance from the original vulnerability, which recurs at the reusing projects, known as recurring vulnerabilities (RVs). Traditional general-purpose vulnerability detection approaches struggle with scalability and adaptability, while learning-based approaches are often constrained by limited training datasets and are less effective against unseen vulnerabilities. Though specific recurring vulnerability detection (RVD) approaches have been proposed, their effectiveness across various RV characteristics remains unclear.

In this paper, we conduct a large-scale empirical study using a newly constructed RV dataset containing 4,569 RVs, achieving a 953% expansion over prior RV datasets. Our study analyzes the characteristics of RVs, evaluates the effectiveness of the state-of-the-art RVD approaches, and investigates the root causes of false positives and false negatives, yielding key insights. Inspired by these insights, we design AntMan, a novel RVD approach that identifies both explicit and implicit call relations with modified functions, then employs inter-procedural taint analysis and intra-procedural dependency slicing within those functions to generate comprehensive signatures, and finally incorporates a flexible matching to detect RVs. Our evaluation has shown the effectiveness, generality and practical usefulness in RVD. AntMan has detected 4,593 RVs, with 307 confirmed by developers, and identified 73 new 0-day vulnerabilities across 15 projects, receiving 5 CVE identifiers.

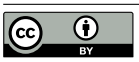
CCS Concepts: • **Security and privacy**; • **Human-centered computing** → **Open source software**;

Additional Key Words and Phrases: Empirical Study, Recurring Vulnerability, Open Source Software

ACM Reference Format:
Yiheng Cao, Susheng Wu, Ruisi Wang, Bihuan Chen, Yiheng Huang, Chenhao Lu, Zhuotong Zhou, and Xin Peng. 2025. Recurring Vulnerability Detection: How Far Are We?. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA026 (July 2025), 23 pages. https://doi.org/10.1145/3728901

*Y. Cao, S. Wu, R. Wang, B. Chen, Y. Huang, C. Lu, Z. Zhou and X. Peng are also with Shanghai Key Laboratory of Data Science, Fudan University, China.
†Bihuan Chen is the corresponding author.

Authors' Contact Information: Yiheng Cao, School of Computer Science, Fudan University, Shanghai, China, caoyh23@m.fudan.edu.cn; Susheng Wu, School of Computer Science, Fudan University, Shanghai, China, scwu24@m.fudan.edu.cn; Ruisi Wang, School of Computer Science, Fudan University, Shanghai, China, rswang23@m.fudan.edu.cn; Bihuan Chen, School of Computer Science, Fudan University, Shanghai, China, bhchen@fudan.edu.cn; Yiheng Huang, School of Computer Science, Fudan University, Shanghai, China, yihenghuang23@m.fudan.edu.cn; Chenhao Lu, School of Computer Science, Fudan University, Shanghai, China, chlu22@m.fudan.edu.cn; Zhuotong Zhou, School of Computer Science, Fudan University, Shanghai, China, zhouzt23@m.fudan.edu.cn; Xin Peng, School of Computer Science, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2025/7-ARTISSTA026
https://doi.org/10.1145/3728901

1 Introduction

With the rapid development of open-source software, reusing code has become a common practice to accelerate software development. However, if reused code contains vulnerabilities, those vulnerabilities can be inherited, resulting in recurring vulnerabilities (RVs) to downstream projects. These RVs, sharing similar characteristics with the original vulnerabilities, commonly exist in real-world projects [9, 24, 50], posing significant security risks that require timely detection.

Current general-purpose vulnerability detection approaches include traditional approaches such as static analysis (e.g., [2, 7, 11, 16, 37, 50]), symbolic execution (e.g., [6, 27, 33, 40, 42]) and fuzzing (e.g., [3, 4, 20, 38, 39, 52]), as well as learning-based approaches (e.g., [22, 23, 29, 32, 51]). However, they struggle with recurring vulnerability detection (RVD). Traditional vulnerability detection approaches face scalability challenges. Static analysis relies on pre-defined rules, symbolic execution suffers from path explosion, and fuzzing depends on compilation and input coverage, which can hinder broader applicability. Although learning-based approaches offer greater flexibility, they depend on the quantity and quality of training datasets, restricting their effectiveness against previously unseen vulnerabilities. To address this gap, several specific RVD approaches have been proposed, from early approaches like ReDeBug [15], VUDDY [18] to more recent ones like MVP [49], MOVERY [45], Tracer [17], V1scan [44], FIRE [10] and VMUD [14]. These approaches extract lexical, syntactic, or semantic features from known vulnerability and search for similar patterns to detect RVs.

Empirical Study. Despite advances in RVD approaches, their effectiveness across various RV characteristics remains unclear. To bridge this gap, we conduct the first large-scale empirical study with three research questions to analyze the characteristics of RVs, evaluate the effectiveness of existing RVD approaches, and identify the root causes of false positives (FPs) and false negatives (FNs).

- **RQ1 Characteristic Analysis of RVs.** What are the characteristics of RVs?
- **RQ2 Effectiveness Evaluation of RVD.** How is the effectiveness of state-of-the-art RVD?
- **RQ3 FP/FN Analysis of RVD.** What are causes of false positives and false negatives of RVD?

To ensure a comprehensive and robust analysis, we construct the largest known ground truth dataset of RVs, comprising 4,569 RVs with an increase of 953% over the largest existing dataset constructed by MOVERY [45], which only contained 434 RVs. Our dataset construction and verification involves approximately 2,000 human hours by three security experts.

In **RQ1**, our analysis reveals that only 28% of the RVs are identical with the original vulnerabilities (Type-I clones). In contrast, 63% are quite different (Type-III clones) due to syntactic or semantic changes in both version evolution and code reuse. Further, there are 36 detected vulnerabilities that have significant discrepancy with the original functionality and code semantic, which is regarded as 0-day, indicating RVD approaches also have the capability to detect 0-day vulnerabilities. In **RQ2**, all RVD approaches suffer a significant performance drop from 0.08 to 0.34 in F1-score when detecting Type-III vulnerabilities compared to Type-I vulnerabilities. In **RQ3**, we deconstruct the existing RVD approaches and analyze their strategies across different stages to identify the root causes of FPs and FNs. Finally, we summarize three key insights for a better RVD approach, i.e., **broad context awareness (I-1)**, **fine-grained signature (I-2)**, and **flexible matching (I-3)**.

Our Approach. We propose a novel approach, AntMan, to detect RVs more effectively. To achieve **I-1**, AntMan begins by constructing a normalized call graph for both the original repository and the target repository. It first performs a comprehensive normalization including macro expansion, control block standardization, assignment statement deconstruction and permutation, and operator rewriting. After normalization, AntMan generates normalized call graphs according to the patch of the original vulnerability, allowing to trace broad context for vulnerability spread. To achieve **I-2**, AntMan traces the sensitive variables across inter-procedural taint analysis, as well as intra-procedural dependency slicing to extract fine-grained vulnerability path. These paths are further

used to generate signatures, representing as comprehensive inter-procedural code property graph to provide a detailed understanding of RVs. To achieve **I-3**, AntMan applies a graph matching technique enhanced by a code language model, allowing for adaptive and nuanced detection of RVs and addressing the rigidity of traditional matching strategies in existing RVD approaches.

Evaluation. We conducted extensive experiments to assess AntMan's effectiveness, ablation, generality, and usefulness. AntMan achieved a precision of 0.84 and a recall of 0.85 on the ground truth dataset, surpassing the best RVD approach by 27% in precision and 63% in recall. Our ablation and threshold sensitivity analyses confirm the contributions of the improved strategies in AntMan to its overall effectiveness. We also constructed a generality datasets of RVs, where AntMan outperformed both RVD and learning-based vulnerability detection approaches. Our experiments also showed that AntMan excelled in identifying 0-day vulnerabilities, covering 73 (89%) 0-day vulnerabilities, which is 13% higher than other approaches. For usefulness, AntMan successfully detected 4,593 RVs, with 307 confirmed by developers, and uncovered 73 new 0-day vulnerabilities across 15 projects, receiving 5 CVE identifiers.

Contribution. In summary, this work makes the following main contributions.

- *Large-Scale RV Dataset.* We constructed the largest ground truth dataset of RVs to date, comprising 4,569 RVs with an increase of at least 953% over all previous datasets.
- *First Thorough Empirical Study.* We conducted the first large-scale empirical study on RVs, analyzing their characteristics, evaluating state-of-the-art RVD approaches, and identifying the root causes of false positives and false negatives.
- *Novel RVD Approach.* Guided by our empirical findings, we proposed a novel approach, AntMan, that addresses existing limitations by incorporating broad context analysis, fine-grained signature extraction, and flexible matching mechanism.
- *Comprehensive Evaluation.* AntMan outperformed existing RVD approaches by at least 27% in precision and 63% in recall. In the generality dataset, AntMan surpassed both RVD and learning-based approaches by at least 27% in precision and 6% in recall. AntMan also had the capability in 0-day vulnerability detection, outperforming the best state-of-the-art by 13%. AntMan detected 4,520 1/N-day vulnerabilities with 240 confirmed by developers, and discovered 73 0-day vulnerabilities with 67 confirmed by developers and 5 CVE identifiers assigned.

2 Related Work

General-Purpose Vulnerability Detection. Many approaches including traditional and learning-based approaches are proposed to detect general vulnerabilities. Traditional approaches such as static analysis, symbolic execution, and fuzzing, encounter significant limitations in RVD. In particular, static analysis based approaches [7, 37, 50] depend heavily on pre-defined rules, limiting adaptability to new vulnerabilities. Symbolic execution [27, 33, 36, 42] suffers from path explosion, and fuzzing [3, 4, 20, 38, 39, 52] requires compilation, which restricts their scalability and efficiency. In contrast, learning-based approaches [1, 22, 32, 43, 51] offer more flexible detection by learning patterns from the vulnerabilities in training dataset. However, their effectiveness largely depends on the quality and the quantity of training dataset. In summary, they are not applicable to RVD.

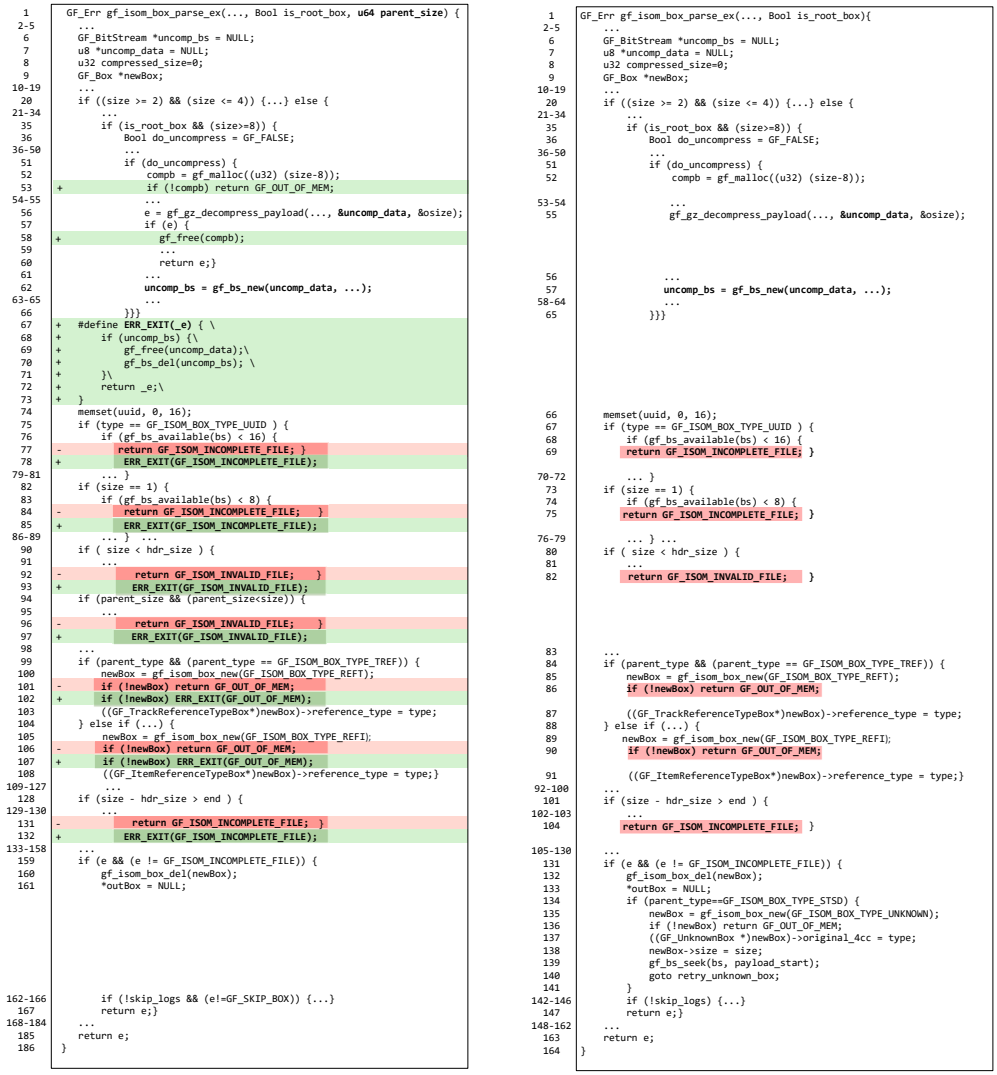
Recurring Vulnerability Detection. Several RVD approaches have been proposed [10, 15, 17, 18, 44, 45, 49]. ReDeBug [15] and VUDDY [18] are the pioneer works. While ReDeBug [15] uses hunk-level hard matching, VUDDY [18] uses function-level hard matching. VUDDY [18] first extracts all functions involving deleted lines from the patch and all functions from the target repository. Then, it normalizes each function by removing comments and abstracting code elements (e.g., local variables) into placeholders before computing their MD5 hashes. Matching hashes indicates potential RVs. This approach is efficient for RVD, but its coarse-grained matching limits the effectiveness in identifying variations with substantial modifications. To address this limitation, MVP

[49] uses program slicing to extract modified statements and their corresponding dependencies from patches, while simultaneously extracting all statements and dependencies from each function in the target repository. After normalizing and abstracting these statements, it computes their MD5 hashes, with matching hashes suggesting potential RVs. MOVERY [45] extracts the modified statements from a patch and confirms their relevance by verifying that these statements appear in the earliest known vulnerable version. It then retrieves all statements from the corresponding functions, which are identified by software composition analysis in the target repository. After normalization and abstraction, it performs exact string matching between the patched statements and those in the target code. A match indicates a potential RV. However, the coarse-grained slicing strategy in MVP and MOVERY often includes irrelevant statements, and their inaccurate abstraction obscures vulnerability characteristics. Besides, MOVERY incorporates characteristics from the oldest version of vulnerabilities revealed in CPE, which is often incomplete or inaccurate [46, 48]. Tracer [17] focuses on memory-related RVs by taint analysis and requires specific compilation conditions, limiting its generality. V1scan [44] employs a dual-detection methodology that integrates version-based and code-based detectors to identify potential RVs. An alert from either detector is sufficient to flag an RV. For version-based detector, it extracts signatures from the entire codebases of both the original and the target repositories. Using software composition analysis, it checks if the target repository is more similar to the vulnerable version than the fixed one. If yes, it flags a potential RV. For code-based detector, it extracts functions from the vulnerable version and all functions from the target repository, normalizes them, and uses locality-sensitive hashing to match. For matched functions, it extracts the modified statements from the patch and the statements from the target function, normalizes them, and performs exact string matching on them. A match indicates a potential RV. This approach is efficient for RVD, but its function-level matching and exact string matching also struggle to detect variations involving extensive modifications. VMUD [14] specifically focuses on detecting RVs with multiple fixing functions by critical function selection and contextual semantic equivalent statement matching. FIRE [10] is a scalable approach for detecting RVs. It extracts tokens from the patch's modified functions and all functions in the target repository, then uses Jaccard similarity to quickly identify related candidates. It then refines this selection by extracting tokens from the functions' ASTs and reapplying Jaccard similarity to capture structural similarities. For the remaining functions, it normalizes and exactly matches modified statements from the patch with those in candidate functions. When matches occur, FIRE constructs and normalizes taint paths that capture data flow, comparing them using cosine similarity. A high similarity indicates a potential RV.

3 Motivation Example

We use a vulnerability CVE-2022-46489 [25] to illustrate the limitations of existing RVD approaches (i.e., VUDDY [18], MVP [49], MOVERY [45], V1scan [44], and FIRE [10]). As shown in Figure 1(a), a memory leak vulnerability is introduced in function `gf_isom_box_parse_ex` in project *gpac*. It arises when memory allocated for `uncomp_bs` (Line 62) remains unreleased across multiple function exit points (Lines 77, 84, 92, 96, 101, 106, and 131), leading to resource exhaustion. Its patch [12] introduces an `ERR_EXIT` macro (Lines 67-73), ensuring memory cleanup before any function return. An RV is observed in version 1.0.1 of *gpac*, as shown in Figure 1(b), where the function contains similar vulnerable code with Figure 1(a) but includes additional vulnerability-irrelevant code.

VUDDY, MVP, MOVERY and V1Scan fail to identify this RV due to their improper handling of irrelevant code. VUDDY and V1Scan, relying on function-level code matching between vulnerable and target functions, erroneously include irrelevant code lines (Lines 131-147) in their target signature, resulting in failed RV detection. MVP and MOVERY first incorporate the vulnerability-irrelevant



(a) Patch for CVE-2022-46489 (b) An RV of CVE-2022-46489

Fig. 1. Patch *44e8616e* for CVE-2022-46489 in *gpac* and Vulnerable Version *1.0.1* of *gpac*

variable newBox (Line 86) into their target signature. Then through control and data dependency relations with newBox, they extract the irrelevant code (Lines 135-138), resulting in failed RV detection. FIRE detects RVs by extracting taint paths from identifier nodes to call nodes. In this case, it successfully identifies the RV because Lines 131-147 contain no function calls involving newBox. However, it remains prone to including irrelevant identifiers, leading to potential false positives. This example motivates the need of our empirical study to comprehensively uncover their limitations.

4 Empirical Study

To better understand RV and RVD approaches, we conducted the first large-scale empirical study of analyzing RV characteristics and evaluating the effectiveness of existing RVD approaches to gain deeper insights into the limitations of current RVD approaches.

4.1 Study Design

4.1.1 RVD Approach Selection. To select RVD approaches as our baselines, we conducted a literature review starting with VMUD [14], the most recent RVD work. We applied literature snowballing while focusing specifically on the latest RVD approaches since 2014. This process led us to seven source code-based RVD approaches, i.e., VUDDY [18], MVP [49], MOVERY [45], TRACER [17], V1SCAN [44], FIRE [10] and VMUD [14]. We excluded VMUD because of its limited scope in detecting RVs with multiple fixing functions and TRACER along with all binary-based RVD approaches due to their reliance on compiling, which limits the applicability for large-scale RV detection. Finally, we adopted the default configurations from the original publications of the five selected RVD approaches.

4.1.2 Ground Truth Dataset Construction. Since no open-source dataset on RVs was available, we constructed the largest known ground truth dataset of RVs, enabling rigorous empirical analysis. Each sample in our dataset is denoted as a six-tuple $\langle cveid, pat, repo_{\text{o}}, repo_{\text{t}}, if_{\text{rv}}, F_{\text{rv}} \rangle$, where $cveid$ is the CVE identifier of the original vulnerability; pat is a patch commit from the original repository $repo_{\text{o}}$ where the original vulnerability locates; $repo_{\text{t}}$ is the target repository to be detected with its specific version; if_{rv} is a boolean indicating whether $repo_{\text{t}}$ contains an RV; and F_{rv} is the detected vulnerable function set by RVD approaches and verified by experts. Unlike existing RVD approaches that took baseline-detected samples as ground truth [10, 18, 44, 45, 49], we employed a three-step process integrated by additional human effort to mitigate baseline-induced false positives and negatives.

Step 1: Vulnerability and Patch Collection. We first selected the original vulnerability $cveid$ and its patch commit pat of $repo_{\text{o}}$ from the NVD Data Feeds [26]. After filtering for C/C++ vulnerabilities from 1 January 2020 to 1 January 2024, we collected a total of 2,115 vulnerabilities with their associated patches. Then, we further excluded patches that modified only global declarations (e.g., macros and structures), C/C++ configuration files, or non-C/C++ files. This restricted our selection to a final dataset of 2,088 vulnerabilities with their associated patches.

Step 2: Target Repository Collection. To ensure a diverse set of RVs, we targeted high-profile GitHub repositories, selected based on star counts, while excluding archived or outdated projects. By August 2024, we gathered the top 600 active C/C++ repositories. We then gathered all the released versions (i.e., 12,088) of the repositories. Given the large number of versions, we opted for sampling to reduce approach runtime and manual effort. To achieve this, we first sorted versions of a repository in chronological order and divided the versions by season. We then selected the first version released within each season to represent the evolution of code over specific periods, discarding all other versions from that period. If no version was available for a particular season, it was simply excluded. This process resulted in 3,873 distinct repositories with version tags, and each is denoted as $repo_{\text{t}}$.

Step 3: RV Detection and Confirmation. To maximize detection coverage and mitigate single-tool bias, we ran all selected five RVD approaches to identify RVs in each $repo_{\text{t}}$ with each patch pat as input. This process generated samples that were detected by at least one RVD approach. False alarms were classified by human experts verification. If a detected sample was confirmed as an RV, it was marked as a positive sample, with if_{rv} set to `True` and the corresponding vulnerable functions listed in F_{rv}. If a detected sample was not an RV, it was classified as a negative sample, with if_{rv} set to `False` and F_{rv} left empty. This confirmation was conducted by two experienced security professionals, each with over three years of experience. Any disagreements were resolved by a third expert, ensuring consensus. This process identified 3,834 positive samples and 4,469 negative samples. Moreover, as RVs can persist across multiple versions, the experts extended their manual analysis by recursively checking earlier and later versions of $repo_{\text{t}}$ where no sample was identified by RVD approaches, continuing until no further vulnerable versions were found. This thorough examination mitigates the risk of missing RVs and avoids potential false negatives, providing a more complete set of RVs. Any discrepancies were resolved through consultation with a third expert to maintain

Table 1. Distribution w.r.t. Similarity Types, Patch Scopes, and ∗-Day Vulnerability Types

	Original Repositories			Transferred Repositories			Original Repositories		Transferred Repositories	
	Type-I	Type-II	Type-III	Type-I	Type-II	Type-III	1/N-day	0-day	1/N-day	0-day
$\mathcal{S}$	818 (34%)	262 (11%)	1,295 (55%)	101 (39%)	29 (12%)	126 (49%)	2,365 (99.6%)	10 (0.4%)	244 (95.3%)	12 (4.7%)
$\mathcal{M}$	300 (19%)	81 (5%)	1,221 (76%)	50 (15%)	34 (10%)	252 (75%)	1,597 (99.7%)	5 (0.3%)	327 (97.3%)	9 (2.7%)
$\mathcal{S} \cup \mathcal{M}$	1,118 (28%)	343 (9%)	2,516 (63%)	151 (26%)	63 (11%)	378 (63%)	3,962 (99.6%)	15 (0.4%)	571 (96.5%)	21 (3.5%)

accuracy and consistency. Ultimately, we gathered 4,569 positive samples across 1,300 $repo_t$ and 4,469 negative samples across 1,234 $repo_t$, costing 2,000 human hours. We achieved a Cohen's Kappa coefficient of 0.934 for sample confirmation and 0.936 for sample expansion.

4.1.3 Metrics. We measured RVD approaches using true positives (TP), false positives (FP), false negatives (FN), precision (Pre.), recall (Rec.), and F1-score (F1.).

4.2 Characteristic Analysis of RVs (RQ1)

We focus on three characteristics of RVs, *similarity types*, *patch scopes*, and *∗-day vulnerability types*. We analyzed the characteristics of RVs in two contexts: (1) RVs recurring within the same repository (where $repo_t$ and $repo_o$ are the same, referred to as the "original repository"), and (2) RVs recurring in different repositories (where $repo_t$ and $repo_o$ are different, referred to as the "transferred repository"). This distinction helps us understand how RVs distribute in different repositories.

Setup of RV Similarity Type Analysis. To analyze the impact of code duplication and variation on RVD, we categorized RVs into three distinct similarity types based on well-established definitions of code clones [41], i.e., **Type-I (Exact Clones)**: if all functions in F_{rv} are identical with the corresponding functions in *pat*, Type-I is detected; **Type-II (Renamed Clones)**: if all functions in F_{rv} are identical with the corresponding functions in *pat* after identifier (i.e., variables, function calls, type declarations, and string literals) renaming, Type-II is detected; and **Type-III (Semantic Clones)**: if there exists at least one function in F_{rv} that differs from its corresponding function in *pat* after identifier renaming, Type-III is detected.

Setup of Patch Scope Analysis. Since current RVD approaches primarily rely on analyzing the modified functions within a patch (with V1SCAN also considering changes to global identifiers), it is essential to understand how different types of function modifications impact RVD. Therefore, we categorized patches into two distinct groups based on their function modification scope: (1) patches with single-function modifications (pat_s) and (2) patches with multi-function modifications (pat_m). This classification allows us to distinguish between two RV detection scenarios, $\mathcal{S}$ for pat_s, where changes are isolated to a single function, and $\mathcal{M}$ for pat_m, where changes span multiple functions.

Setup of ∗-day Vulnerability Type Analysis. RVD approaches are primarily designed to detect 1/N-day vulnerabilities. To assess their capability of detecting 0-day vulnerabilities, we focused on scenarios where the functionality and semantics of functions in F_{rv} significantly differ from those in *pat*, sharing only the underlying vulnerability logic without direct code reuse. To distinguish 0-day vulnerabilities, we conducted a manual review. Finally, we tagged 36 0-day vulnerabilities, and reported them to repository owners with 21 confirmed and 15 in progress.

RV Distribution w.r.t. Similarity Types. As shown in Table 1, there is a consistent distribution of Type-I, Type-II, and Type-III clones across both original and transferred repositories. Type-III clones were the most prevalent, accounting for 63% in both original and transferred repositories. This suggests that significant syntactic or semantic changes are common among RVs. Type-I clones were also present in substantial numbers, comprising 28% in original repositories, and 26% in transferred repositories, reflecting frequent direct reuse of vulnerable code. In contrast, Type-II clones were relatively rare, making up 9% in original repositories, and 11% in transferred repositories.

Finding 1: The distribution of similarity types is similar across original and transferred repositories, with Type-III clones being the most prevalent, accounting for 63%.

RV Distribution w.r.t. Patch Scopes. 2,375 (60%) and 1,602 (40%) RVs belonged to $\mathcal{S}$ and $\mathcal{M}$ in original repositories, respectively. 256 (43%) and 336 (57%) RVs belonged to $\mathcal{S}$ and $\mathcal{M}$ in transferred repositories, respectively. Specifically, for $\mathcal{S}$, Type-I and Type-III clones were prominent, accounting for 34% and 55% in original repositories, and 39% and 49% in transferred ones, respectively. Type-II clones remained relatively rare. In contrast, for $\mathcal{M}$, Type-III clones were dominant, making up 76% and 75% in original and transferred repositories, respectively. This trend underscores that $\mathcal{M}$ is often extensive, resulting in significant modifications threatening RVD approaches.

Finding 2: RVs whose patches involve multi-function modifications (i.e., $\mathcal{M}$) are common, accounting for 40% and 57% in original and transferred repositories, respectively. Type-I and Type-III clones are prominent in $\mathcal{S}$, while Type-III clones are dominant in $\mathcal{M}$.

RV Distribution w.r.t. $*$-Day Vulnerability Types. As shown in Table 1, in original repositories, the majority (99.6%) of RVs were classified as 1/N-day vulnerabilities, indicating a high prevalence of code reuse within the same repository. This overall trend remains consistent across $\mathcal{M}$ and $\mathcal{S}$. Further analysis on the 15 0-day vulnerabilities revealed that 7 of them consistently appeared within the same file as the original vulnerabilities, while the remaining 8 were located in different files within the same folder, suggesting a proximity-based occurrence of 0-day vulnerabilities. In transferred repositories, the distribution shifts slightly but still maintains a dominant presence of 1/N-day vulnerabilities across $\mathcal{M}$ and $\mathcal{S}$. It is worth mentioning that while 1/N-day vulnerabilities are prevalent in both original and transferred repositories, the likelihood of encountering a 0-day vulnerability in transferred repositories is significantly higher, with an increase of 842% compared to original repositories. It underscores the potential risk of 0-day vulnerabilities in transferred code, where variations in code logic can lead to unique vulnerabilities that are less common in original code.

Finding 3: Although 1/N-day vulnerabilities are dominant across both original and transferred repositories, some RVD approaches demonstrate the capability to detect 0-day vulnerabilities within both contexts, highlighting the potential applicability of RVD approaches.

4.3 Effectiveness Evaluation of RVD (RQ2)

Detection Criteria Setup. In current RVD approaches, only V1SCAN is designed to support multi-function modification scenarios, where an RV is detected if at least one modified function from pat_{m} matches in $repo_{\mathrm{t}}$. The other approaches (VUDDY, MVP, MOVERY and FIRE) are designed primarily for single-function modification scenarios. As our dataset includes both single-function and multi-function RVs, we extended the detection criteria of these four approaches to align with V1SCAN, allowing them to detect RVs based on matching at least one modified function from pat_{m}.

Effectiveness w.r.t. Similarity Types. As shown in Table 2, for Type-I clones, VUDDY achieves the highest recall (0.85) but a lower precision (0.67), resulting in the highest F1-score of 0.75. Conversely, MVP prioritizes precision (0.74) over recall (0.52), and has the lowest F1-score of 0.61. For Type-II clones, VUDDY maintains the balance with a precision of 0.72 and a recall of 0.66, yielding the highest F1-score of 0.69. However, MOVERY, with the lowest F1-score of only 0.39, struggles significantly due to its low precision (0.29) and moderate recall (0.57). For the more complex Type-III vulnerabilities, all approaches suffer a steep decline. FIRE performs the best with an F1-score of 0.55.

Table 2. Effectiveness of the State-of-the-Art RVD Approaches

		VUDDY			MVP			MOVERY			V1SCAN			FIRE		
		$\mathcal{S}$	$\mathcal{M}$	$\mathcal{S}\cup\mathcal{M}$	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{S}\cup\mathcal{M}$	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{S}\cup\mathcal{M}$	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{S}\cup\mathcal{M}$	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{S}\cup\mathcal{M}$
Type-I	**TP**	761	319	1,080	447	215	662	646	295	941	598	258	856	559	211	770
	FP	325	199	524	144	88	232	407	237	644	189	129	318	280	142	422
	FN	158	31	189	472	135	607	273	55	328	321	92	413	360	139	499
	Pre.	0.70	0.62	0.67	**0.76**	**0.71**	**0.74**	0.61	0.55	0.59	**0.76**	0.67	0.73	0.67	0.60	0.65
	Rec.	**0.83**	**0.91**	**0.85**	0.49	0.61	0.52	0.70	0.84	0.74	0.65	0.74	0.67	0.61	0.60	0.61
	F1.	**0.76**	**0.74**	**0.75**	0.59	0.66	0.61	0.66	0.67	0.66	0.70	0.70	0.70	0.64	0.60	0.63
Type-II	**TP**	172	96	268	81	61	142	147	83	230	97	58	155	127	68	195
	FP	29	77	106	2	19	21	414	144	558	8	35	43	24	36	60
	FN	119	19	138	210	54	264	144	32	176	194	57	251	164	47	211
	Pre.	0.86	0.55	0.72	**0.98**	**0.76**	**0.87**	0.26	0.37	0.29	0.92	0.62	0.78	0.84	0.65	0.76
	Rec.	**0.59**	**0.83**	**0.66**	0.28	0.53	0.35	0.51	0.72	0.57	0.33	0.50	0.38	0.44	0.59	0.48
	F1.	**0.70**	**0.67**	**0.69**	0.43	0.63	0.50	0.35	0.49	0.39	0.49	0.56	0.51	0.57	0.62	0.59
Type-III	**TP**	0	817	817	434	849	1,283	697	1,159	1,856	105	654	759	547	845	1,392
	FP	0	285	285	275	595	870	1,276	1,240	2,516	24	198	222	310	441	751
	FN	1,421	656	2,077	987	624	1,611	724	314	1,038	1,316	819	2,135	874	628	1,502
	Pre.	0.0	0.74	0.74	0.61	0.59	0.60	0.35	0.48	0.42	**0.81**	**0.77**	**0.77**	0.64	0.66	0.65
	Rec.	0.0	0.55	0.28	0.31	0.58	0.44	**0.49**	**0.79**	**0.64**	0.07	0.44	0.26	0.38	0.57	0.48
	F1.	0.0	0.63	0.41	0.41	0.58	0.51	0.41	0.60	0.51	0.14	0.56	0.39	**0.48**	**0.61**	**0.55**
All	**TP**	933	1,232	2,165	962	1,125	2,087	1,490	1,537	3,027	800	970	1,770	1,233	1,124	2,357
	FP	354	561	915	421	702	1,123	2,097	1,621	3,718	221	362	583	614	619	1,233
	FN	1,698	706	2,404	1,669	813	2,482	1,141	401	1,542	1,831	968	2,799	1,398	814	2,212
	Pre.	0.72	0.69	0.70	0.70	0.62	0.65	0.42	0.49	0.45	**0.78**	**0.73**	**0.75**	0.67	0.64	0.66
	Rec.	0.35	0.64	0.47	0.37	0.58	0.46	**0.57**	**0.79**	**0.66**	0.30	0.50	0.39	0.47	0.58	0.52
	F1.	0.48	0.66	0.57	0.48	0.60	0.54	0.48	0.60	0.54	0.44	0.59	0.51	**0.55**	**0.61**	**0.58**

V1SCAN has the highest precision of 0.77 but the lowest recall of 0.26, which leads to the lowest F1-score of 0.39. VUDDY performs the best in Type-I and Type-II vulnerabilities, but drops to the second worst in Type-III vulnerabilities, achieving decent precision (0.74) but very low recall (0.28).

Finding 4: VUDDY performs the best in Type-I and Type-II vulnerability detection, but suffers a significant drop in Type-III vulnerability detection. FIRE demonstrates relatively consistent performance across all similarity types, achieving an F1-score around 0.60. MVP, MOVERY, and V1SCAN show well performance in Type-I but suffer a drop in Type-II and Type-III.

Effectiveness w.r.t. Patch Scopes. As shown in Table 2, the F1-score on $\mathcal{S}$ and $\mathcal{M}$ are very close in Type-I and Type-II. In contrast, in Type-III, the F1-score on $\mathcal{M}$ is higher than that on $\mathcal{S}$ across all the approaches by 0.13 to 0.63. The underlying reason is that according to the detection criteria, RVs in $\mathcal{M}$ are detected if only one of the modified multiple functions is matched for $repo_{\text{o}}$ and $repo_{\text{t}}$, which latently increases the matching probability and increases the recall of all the RVD approaches. On the other hand, not all modified functions in pat_m are directly related to vulnerabilities. Matching such unrelated functions can lead to a drop in precision. An exception is MOVERY, which shows an increase of 0.07 in precision for $\mathcal{M}$. This anomaly may be attributed to the inherent randomness in MOVERY that affects its low precision performance.

Finding 5: The effectiveness of all RVD approaches consistently declines as the complexity of vulnerabilities increases, particularly from Type-I to Type-II and Type-III vulnerabilities. The F1-score on $\mathcal{S}$ and $\mathcal{M}$ are very close in Type-I and Type-II, but in Type-III, a higher recall but a lower precision are achieved on $\mathcal{M}$ than on $\mathcal{S}$ due to the single-function matching criteria.

4.4 FP/FN Analysis of RVD (RQ3)

Sampling Setup. We began by sampling FPs and FNs for each RVD approach to reduce manual cost, resulting in 173, 814, 427, 208, 299 FPs and 881, 1,180, 314, 1,323, 879 FNs for the five approaches respectively. Sampling was performed at a 99% confidence level with a 3% confidence interval. Two authors independently selected 100 FPs and 100 FNs from each approach for a pilot study. Following an open coding procedure [30], they identified the approach **stages** and atomic approach

Table 3. Taxonomy of Stages and Strategies Used in the State-of-the-Art RVD Approaches

Stage	Granularity	Strategy	VUDDY	MVP	MOVERY	V1SCAN	FIRE
Stage 1	Component-Level	**S1-1:** extract signatures of the **entire codes** in $repo_o$ and $repo_t$				✔	
	Function-Level	**S1-2:** extract the **entire pre-patch functions** in $repo_o$ and all the functions in $repo_t$	✔			✔	
		S1-3: extract **function relations** in pat and in $repo_t$					
	Statement-Level	**S1-4:** extract **modified statements** in pat and **all the statements** in the target function		✔	✔	✔	✔
		S1-5: extract **modified statements with their dependency context** in pat verified in earliest vulnerable version, and **all the statements** in the target function			✔		
		S1-6: extract **modified statements with their dependency context** in pat and pair them with **dependency relation**, and extract **all the statements and their dependency pair** in the target function		✔			
		S1-7: extract **taint paths** for variables within pat and the target function					✔
	Token-Level	**S1-8:** extract tokens of **unstructured C/C++ keywords** from the functions in pat and $repo_t$					✔
		S1-9: extract tokens of **AST** from the functions in pat and $repo_t$					✔
Stage 2	Statement-Level	**S2-1:** normalize by removing **irrelevant syntactic structures** (e.g., comment)	✔	✔	✔	✔	✔
		S2-2: abstract **identifier** (e.g., local variables to LVAR)	✔	✔	✔		
Stage 3	Component-Level	**S3-1:** match by **SCA tools**				✔	
	Function-Level	**S3-2:** match by **MD5 hash**	✔				
		S3-3: match by **local sensitive hash**				✔	
	Statement-Level	**S3-4:** match by **MD5 hash or string exactly matching**		✔	✔	✔	✔
		S3-5: match by **cosine similarity**					✔
	Token-Level	**S3-6:** match by **Jaccard similarity**					✔

Table 4. The Top Three Strategies That Introduced the Most FPs and FNs

		$\mathcal{S}$		$\mathcal{M}$		$\mathcal{S} \cup \mathcal{M}$	
		Top 3 Strategies	Sum	Top 3 Strategies	Sum	Top 3 Strategies	Sum
Type-I	FP	S1-4 (35%) | S1-7 (19%) | S2-2 (13%)	67%	S1-7 (22%) | S1-3 (21%) | S2-1 (15%)	58%	S1-4 (21%) | S1-7 (21%) | S1-3 (16%)	58%
	FN	S3-4 (32%) | S1-4 (19%) | S2-2 (12%)	63%	S3-4 (34%) | S1-4 (16%) | S1-6 (10%)	60%	S3-4 (32%) | S1-4 (18%) | S3-2 (11%)	61%
Type-II	FP	S1-7 (36%) | S1-4 (36%) | S1-3 (28%)	100%	S1-3 (40%) | S2-2 (24%) | S1-4 (15%)	79%	S1-3 (39%) | S2-2 (21%) | S1-4 (18%)	78%
	FN	S3-4 (29%) | S3-2 (16%) | S1-4 (14%)	59%	S3-4 (31%) | S1-4 (16%) | S1-6 (10%)	57%	S3-4 (30%) | S3-2 (14%) | S1-4 (13%)	57%
Type-III	FP	S2-2 (38%) | S1-4 (31%) | S1-5 (12%)	81%	S2-2 (27%) | S1-4 (23%) | S1-6 (15%)	65%	S2-2 (30%) | S1-4 (25%) | S1-6 (12%)	67%
	FN	S3-2 (30%) | S3-3 (26%) | S3-4 (17%)	73%	S3-2 (25%) | S3-3 (21%) | S3-4 (20%)	66%	S3-2 (28%) | S3-3 (24%) | S3-4 (18%)	70%
All	FP	S1-4 (33%) | S2-2 (30%) | S1-7 (9%)	72%	S2-2 (24%) | S1-4 (21%) | S1-3 (14%)	59%	S2-2 (26%) | S1-4 (24%) | S1-3 (12%)	62%
	FN	S3-2 (25%) | S3-4 (21%) | S3-3 (21%)	67%	S3-2 (22%) | S3-4 (21%) | S3-3 (19%)	62%	S3-2 (24%) | S3-4 (21%) | S3-3 (20%)	65%

strategies where inaccuracy was introduced into each RVD approach. We determined the root cause in each strategy that could cause FPs and FNs. To ensure inter-rater reliability, Cohen's Kappa was calculated, yielding 0.937 for FPs and 0.949 for FNs. Discrepancies were resolved with the involvement of a third author during both the pilot and final labeling phases. After labeling, each FP and FN was paired with the associated strategy where the root cause was introduced. Since multiple approaches could employ the same strategy, duplicate FP-strategy and FN-strategy pairs were consolidated. This deduplication process led to a total of 1,456 FP-strategy pairs and 3,643 FN-strategy pairs. The subsequent analysis of **RQ3** is based on these deduplicated FP-strategy and FN-strategy pairs. This analysis involves about 600 human hours by three security experts.

Taxonomy of Stages and Strategies. We summarized three primary stages in current RVD approaches, as detailed in Table 3. Each stage adopts various strategies, which can introduce FPs/FNs.

(1) *Stage 1: Original & Target Signature Extraction.* This stage involves extracting signatures from the original and target repositories across multiple levels (e.g., component, function, statement).
(2) *Stage 2: Signature Generalization.* This stage normalizes and abstracts the extracted signatures.
(3) *Stage 3: RV Detection.* This stage matches the generalized signatures to detect RVs.

Table 4 provides a detailed breakdown of the primary strategies that lead to most FPs and FNs across different similarity types and patch scopes, highlighting recurring challenges in RVD.

False Positive Analysis. As shown in Table 4, our analysis uncovers the following three root causes in the corresponding strategies that are the most significant contributors to FPs, collectively accounting for 62% of the total FP-strategy pairs. These FPs primarily stem from the design limitations within these strategies, resulting in a high rate of misclassification.

- **Inaccurate Abstraction in S2-2.** Inaccurate abstraction is a major cause of false positives, accounting for approximately 26% of the FPs across VUDDY, MVP and MOVERY. This strategy excessively abstracts variables, function calls and strings, failing to preserve essential fine-grained contextual details and relationships between statements, which leads to false alarms. Besides, the global-insensitive nature of RVD approaches often results in inaccurate abstraction of macros into local variables, causing the loss of critical semantics and false alarms.
- **Modified Statements without Context in S1-4.** Except for VUDDY, all RVD approaches only consider modified statements in signature extraction, but do not consider unmodified statements that have control or data dependency on the modified statements, contributing to 24% of the FPs.
- **Missing Function Relations in S1-3.** None of the approaches effectively handle inter-procedural dependencies in multi-function patches (pat_{m}). Approaches that treat multi-function patches as isolated modifications often overlook these dependencies. Therefore, such missing function relations lead to false alarms, accounting for about 12% of the FPs across all RVD approaches.

Interestingly, statement-level strategies frequently rank among the top contributors to FPs in both single-function and multi-function modification scenarios. Notably, S1-6, which pairs statements based on dependencies, accounts for a significant share of FPs (i.e., 3% in single-function scenarios and 14% in multi-function scenarios). Similarly, S1-7, which extracts taint paths from vulnerable function variables, also contributes to FPs notably (i.e., 9% in single-function scenarios and 12% in multi-function scenarios). These strategies often falter due to inadequate consideration of fine-grained relationships or excessive slicing. For example, S1-4 and S1-5 extract modified statements or their context without addressing inter-dependencies, while S1-6 pairs statements after over-slicing, which can lead to unnecessary or irrelevant associations. Likewise, S1-7 extracts taint paths indiscriminately, including those unrelated to vulnerabilities, resulting in false matches.

False Negative Analysis. FNs primarily arise from issues in matching strategies, particularly at the function level (S3-2 and S3-3) and statement level (S3-4). Such strategies collectively introduce 65% of the total FN-strategy pairs, highlighting the limitations of current RVD approaches.

- **Coarse-Grained Signature Matching in S3-2 and S3-3.** Function-level matching introduces significant FNs, accounting for 44% of the FNs (24% from S3-2 and 20% from S3-3), which is used in VUDDY and V1SCAN. Such coarse-grained strategies often fail to detect subtle structural changes, particularly with Type-II and Type-III vulnerabilities, leading to missed RVs.
- **Inflexible Matching in S3-4.** Hard matching like MD5 hash matching or string exactly matching also contributes to a substantial portion of FNs, accounting for 21%, which is used in MVP, MOVERY, V1SCAN and FIRE. Such inflexibility prevents it from recognizing complex code changes, which is a major issue discussed in **RQ1** for Type-III vulnerabilities. This rigidity directly impacts recall, as seen in **RQ2** where approaches like VUDDY under-perform on Type-III vulnerabilities.

Beyond the inflexibility in S3-4, specific statement-level strategies (e.g., S1-4 and S1-6) and abstraction techniques (e.g., S2-2) are also among the top contributors to FNs across various similarity types and patch scopes. Inaccurate signature generation in Stage 1 often misguides matching strategies, particularly when hard matching, such as MD5 hashing and string exactly matching. Likewise, overly generalized signatures in Stage 2 can obscure sensitive vulnerability-related identifiers, further increasing FNs. These challenges in Stages 1 and 2 not only contribute to FPs but also indirectly lead to FNs in the final matching stages. This underscores the need for more flexible matching strategies and higher-quality signatures to improve overall detection performance.

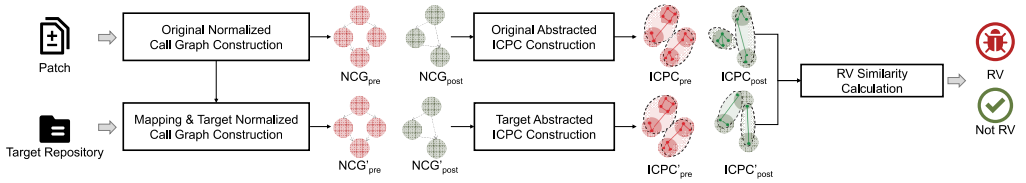


Fig. 2. Overview of AntMan

Finding 6: FPs and FNs are introduced in specific stages of existing RVD approaches. FPs are significantly influenced by inaccurate abstraction (in S2-2), insufficient handling of multi-function patches (in S1-3), and issues in statement-level signature generation (in S1-4, S1-5, S1-6 and S1-7). On the other hand, FNs are primarily caused by the inability to detect subtle variations at both the function level and statement level (in S3-2, S3-3 and S3-4).

4.5 Insights

With our FP and FN analysis, we obtain three key insights that can improve RVD's effectiveness.

- **I-1: Broad Context Awareness.** A context-aware approach is crucial for handling multi-function patches, especially those with global changes like macro adjustments. Expanding the detection process to capture broader context across functions and global identifiers can help mitigate limitations found in S1-3 and S2-2, improving precision and recall in RVD.
- **I-2: Fine-Grained Signature.** A fine-grained strategy in signature extraction, with a focus on refined statement-level analysis, can address issues related to inadequate extraction (as seen in S1-4 through S1-7) and overly broad abstractions (S2-2). This refined approach targets sensitive contextual details, improving precision and recall in RVD.
- **I-3: Flexible Matching.** The need for flexible matching mechanisms is evident in addressing the nuanced detection of RVs. Shifting from rigid, coarse-grained strategies towards more adaptable matching strategies can overcome challenges in S3-2, S3-4 and S3-3, improving recall in RVD.

5 Approach

Based on these insights, we propose a novel approach, AntMan, to detect RVs more effectively. To achieve **I-1**, AntMan begins by constructing a normalized call graph for both the original repository ($repo_o$) and the target repository ($repo_t$). It first performs a syntactic normalization, and then generates call graphs *NCG* based on the patch (*pat*) of the original vulnerability, allowing to trace broad context for vulnerability spread. To achieve **I-2**, AntMan traces the sensitive variables across inter-procedural taint analysis as well as intra-procedural dependency slicing to generate the RV signatures, representing as comprehensive inter-procedural code property clusters *ICPC*, to provide a detailed understanding of RVs. To achieve **I-3**, AntMan applies a graph matching technique enhanced by a code language model, allowing for adaptive and nuanced detection of RVs. To implement the three goals, AntMan works in the following five steps as shown in Figure 2.

(1) **Original Normalized Call Graph Construction.** Taking *pat* as an input, AntMan identifies the vulnerable version and the fixed version of $repo_o$, denoted as $repo_{pre}$ and $repo_{post}$. Then, it normalizes macros, control blocks, statements and operators of $repo_{pre}$ (resp. $repo_{post}$). Based on the normalized repositories, AntMan constructs normalized call graph (*NCG*) for $repo_{pre}$ (resp. $repo_{post}$) based on *pat*, denoted as NCG_{pre} (resp. NCG_{post}).

(2) **Original Abstracted ICPC Construction.** AntMan compares the functions in NCG_{pre} and NCG_{post} to generate a normalized patch pat_{norm}. Starting from modified functions in pat_{norm}, it performs fine-grained inter-procedural taint path and intra-procedural dependency path

extraction on NCG_{pre} (resp. NCG_{post}), generating vulnerable signature and fixed signature after type-sensitive abstraction, denoted as $ICPC_{\text{pre}}$ (resp. $ICPC_{\text{post}}$).

(3) **Mapping & Target Normalized Call Graph Construction.** AntMan maps functions, modified statements, and variables from NCG_{pre} (resp. NCG_{post}) to $repo_{\text{t}}$. Given mapped functions, it constructs the potential vulnerable (resp. fixed) call graphs, denoted as NCG'_{pre} (resp. NCG'_{post}).

(4) **Target Abstracted ICPC Construction.** Given mapped statements and variables, AntMan constructs the inter-procedural taint path and intra-procedural dependency path, and then constructs the potential vulnerable (resp. fixed) signature, denoted as $ICPC'_{\text{pre}}$ (resp. $ICPC'_{\text{post}}$).

(5) **RV Similarity Calculation.** AntMan vectorizes each statement and edge within $ICPC$, assigning weights according to their proximity to modified statements in $repo_{\text{o}}$ and the corresponding mapped modified statements in $repo_{\text{t}}$. It then calculates the similarity between $ICPC_{\text{pre}}$ and $ICPC'_{\text{pre}}$ as well as between $ICPC_{\text{post}}$ and $ICPC'_{\text{post}}$ to identify the matched vulnerable clusters and the matched fixed clusters. If the proportion of matched vulnerable clusters exceeds a threshold while the proportion of matched fixed clusters remains below a threshold, $repo_{\text{t}}$ is identified as vulnerable and a potential RV is detected.

5.1 Original Normalized Call Graph Construction

Normalization standardizes code syntax and semantics, which can recover the hidden vulnerability-related elements while reducing the impact of stylistic or minor structural differences, ensuring that only substantive changes are highlighted. Besides, function calls reveal critical dependencies, allowing to trace potential pathways for vulnerability spread by identifying how functions interact.

5.1.1 Original Repository Normalization. Beyond merely removing whitespace and line breaks from functions, as in traditional RVD approaches, AntMan performs a comprehensive normalization on $repo_{\text{pre}}$ (resp. $repo_{\text{post}}$), from coarse-grained to fine-grained adjustments across four main steps.

(1) **Macro Expansion.** AntMan begins by iteratively processing `include` statements in each file within $repo_{\text{pre}}$ (resp. $repo_{\text{post}}$) to identify dependencies. Macro definitions from these files are consolidated into a dedicated header file, `pre.h` (resp. `post.h`), respectively. Subsequently, AntMan expands macros across all files, including dependencies, by GCC's pre-compilation instruction `gcc -E -w -include file.h file.c -o file.c`. This expansion covers statement hunks, statements, variables, and constants, ensuring consistency across the repository.

(2) **Control Block Standardization.** AntMan removes dead control blocks (e.g., `while(false)` and `if(0)`) from each function. It expands bodies of pseudo-loops like `do-while(0)`. AntMan also converts all `for` statements into `while` statements to achieve consistency in control flow representation based on C11 standard documentation [28].

(3) **Assignment Statement Deconstruction and Permutation.** AntMan separates compound assignment statements (containing both declaration and assignment) into distinct declaration and assignment statements. It reorders all declarations to the beginning of their respective scopes, arranging data types (e.g., `string`, `int`) and variable names alphabetically for uniformity.

(4) **Operator Rewriting.** AntMan standardizes operators by reordering operands to canonical forms (e.g., transforming > into <), and unifies conditional expressions within `for` and `while` loops. Additionally, AntMan expands hybrid operators (e.g., `+=`, `-=`, `>=`) into their standard equivalents, covering a set of 13 common operator transformations.

5.1.2 Patch-Based Call Graph Construction. AntMan performs selective source code extraction by starting with the modified files and recursively following their include dependencies (via `include` statements) to gather relevant source files, and generates partial call graphs using Joern [31] to address its scalability limitation when analyzing large repositories. However, such call graphs still

include many functions and call relations unrelated to the specific vulnerability, which can complicate our analysis. Hence, AntMan restricts the call graph by isolating only those functions directly affected by the patch and their callees, imposing a call depth limit of three during the call graph construction. This limit aligns with typical vulnerability propagation patterns (around 2.8 [21]).

Through macro expansion, AntMan recovers hidden calls within macros, enhancing the call graph with previously obscured relations. The resulting patch-based normalized call graphs are denoted as NCG_{pre} and NCG_{post}. Each NCG is defined by a tuple $\langle NF, NE \rangle$, where NF represents the set of functions, and NE represents the set of call relations. Each relation $ne \in NE$ is represented by a pair $\langle nf_i, nf_j \rangle$, with nf_i as the caller function and nf_j as the callee function.

5.2 Original Abstracted ICPC Construction

Inter-procedural paths trace variable flows across functions, crucial for understanding vulnerabilities that spread through interactions among functions. Meanwhile, intra-procedural paths focus on individual functions, extracting dependency chains directly tied to modified statements. By integrating both paths during signature generating, AntMan captures both the broader vulnerability impacts across functions and the finer, localized dependencies within functions. AntMan first runs `git diff` on modified functions in NF_{pre} and NF_{post}, then generates a normalized patch (pat_{norm}) that recovered hidden statements and variables.

5.2.1 Original Inter-Procedural Taint Path Extraction. AntMan first identifies variable changes at both entry points (function declarations) and exit points (function calls, returns, exception handling, and their dominating control statements) within each modified function in pat_{norm}. Using Tree-sitter [35], AntMan marks variables as sensitive when they undergo modifications (including renaming, replacement, addition, or deletion) at these critical points.

For each identified sensitive variable in the pre-patch (and corresponding post-patch) function, AntMan regards the variable as a sink, and performs backward taint analysis to trace statements influencing these variables' values, extending into caller functions. This process repeats iteratively until variables are unreachable in the caller or the caller has no further callers in NCG. Subsequently, AntMan regards the variable as a source, and conducts forward taint analysis to track how these variables impact other statements, extending into callee functions iteratively until the variables are unreachable or the callee has no further callees in NCG. The resulting paths are termed taint paths, denoted as $p^{\text{inter}}_{\text{pre}}$ (resp. $p^{\text{inter}}_{\text{post}}$).

5.2.2 Original Intra-Procedural Dependency Path Extraction. AntMan first identifies hunk changes by processing modified hunks (i.e., contiguous blocks of added or deleted statements) in each modified function in pat_{norm}. Similar to sensitive variables identification in Section 5.2.1, AntMan detects sensitive variables (i.e., renamed, replaced, added, and deleted variables) in modified hunks. Then, AntMan employs Joern [31] for forward and backward data dependency and control dependency slicing based on the program dependence graph (PDG), following MVP's slicing criteria [49]. However, unlike MVP, which applies slicing to all modified statements and structures, AntMan narrows the scope by focusing exclusively on sensitive variables. Furthermore, AntMan extends this definition of sensitive variables to include structure fields. This selective approach results in intra-procedural dependency paths, denoted as $p^{\text{intra}}_{\text{pre}}$ (resp. $p^{\text{intra}}_{\text{post}}$). When no variable or structure field is changed, AntMan defaults to the full dependency analysis approach used by MVP.

5.2.3 Original Abstracted ICPC Construction. AntMan applies type-sensitive abstraction to each statement in $p^{\text{inter}}_{\text{pre}}$ and $p^{\text{intra}}_{\text{pre}}$ (resp. $p^{\text{inter}}_{\text{post}}$ and $p^{\text{intra}}_{\text{post}}$) to enhance established RVD abstraction techniques. When data type changes are detected, AntMan preserves the original data types to capture vulnerabilities sensitive to type changes (e.g., integer overflow). Next, AntMan merges $p^{\text{inter}}_{\text{pre}}$ with

Algorithm 1 Cluster Similarity Calculation

Input: $icpc_x$, a 2-tuple of $\langle S_x, E_x \rangle$; $icpc_y$, a 2-tuple of $\langle S_y, E_y \rangle$ **Output:** similarity score of $icpc_x$ and $icpc_y$

Step 1: Compute Statement Edit Cost

for $s_i \in S_x$ and $s_j \in S_y$ **do**
  $c_s(s_i, s_j) = 1 - s_{\text{sim}}(s_i, s_j) * \frac{w_i + w_j}{2}$
end for

Step 2: Compute Statement Set Edit Cost

$c_S(S_x, S_y) = \text{Hungarian}_S(S_x, S_y)$

Step 3: Compute Edge Edit Cost

for $e_i \in E_x$ and $e_j \in E_y$ **do**
  $c_e(e_i, e_j) = 1 - \frac{c_s(e_i.s_1, e_j.s_1) + c_s(e_i.s_2, e_j.s_2)}{2}$
end for

Step 4: Compute Edge Set Edit Cost

$c_E(E_x, E_y) = \text{Hungarian}_E(E_x, E_y)$

Step 5: Compute Similarity of Cluster Pair

$c_{icpc}(icpc_x, icpc_y) = \frac{c_S(S_x, S_y) + \sqrt{c_S(E_x, E_y)}}{|S_x| + |S_y|}$
$sim_{icpc}(icpc_x, icpc_y) = 1 - c_{icpc}(icpc_x, icpc_y)$
Return $sim_{icpc}(icpc_x, icpc_y)$

$p^{\text{intra}}_{\text{pre}}$ (resp. $p^{\text{inter}}_{\text{post}}$ with $p^{\text{intra}}_{\text{post}}$) wherever shared statements exist, iteratively repeating this process until no further merging is possible. The resulting structure is termed the inter-procedural code property clusters (*ICPC*). The clusters derived from NCG_{pre} and NCG_{post} are represented as the vulnerable signature $ICPC_{\text{pre}}$ and the fixed signature $ICPC_{\text{post}}$, respectively.

5.3 Mapping & Target Normalized Call Graph Construction

AntMan first conducts the same normalization process as in Section 5.1, and then maps the functions, modified statements and sensitive variables from NCG_{pre} and NCG_{post} to $repo_{\text{t}}$ as follows.

(1) **Normalized Function Mapping.** AntMan first performs macro expansion on the entire target repository $repo_{\text{t}}$, and then maps modified functions from NCG_{pre} and NCG_{post} to $repo_{\text{t}}$ using SAGA [19], a clone detection tool optimized for high recall. Function-level clone detection is conducted with a low similarity threshold of 0.2 to capture all potential matches. In cases where multiple functions are matched, the one with the highest similarity score is selected.
(2) **Normalized Modified Statement Mapping.** AntMan maps each normalized changed statement in the matched functions based on an edit distance threshold of 0.55, as suggested by previous work [8]. Since multiple candidate statements may exist within the matched functions, AntMan applies forward and backward dependency slicing on each candidate to identify the matched statement with the most similar paths using edit distance comparisons.
(3) **Sensitive Variables Mapping.** Sensitive variables identified in $repo_{\text{o}}$ may be difficult to match due to changes like renaming, reordering, addition, or deletion. To address this, AntMan conducts a taint analysis on each variable within the mapped statements, achieving precise one-to-one mapping of variables despite these modifications.

Based on the mapped functions, AntMan constructs potential vulnerable and fixed normalized call graph (denoted as NCG'_{pre} and NCG'_{post}, respectively) in the same way in Section 5.1.

5.4 Target Abstracted ICPC Construction

Given mapped modified statements and sensitive variables, AntMan constructs potential vulnerable and fixed signature (denoted as $ICPC'_{\text{pre}}$ and $ICPC'_{\text{post}}$, respectively) in the same way in Section 5.2.

5.5 RV Similarity Calculation

5.5.1 Cluster Similarity Calculation. *ICPC* consists of several clusters, and each cluster is denoted as *icpc*. Each *icpc* is represented as a tuple $\langle S, E \rangle$, where S is the set of statements and E is the set of taint or dependency edges. Each edge $e \in E$ is a pair $\langle s_1, s_2 \rangle$, where s_1 is the source statement and s_2 is the destination statement. AntMan leverages UniXcoder [13], a cross-modal language model, to

Table 5. Results of Our Effectiveness Evaluation

		Type-I				Type-II			
		Pre.	**Rec.**	**F1.**	**SOTA**	**Pre.**	**Rec.**	**F1.**	**SOTA**
AntMan	$\mathcal{S}$	0.89 (↑**0.19**)	0.83 (–)	0.86 (↑**0.10**)	VUDDY	0.93 (↑**0.07**)	0.87 (↑**0.28**)	0.90 (↑**0.20**)	VUDDY
	$\mathcal{M}$	0.79 (↑**0.17**)	0.90 (↓**0.01**)	0.84 (↑**0.10**)	VUDDY	0.84 (↑**0.29**)	0.86 (↑**0.03**)	0.85 (↑**0.18**)	VUDDY
	$\mathcal{S}\cup\mathcal{M}$	0.86 (↑**0.19**)	0.85 (–)	0.85 (↑**0.10**)	VUDDY	0.90 (↑**0.18**)	0.86 (↑**0.20**)	0.88 (↑**0.19**)	VUDDY
		Type-III				**All**			
		Pre.	**Rec.**	**F1.**	**SOTA**	**Pre.**	**Rec.**	**F1.**	**SOTA**
AntMan	$\mathcal{S}$	0.87 (↑**0.23**)	0.84 (↑**0.46**)	0.85 (↑**0.37**)	FIRE	0.88 (↑**0.21**)	0.84 (↑**0.37**)	0.86 (↑**0.31**)	FIRE
	$\mathcal{M}$	0.80 (↑**0.14**)	0.85 (↑**0.28**)	0.82 (↑**0.21**)	FIRE	0.80 (↑**0.16**)	0.86 (↑**0.38**)	0.83 (↑**0.28**)	FIRE
	$\mathcal{S}\cup\mathcal{M}$	0.83 (↑**0.18**)	0.84 (↑**0.36**)	0.84 (↑**0.29**)	FIRE	0.84 (↑**0.18**)	0.85 (↑**0.33**)	0.84 (↑**0.26**)	FIRE

convert statements into vectors. After normalizing these vectors with L2 norm, it computes statement similarities using cosine similarity, ensuring accurate comparison of code semantics. Based on the statement similarities, the similarity between $icpc_x$ and $icpc_y$ from $ICPC_{\text{pre}}$ and $ICPC'_{\text{pre}}$ (respectively $ICPC_{\text{post}}$ and $ICPC'_{\text{post}}$) is calculated following the five steps, as shown in Algorithm 1.

Inspired by the findings of Cui et al. [5] that statements farther from the root (containing sensitive identifiers) in a dependency path have less impact on the vulnerability, each statement's weight w is set to $\frac{1}{d+1}$, where d is the distance to the nearest statement with sensitive variables in $icpc$. The edit cost between two statements within $icpc_x$ and $icpc_y$ is then calculated with this weighting (Line 2 of Algorithm 1). AntMan computes the weighted edit cost of statements within $icpc_x$ and $icpc_y$ using Hungarian matching, as described by Cui et al. [5] (Line 4). Similarly, the edge edit costs and edge set edit costs are computed (Lines 6 and 8), based on the average similarity between the source and target statements in each edge. Finally, AntMan calculates the similarity between $icpc_x$ and $icpc_y$ by aggregating the costs of statements and edges (Lines 9 and 10). A similarity score above the threshold th_{vul} for $icpc_x$ and $icpc_y$ from $ICPC_{\text{pre}}$ and $ICPC'_{\text{pre}}$ indicates a vulnerable cluster pair $icpc_{\text{vul}}$, while a similarity score above the threshold th_{fix} for $icpc_x$ and $icpc_y$ from $ICPC_{\text{post}}$ and $ICPC'_{\text{psot}}$ marks a fixed cluster pair $icpc_{\text{fix}}$.

5.5.2 RV Detection. Following Equation 1 and 2, if the proportion of vulnerable cluster pairs exceeds pro_{vul} and the proportion of fixed cluster pairs remains below pro_{fix}, $repo_{\text{t}}$ is flagged as vulnerable and an RV is detected; otherwise, $repo_{\text{t}}$ is considered as not containing the RV.

$$\frac{|icpc_{\text{vul}}|}{|ICPC_{\text{pre}}.icpc|} \geq pro_{\text{vul}} \quad (1) \qquad \frac{|icpc_{\text{fix}}|}{|ICPC_{\text{post}}.icpc|} < pro_{\text{fix}} \quad (2)$$

6 Evaluation

We design the following seven research questions to evaluate the effectiveness, efficiency and practical usefulness of AntMan. We conduct the experiments on a machine with an Intel(R) Xeon(R) Silver 4314 CPU, a GeForce RTX 3090 GPU and 256 GB memory, running Ubuntu 22.04 OS.

- **RQ4 Effectiveness Evaluation.** How is the effectiveness of AntMan?
- **RQ5 Ablation Study.** How is the contribution of each component to the effectiveness of AntMan?
- **RQ6 Parameter Sensitivity Analysis.** How do the configurable parameters affect AntMan?
- **RQ7 Generality Evaluation.** How is the generality of AntMan beyond our dataset?
- **RQ8 0-Day Detection Capability.** How is AntMan's capability to detect 0-day vulnerabilities?
- **RQ9 Efficiency Evaluation.** How is the efficiency of AntMan?
- **RQ10 Usefulness Evaluation.** How is the practical usefulness of AntMan?

6.1 Effectiveness Evaluation (RQ4)

To evaluate AntMan's effectiveness, we compared it against the best RVD approach for each similarity type and patch scope, as denoted by "SOTA" in Table 5. We used F1-score as the primary metric to

Table 6. Results of Our Ablation Study

	w/o *norm.*	w/o p^{intra}	w/o p^{inter}	w/o *abs.*	w/o w	w/ *LD.*	w/ *CodeBERT*	w/ *L1.*
Pre.	0.81 (↓**0.03**)	0.64 (↓**0.20**)	0.67 (↓**0.17**)	0.79 (↓**0.05**)	0.73 (↓**0.11**)	0.69 (↓**0.15**)	0.74 (↓**0.10**)	0.73 (↓**0.11**)
Rec.	0.76 (↓**0.09**)	0.74 (↓**0.11**)	0.76 (↓**0.09**)	0.77 (↓**0.08**)	0.79 (↓**0.06**)	0.80 (↓**0.05**)	0.83 (↓**0.02**)	0.81 (↓**0.04**)
F1.	0.78 (↓**0.07**)	0.69 (↓**0.16**)	0.71 (↓**0.14**)	0.78 (↓**0.07**)	0.76 (↓**0.09**)	0.74 (↓**0.11**)	0.78 (↓**0.07**)	0.77 (↓**0.08**)

determine the best-performing approach, given its comprehensive measure of both precision and recall. According to Table 2, VUDDY is the best RVD approach for Type-I and Type-II clones across both $\mathcal{S}$ and $\mathcal{M}$; and FIRE is the best RVD approach for Type-III and all clones across both $\mathcal{S}$ and $\mathcal{M}$.

Overall Analysis. When considering all types of clones and patch scopes, AntMan achieves the highest precision of 0.84 and recall of 0.85, leading to the highest F1-score of 0.84. AntMan outperforms the best state-of-the-art FIRE by 0.18 (27%) in precision, 0.33 (63%) in recall, and 0.26 (45%) in F1-score, demonstrating substantial improvements over the state-of-the-arts. For Type-I clones, AntMan has an F1-score of 0.86 for $\mathcal{S}$, with an improvement by 0.10 (13%) and 0.84 for $\mathcal{M}$ with an improvement by 0.10 (14%), surpassing the best state-of-the-art VUDDY. For Type-II clones, AntMan attains an even higher F1-score of 0.90 for $\mathcal{S}$, with an improvement by 0.20 (29%) and 0.85 for $\mathcal{M}$, with an improvement by 0.18 (27%), surpassing the best state-of-the-art VUDDY. For Type-III clones, which are the most challenging and the largest proportion in RVD, AntMan achieves the highest precision of 0.83 and recall of 0.84, leading to the highest F1-score of 0.84. AntMan significantly outperforms the best state-of-the-art FIRE by 0.18 (28%) in precision, 0.36 (75%) in recall, and 0.29 (53%) in F1-score.

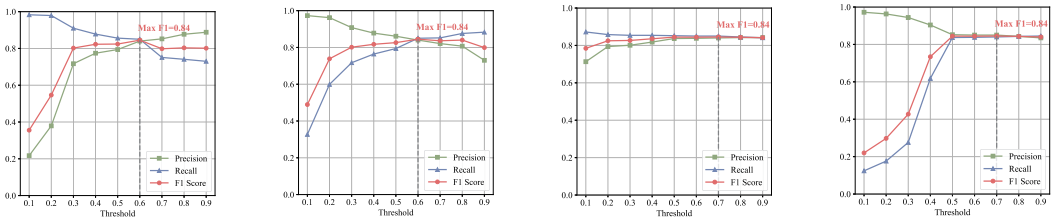
FP/FN Analysis. For our constructed dataset, AntMan generates 721 FPs and 698 FNs. We summarize four major reasons for them. First, in the absence of changed variables, AntMan constructs a dependency path without sensitive variables, resulting in irrelevant dependencies, which leads to FPs and FNs. Second, original vulnerabilities may have different fixing logics across branches, which are not all captured by the fixed signatures, leading to FNs. Third, AntMan uses precompiled instructions for macro expansion, and some macros incorporate compilation optimizations in their semantics, introducing unrelated semantic information to the signatures during macro expansion, which leads to FPs and FNs. Fourth, AntMan leverages Joern to perform dependency slicing, but the inaccurate slicing of Joern sometimes leads to FPs and FNs.

Summary: AntMan achieves the highest precision of 0.84 and recall of 0.85, leading to the highest F1-score of 0.85 across all clone types and patch scopes, outperforming the best state-of-the-art FIRE by 0.18 (27%) in precision, 0.33 (63%) in recall, and 0.26 (45%) in F1-score, demonstrating substantial improvements over the state-of-the-arts. For Type-III clones, which are the most challenging and the largest proportion in RVD, AntMan significantly outperforms FIRE by 0.18 (28%) in precision, 0.36 (75%) in recall, and 0.29 (53%) in F1-score.

6.2 Ablation Study (RQ5)

We created eight ablated versions of AntMan, i.e., (1) constructing call graphs without normalization (w/o *norm.*); (2) constructing dependency paths without sensitive variables (w/o p^{intra}); (3) constructing *ICPC* without inter-procedural taint paths (w/o p^{inter}); (4) constructing *ICPC* without abstraction (w/o *abs.*); (5) constructing *ICPC* without weights (w/o w); (6) replacing UniXcoder in RV similarity calculation with Levenshtein distance (w/ *LD.*); (7) replacing UniXcoder with CodeBERT (w/ *CodeBERT*); and (8) replacing L2 norm in RV similarity calculation with L1 norm (w/ *L1.*).

Table 6 reports the results of our ablation study. Overall, both precision and recall decrease across the eight ablated versions. AntMan w/o p^{intra} exhibits the most substantial precision drop of 0.20 and the most significant recall decrease of 0.11, resulting in a notable F1-score drop of 0.16, emphasizing the importance of dependency paths involving sensitive variables. Meanwhile, AntMan w/o



(a) th_{vul} (b) th_{fix} (c) pro_{vul} (d) pro_{fix}

Fig. 3. Results of Our Parameter Sensitivity Analysis

Table 7. Results of Our Generality Evaluation

	VUDDY	MVP	MOVERY	VIScan	FIRE	SySeVR	DeepDFA	AntMan
TP	287	244	438	255	340	106	675	722
FP	73	170	206	92	137	146	348	137
FN	526	569	375	558	473	707	138	91
Pre.	0.80	0.59	0.68	0.73	0.71	0.42	0.66	**0.84**
Rec.	0.35	0.30	0.54	0.31	0.42	0.13	0.83	**0.88**
F1.	0.49	0.40	0.60	0.44	0.53	0.18	0.73	**0.86**

p^{inter} suffers the second largest precision drop of 0.17 with a large recall drop of 0.09, leading to the second largest F1-score drop of 0.14, revealing the importance of inter-procedural taint paths. In addition, ablating other components of AntMan consistently results in performance degradation, demonstrating the value of these components in maintaining AntMan's effectiveness.

Summary: Ablating all components of AntMan results in substantial effectiveness drops. Specifically, ablating dependency paths of sensitive variables (w/o p^{intra}) suffers the largest F1-score drop of 0.16, while constructing *ICPC* without inter-procedural taint paths (w/o p^{inter}) results in the second largest F1-score drop of 0.14.

6.3 Parameter Sensitivity Analysis (RQ6)

Four parameters are configurable in AntMan, including the threshold th_{vul} for vulnerable *icpc* cluster pair and th_{fix} for fixed *icpc* cluster pair (see Section 5.5.1), and the proportion threshold pro_{vul} for vulnerable detection and pro_{fix} for fixed detection (sec Section 5.5.2). We reconfigured one parameter by a step of 0.1 and fixed the other three to evaluate affect to effectiveness of AntMan. Figure 3 reports the results. We can observe that these parameters should be configured to a value that is larger than 0.5 for a better performance. AntMan performs the best with th_{vul} set to 0.6, th_{fix} set to 0.6, pro_{vul} set to 0.7, and pro_{fix} set to 0.7, which is the configuration used in other RQs.

Summary: AntMan performs the best with th_{vul}, th_{fix}, pro_{vul}, pro_{fix} set to 0.6, 0.6, 0.7, 0.7.

6.4 Generality Evaluation (RQ7)

Generality Dataset Construction. AntMan was designed based on insights from our empirical study. To evaluate its generality, we constructed a new dataset following the methodology introduced in Section 4.1.2. Specifically, we collected the newest C/C++ vulnerabilities reported between 1 January 2024 and 7 August 2024, gathering a set of 186 vulnerabilities with their corresponding patches. We then used these original vulnerability patches as inputs to detect RVs by the existing RVD approaches and AntMan. After sample confirmation and expansion, we finally gathered 813 positive and 260 negative samples with Cohen's Kappa coefficient of 0.958 and 0.969, respectively.

Baseline Selection. We selected the five RVD approaches previously discussed, along with two state-of-the-art, general-purpose vulnerability detection approaches, i.e., SySeVR [22] and DeepDFA [32], both of which employ learning-based models. We trained these models on our ground truth dataset and evaluated their performance using the new dataset to ensure a fair comparison.

Table 8. Results of the 0-Day Detection Capability

	VUDDY	MVP	V1scan	MOVERY	FIRE	SySeVR	DeepDFA	AntMan
#. (Proportion)	0 (0%)	42 (51%)	0 (0%)	24 (29%)	29 (35%)	6 (7%)	65 (79%)	**73 (89%)**

Table 9. Results of Our Efficiency Evaluation

	VUDDY	MVP	MOVERY	V1scan	FIRE	AntMan
Time (s)	**45.3**	195.1	187.3	53.9	112.9	223.1

Overall Results As shown in Table 7, for the new dataset, AntMan achieved a precision of 0.84, a recall of 0.88, and an F1-score of 0.86. When compared to existing RVD approaches, AntMan showed significant improvements in precision by 0.16 (24%), in recall by 0.34 (63%), and in F1-score by 0.26 (43%). For learning-based vulnerability detection approaches, SySeVR achieved a precision of 0.42 and a recall of 0.13, leading to an F1-score of 0.18, while DeepDFA had a precision of 0.66 and a recall of 0.83, resulting in an F1-score of 0.73. AntMan outperformed the best one DeepDFA by 0.18 (27%) in precision, 0.05 (6%) in recall, and 0.13 (18%) in F1-score.

Summary: AntMan outperformed the best learning-based approach DeepDFA by 27% in precision, 6% in recall, and 18% in F1-score. Compared with the best RVD approaches, AntMan again showed superior performance with an improvement by 24% in precision, 63% in recall, and 43% in F1-score, demonstrating AntMan's effectiveness as the leading approach among all RVD approaches.

6.5 0-Day Detection Capability (RQ8)

Following the same procedure as in Section 4.2, we obtained 46 0-day vulnerabilities in our generality dataset. To assess the 0-day vulnerability detection capability of AntMan, we compared AntMan with the five RVD approaches and the two learning-based approaches in **RQ7** in terms of the proportion of detected 0-day vulnerabilities in our ground truth dataset and generality dataset.

As reported in Table 8, among the 82 0-day vulnerabilities in our ground truth dataset and generality dataset, AntMan detected 73 (89%) 0-day vulnerabilities, outperforming the best learning-based approach by a significant margin of 13%. Besides, all other RVD approaches showed poor performance in detecting 0-day vulnerabilities, with AntMan leading by a notable improvement over the best one, indicating AntMan's superior capability in detecting RVs with significant logic difference.

Summary: AntMan successfully detected 73 (89%) of the 0-day vulnerabilities, outperforming the best state-of-the-art approach by a significant margin of 13%.

6.6 Efficiency Evaluation (RQ9)

We measured the average time taken to detect RVs in a single repository using all the original vulnerabilities collected in Section 4.1.2. Here, we excluded the time of original signature generation for all the approaches because this step can be done offline. As shown in Table 9, AntMan took 223.1 seconds on average to detect RVs in a single repository, which was longer than the time of the five existing RVD approaches. Extremely, AntMan took 27,623 seconds on the largest repository (i.e., Linux v6.5.6) which has 17.2 million lines of code, whereas the fastest approach VUDDY took 21,102 seconds. This increased time overhead is primarily due to our *NCG* construction facilitated by Joern, but AntMan still scales to large repositories. We believe that this time cost is acceptable given AntMan's high effectiveness for RVD.

Summary: AntMan took 223.1 seconds on average to detect RVs in one repository.

6.7 Usefulness Evaluation (RQ10)

For the RVs in our ground truth and generality dataset, AntMan have detected 4,520 1/N-day vulnerabilities and 73 0-day vulnerabilities. We notified target repositories of 274 1/N-day vulnerabilities and all 0-day vulnerabilities via issue reports, pull requests or emails. Among these, 188 1/N-day vulnerabilities have been confirmed and fixed, 52 1/N-day vulnerabilities have been confirmed and promised to be fixed in the next released version, and the other 34 1/N-day vulnerabilities are still under confirmation. Notably, 67 0-day vulnerabilities have been confirmed and fixed due to their high-risk nature, and the other 6 0-day vulnerabilities are still in progress. As a 0-day vulnerability can exist in multiple versions of a repository, we deduplicated our detected 0-day vulnerabilities to isolate 21 unique 0-day vulnerabilities, which were subsequently reported to CVE. Out of these, 5 were successfully assigned a CVE identifier and other 16 are pending to be confirmed.

To further evaluate the usefulness of AntMan, we conducted a human study by surveying 22 active project maintainers whose projects had previously been analyzed by AntMan. 10 (45%) of the surveyed maintainers provided detailed feedback. All respondents confirmed that AntMan enhanced their projects' security and expressed interest in incorporating AntMan into their development workflow to conduct regular scanning. In terms of improvement suggestions, they provided two key insights: (1) the availability of patches for publicly disclosed vulnerabilities can be an issue, and there is a need for a larger number of valid patches to enrich the signature database; and (2) the reports generated by AntMan would be more useful if they included Proof of Concepts (PoCs) for RVs, as this would help expedite the confirmation and remediation process.

Summary: We notified target repositories of 274 1/N-day vulnerabilities with 188 of them confirmed, and notified target repositories of 73 0-day vulnerabilities with 67 of them confirmed. We reported 21 0-day vulnerabilities with 5 CVE identifiers assigned. Our human study validated AntMan's practical value and provided constructive feedback for future improvements.

7 Limitations

First, AntMan currently accepts only one single patch as the input, which may lead to missing patch features in repositories where patches on different branches differ significantly[47]. Therefore, a multi-branch analysis capability would provide more comprehensive detection. Second, AntMan does not currently consider patches that involve only external function modifications, e.g., macro or structural changes, as these are generally simpler and involve minimal functional impact. However, some such changes could have security implications, suggesting the need for a broader analysis scope. Third, we intentionally design AntMan at the source code level for scalability, leveraging Joern to conduct static analysis at the source code level. As a result, inaccuracies for dynamic features like function pointers and virtual functions in Joern might negatively affect AntMan. However, this is orthogonal to AntMan, and more advanced static analysis can be leveraged. Finally, AntMan has been implemented for C/C++, and we plan to adapt it to other languages.

8 Conclusion and Data Availability

We conduct a large-scale empirical study and uncover three key insights (i.e., broad context awareness, fine-grained signature, and flexible matching). Based on these insights, we develop a novel approach AntMan, which demonstrates its effectiveness, generality and practical usefulness through our extensive evaluation. The source code of AntMan is available at our website [34].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62372114 and 62332005).

References

[1] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. 2024. Coca: Improving and Explaining Graph Neural Network-Based Vulnerability Detection Systems. In *Proceedings of the 46th International Conference on Software Engineering*. 1–13.

[2] Checkmarx. 2024. *Checkmarx*. Retrieved October 25, 2024 from https://checkmarx.com/

[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2095–2108.

[4] Qicai Chen, Kun Hu, Sichen Gong, Bihuan Chen, kevin kong, Haowen Jiang, Bingkun Sun, You Lu, and Xin Peng. 2025. Structure-Aware, Diagnosis-Guided ECU Firmware Fuzzing. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[5] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2020. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* 16 (2020), 2004–2017.

[6] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22 USENIX Security Symposium*. 463–478.

[7] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*. 60–71.

[8] Ekwa Duala-Ekoko and Martin P Robillard. 2007. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*. 158–167.

[9] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185.

[10] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. 2024. {FIRE}: Combining {Multi-Stage} Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1867–1884.

[11] GitHub. 2024. *CodeQL*. Retrieved October 25, 2024 from https://codeql.github.com/

[12] gpac. 2024. *CVE-2022-46489*. Retrieved October 25, 2024 from https://github.com/gpac/gpac/commit/44e8616ec6d0c37498cdacb81375b09249fa9daa

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[14] Kaifeng Huang, Chenhao Lu, Yiheng Cao, Bihuan Chen, and Xin Peng. 2024. VMUD: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3958–3972.

[15] 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.

[16] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the Symposium on Security and Privacy*. 258–263.

[17] 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*. 1695–1708.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the Symposium on Security and Privacy*. 595–614.

[19] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*. 272–283.

[20] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 627–637.

[21] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

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

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
[24] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*. IEEE, 692–708.
[25] NVD. 2024. *CVE-2022-46489*. Retrieved October 25, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2022-46489
[26] NVD. 2024. *NVD Data Feeds*. Retrieved October 25, 2024 from https://nvd.nist.gov/vuln/data-feeds
[27] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. 2016. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 780–785.
[28] Openstd. 2024. *Open Std*. Retrieved April 20, 2024 from https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf
[29] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE international conference on machine learning and applications*. 757–762.
[30] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955
[31] ShiftLeftSecurity. 2024. *Joern*. Retrieved October 25, 2024 from https://github.com/ShiftLeftSecurity/joern
[32] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th International Conference on Software Engineering*. 1–13.
[33] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Symposium on Network and Distributed System Security*. 1–16.
[34] AntMan. 2024. *AntMan*. Retrieved October 25, 2024 from https://antman-opensource.github.io/
[35] tree sitter. 2023. Tree-sitter: An incremental parsing system for programming tools. Retrieved September 1, 2024 from https://tree-sitter.github.io/tree-sitter/
[36] Haoxin Tu. 2023. Boosting symbolic execution for heap-based vulnerability detection and exploit generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 218–220.
[37] Julien Vanegue and Shuvendu K Lahiri. 2013. Towards practical reactive security audit using extended static checkers. In *Proceedings of the Symposium on Security and Privacy*. 33–47.
[38] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. 579–594.
[39] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering*. 724–735.
[40] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In *Proceedings of the Symposium on Network and Distributed System Security*. 1–14.
[41] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and evaluation of clone detection techniques with different code representations. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 332–344.
[42] Yuanpeng Wang, Ziqi Zhang, Ningyu He, Zhineng Zhong, Shengjian Guo, Qinkun Bao, Ding Li, Yao Guo, and Xiangqun Chen. 2023. Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2710–2724.
[43] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–357.
[44] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In *Proceedings of the 32nd USENIX Security Symposium*. 6541–6556.
[45] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium*. 3037–3053.
[46] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities. In *Proceedings of the 46th International Conference on Software Engineering*. 1–12.
[47] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, Zhao Junpeng, and Xin Peng. 2025. Mystique: Automated Vulnerability Patch Porting with Semantic and Syntactic-Enhanced LLM. *Proceedings of*

the ACM on Software Engineering 2, FSE (2025).

[48] Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying affected library versions for open source software vulnerabilities. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1447–1459.

[49] 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 *Proceedings of the 29th USENIX Security Symposium*. 1165–1182.

[50] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 499–510.

[51] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*.

[52] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1633–1644.

Received 2025-02-24; accepted 2025-03-31