USENIX Security '24

FIRE: Combining Multi-Stage Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection

Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, Hai Jin

fengsiyue@hust.edu.cn

Huazhong University of Science and Technology (HUST), Wuhan, China
Nanyang Technological University, (NTU), Singapore

Recurring Vulnerabilities

- With the development of software open-sourcing, reusing software becomes common.

96%

of the total codebases **contained** open source

84%

of codebases contained at least **one open source vulnerability**

An increasing number of recurring vulnerabilities

Related Work

Approach	Speed	Syntactic changes	Patch information
VUDDY	★★★★	×	×
MOVERY	★★	★★★	★★
ReDeBug	★★★★	×	×
MVP	★★	★★★	★★★

They either fail to detect recurring vulnerabilities with syntax changes, do not consider patch information, or have high time overhead.

Research Problem

There is a current need for a method that can:

Enable rapid detection of extremely large-scale recurring vulnerabilities.

Support for detecting vulnerabilities that make syntactically different but semantically identical changes.

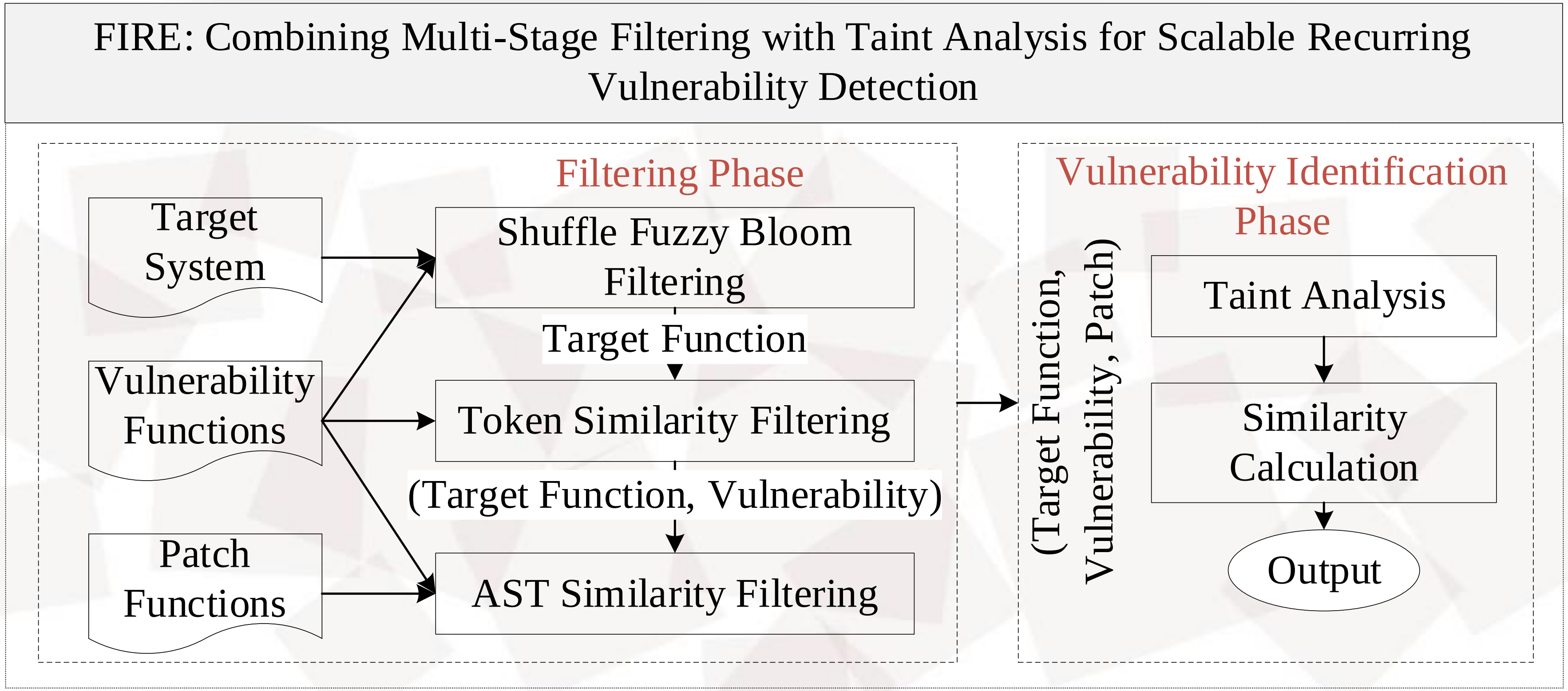
Consider the differences between vulnerabilities and patches.

Our Contributions

- A novel method based on multi-stage filtering and differential tainted paths.
- A prototype system (i.e. ***FIRE***) for effective and scalable detection of recurring vulnerabilities in open-source software.
- A comparative evaluation of FIRE against state-of-the-art vulnerability detection methods.

1 System Design

Overview of FIRE



Filtering Phase

Bloom Filter, Token Filter, and AST Filter

Vulnerability Identification Phase

Extract taint paths from source code

Filtering Phase

Goal	Input	Tool	Output
Reduce the functions to be inspected in the next stage	Source code of the target software	Bloom Filter, Token Filter, and AST Filter	Potentially vulnerable target functions

Crucial Features of Vulnerabilities

The first group consists of 42 **sensitive APIs**, the improper use of which may lead to issues such as memory leaks and buffer overflows.

The second group consists of 20 **format strings**, where improper validation of input or use of format strings can lead to security vulnerabilities such as code injection attacks.

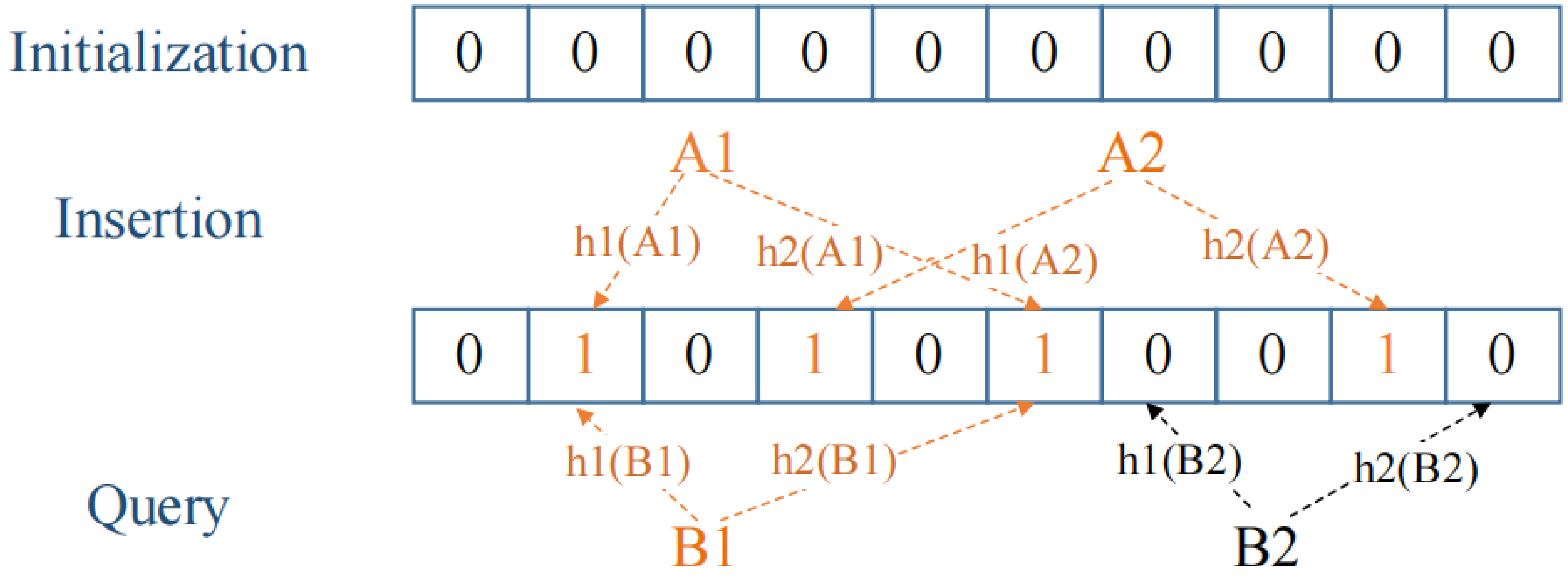
The third group consists of 42 **operators**, the use of which can lead to issues such as integer overflows and bit manipulation errors.

The fourth group consists of 73 **C/C++ keywords**. Keywords are identifiers with special meanings.

Table 4: Simple Vulnerability Features

<table>
<tr><td>sensitive APIs</td><td>alloc, free, mem, copy, new, open, close, delete, create, release, sizeof, remove, clear, dequene, enquene, detach, Attach, str, string, lock, mutex, spin, init, register, disable, enable, put, get, up, down, inc, dec, add, sub, set, map, stop, start, prepare, suspend, resume, connect</td></tr>
<tr><td>format strings</td><td>%d, %i, %o, %u, %x, %X, %f, %F, %e, %E, %g, %G, %a, %A, %c, %C, %s, %S, %p, %n</td></tr>
<tr><td>operators</td><td>bitand, bitor, xor, not, not_eq, or, or_eq, and, ++, –, +, -, *, /, %, =, +=, -=, *=, /=, %=, «=, »=, &=, ≙, |=, &&, ||, !, ==, !=, >=, <=, >, <, &, |, «, », , ;̂ -></td></tr>
<tr><td>C/C++ keywords</td><td>asm, auto, alignas, alignof, bool, break, case, catch, char, char16_t, char32_t, class, const, const_cast, constexpr, continue, decltype, default, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, noexcept, nullptr, operator, private, protected, public, reinterpret_cast, return, short, signed, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, compl, override, final, assert</td></tr>
</table>

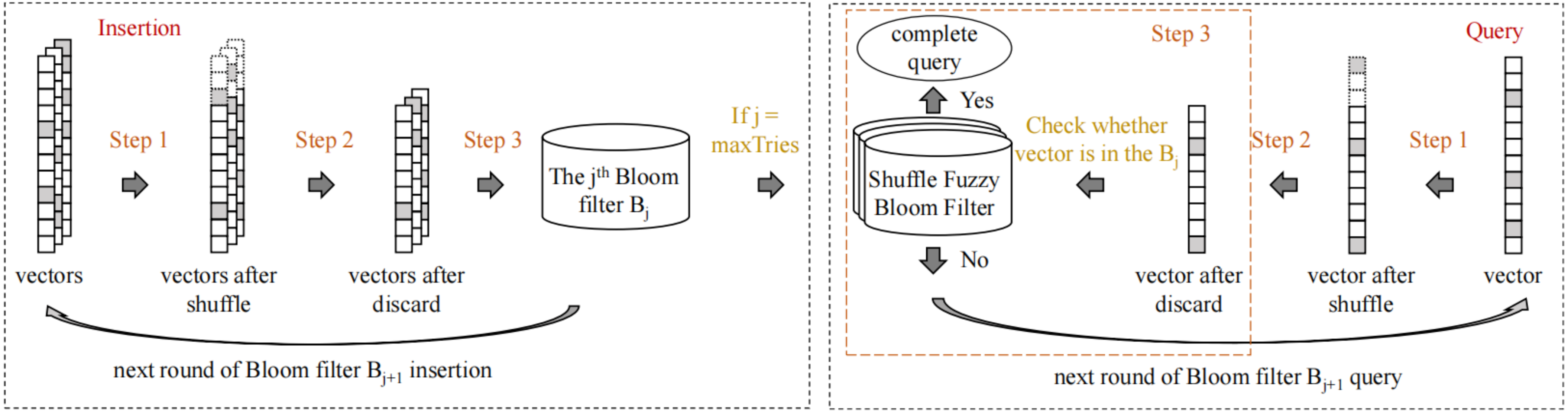
Standard Bloom Filter



A2 (1, 2, 3, 4, 5)

B2 (1, 2, 3, 4, 6)

Shuffle Fuzzy Bloom Filter



Insertion phase: Shuffle A2 (1, 2, 3, 4, 5) to (5, 4, 3, 1, 2) ,
Discard the first element to get A'2 (4, 3, 1, 2).

Query phase: Shuffle B2 (1, 2, 3, 4, 6) to (6, 4, 3, 1, 2),
Discard the first element to get B'2 (4, 3, 1, 2).

Token Similarity Filter

- Parse the function and extract token sets.
- Calculate the similarity score between the token sets of the function and vulnerability by using Jaccard similarity.

$$Jaccard\,(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- Retain the functions with high similarity scores and all their corresponding similar vulnerabilities.

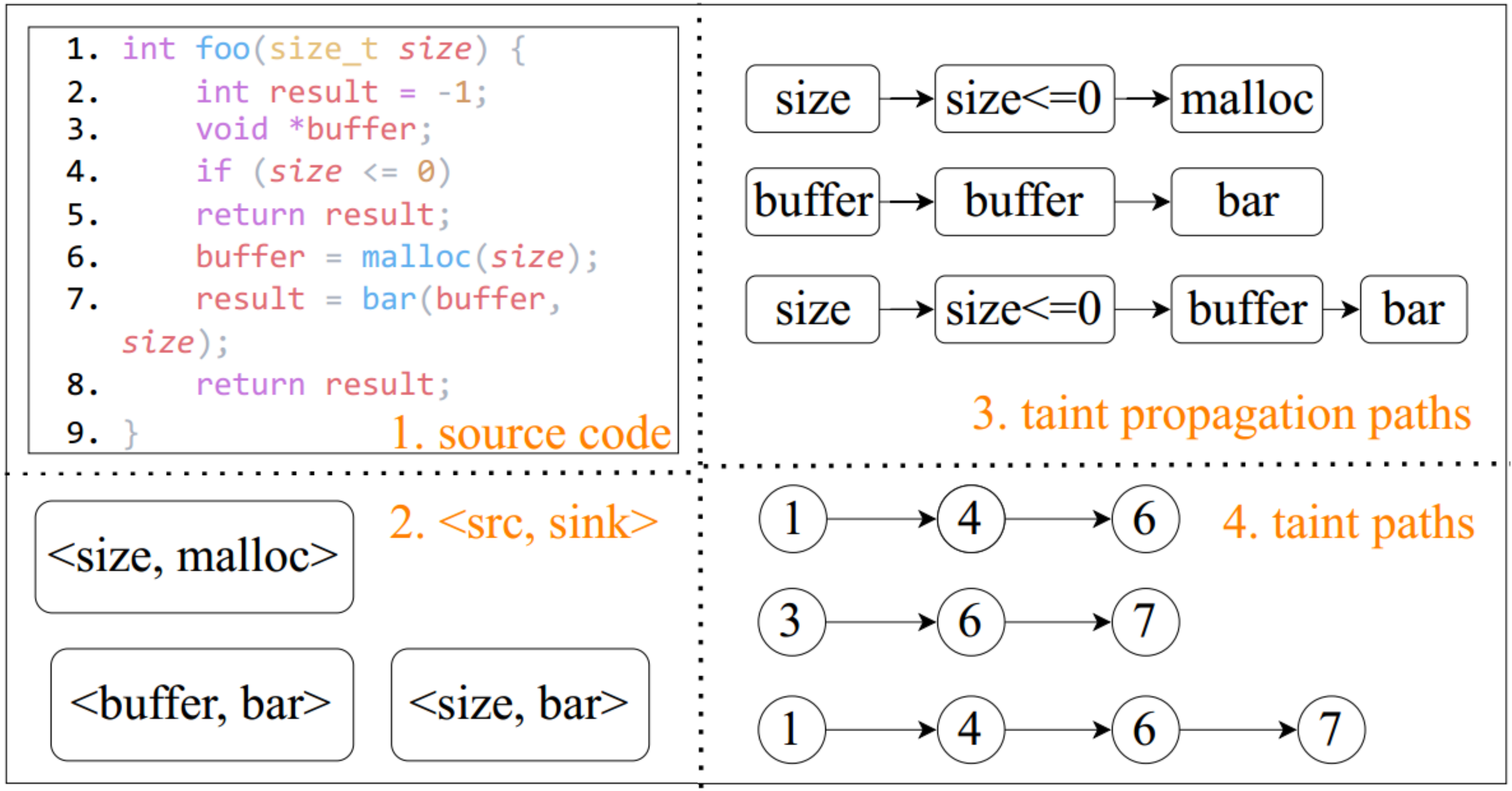
AST Similarity Filter

- Delete Lines and Add Lines
  - **C1: The target function must incorporate all deleted statements, *i.e.,* $\forall h \in S_{del}, h \in F$.**
  - **C2: The target function must not include any of the added statements, i.e., $\forall h \in S_{add}, h \notin F$.**
- Measure the similarity by calculating the number of nodes shared between two ASTs, the target function must satisfy the following conditions:
  - **C3: The similarity between target function and vulnerable function should surpass a predefined threshold, *i.e., Sim(AST_F,AST_Fv) ≥ T2*.**
  - **C4: The target function should have a higher syntactically similarity to the vulnerable function, *i.e., Sim(AST_F,AST_Fv) ≥ Sim(AST_F,AST_Fp)*.**

Vulnerability Identification Phase

Goal	Input	Tool	Output
Determine if the target function is a vulnerability	Target function, vulnerability, and patched functions	Differing taint path	Target functions that are verified as vulnerabilities

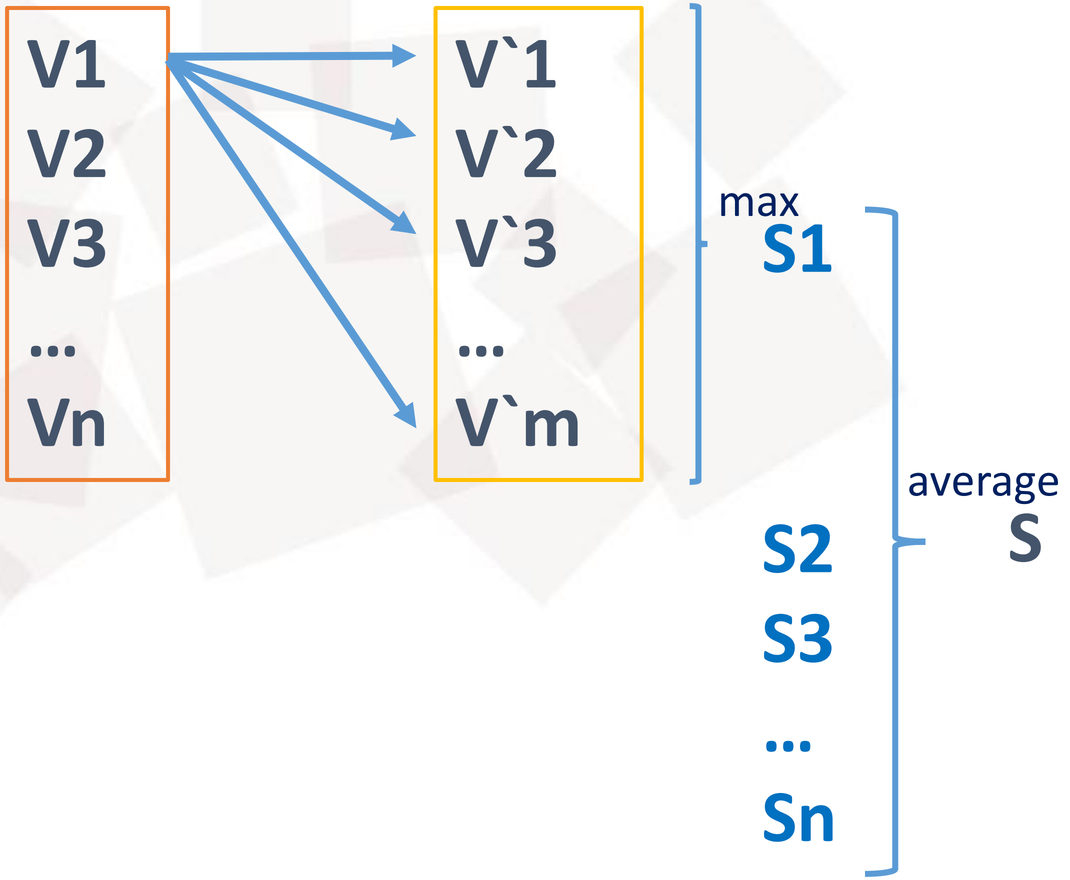
Signature Extraction



- **Extract taint paths from source code**
- **Extract Vulnerability and Patch Signatures**

Vulnerability Detection

- Extract vector representations for all paths
- Compute the similarity between target function vectors and vulnerable function and patch function vectors
- The target function should have a higher similarity to the vulnerable function



2 Evaluations

Evaluations

- Datasets and Metrics
- Detection Effectiveness
- Detection Efficiency
- The Significance of Multi-Stage Filters

Datasets and Metrics

- **Vulnerability Dataset:** 11,167 security patches from PatchDB and 10,874 manually collecting vulnerability from CVE.
- **Target Systems:** Ten popular C/C++ open-source projects that cover various application domains.

IDX	Name	Version	#Lines	Domain
T1	FreeBSD	12.2.0	15,573,896	Operating System
T2	SeaMonkey	2.53.18	8,370,870	Internet App Suite
T3	Turicreate	6.4.1	5,003,684	Machine Learning
T4	MongoDB	r4.2.11	3,295,598	Database
T5	Xemu	0.7.118	1,642,871	Emulator
T6	PHP	8.3.2	1,390,193	Scripting Language
T7	OpenCV	4.5.1	1,201,122	Computer Vision
T8	FFmpeg	4.3.2	1,118,186	Multimedia Processing
T9	Xen	4.17.3	527,124	Virtualization
T10	OpenMVG	2.1	490,103	Image Processing
Total	-	-	38,613,647	-

Datasets and Metrics

Setup:

A server with 3.40 GHz Intel i7-13700k processor and 48 GB of RAM, running on ArchLinux with Linux Zen Kernel

Precision, Recall, F1

Precision = TP/(TP+FP),

Recall = TP/(TP+FN)

F1 = 2*Precision*Recall/(Precision+Recall)

Comparative Systems:

- VUDDY
- MOVERY

Evaluations

- Datasets and Metrics
- **Detection Effectiveness**
- Detection Efficiency
- The Significance of Multi-Stage Filters

Detection Effectiveness

Table 1: The True Positive, False Positive, False Negative, Precision, and Recall of VUDDY, MOVERY, and FIRE

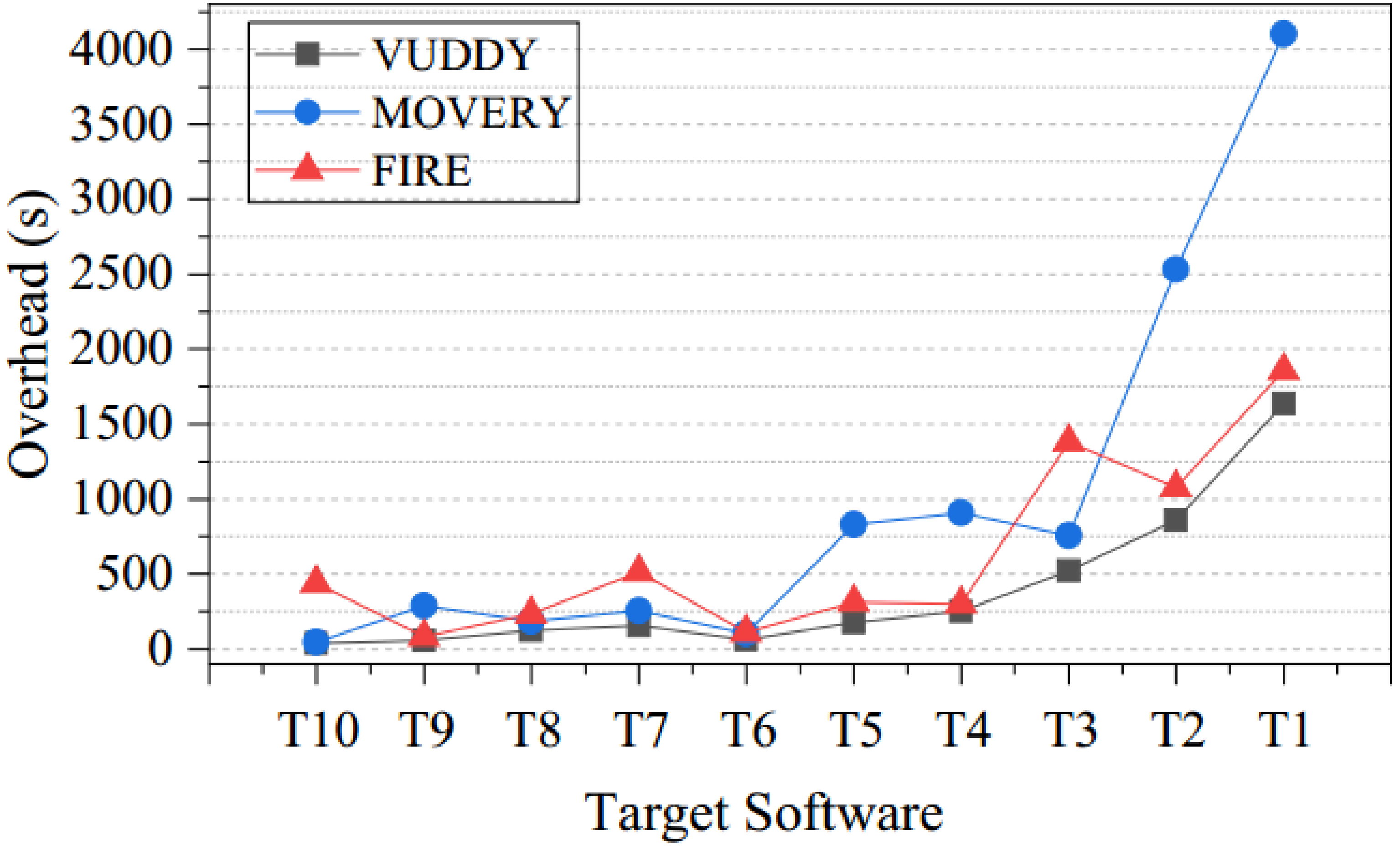
IDX	Target System	GT	VUDDY					MOVERY					FIRE				
			TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
T1	FreeBSD	104	36	17	68	67.9%	34.6%	30	34	74	46.9%	28.8%	78	7	26	91.8%	75.0%
T2	SeaMonkey	23	11	14	12	44.0%	47.8%	3	7	20	30.0%	13.0%	16	1	7	94.1%	69.6%
T3	Turicreate	44	20	11	24	64.5%	45.5%	13	17	31	43.3%	29.5%	38	6	6	86.4%	86.4%
T4	MongoDB	10	6	2	4	75.0%	60.0%	6	7	4	46.2%	60.0%	7	0	3	100.0%	70.0%
T5	Xemu	7	4	21	3	16.0%	57.1%	0	2	7	0.0%	0.0%	4	1	3	80.0%	57.1%
T6	PHP	10	3	4	7	42.9%	30.0%	2	13	8	13.3%	20.0%	7	0	3	100.0%	70.0%
T7	OpenCV	127	74	11	53	87.1%	58.3%	49	29	78	62.8%	38.6%	101	3	26	97.1%	79.5%
T8	FFmpeg	9	3	4	6	42.9%	33.3%	1	4	8	20.0%	11.1%	6	7	3	46.2%	66.7%
T9	Xen	3	0	4	3	0.0%	0.0%	1	2	2	33.3%	33.3%	2	6	1	25.0%	66.7%
T10	OpenMVG	48	20	0	28	100.0%	41.7%	12	4	36	75.0%	25.0%	39	2	9	95.1%	81.3%
Total	-	385	177	88	208	66.8%	46.0%	117	119	268	49.6%	30.4%	**298**	**33**	**87**	**90.0%**	**77.4%**

FIRE outperforms VUDDY and MOVERY in detecting recurring vulnerabilities

Evaluations

- Datasets and Metrics
- Detection Effectiveness
- **Detection Efficiency**
- The Significance of Multi-Stage Filters

Detection Efficiency



FIRE can scale to large software projects, meeting practical needs in real-world applications

Evaluations

- Datasets and Metrics
- Detection Effectiveness
- Detection Efficiency
- **The Significance of Multi-Stage Filters**

The Significance of Multi-Stage Filters

	Bloom Filter	Token Filter	AST Filter	Taint Path
Filtering Rate	80.63%	99.82%	99.96%	99.97%
Recall	93.24%	99.27%	91.97%	99.99%
Speed (f/s)	167.71	54.31	1.43	0.12

The number of functions retained after each filtering layer significantly decreases, indicating that each filtering layer we set up plays a role.

3 Conclusion

Conclusion

✧ ***FIRE:*** **a novel method based on multi-stage filtering and differential tainted paths.**

- Rapidly detect extensive recurring vulnerabilities through multi-stage filtering.
- Support for detecting complex recurring vulnerabilities with syntax changes.
- Consider differences between vulnerabilities and patches by using differential taint paths.

Thanks!

https://github.com/CGCL-codes/FIRE