Zhejiang University

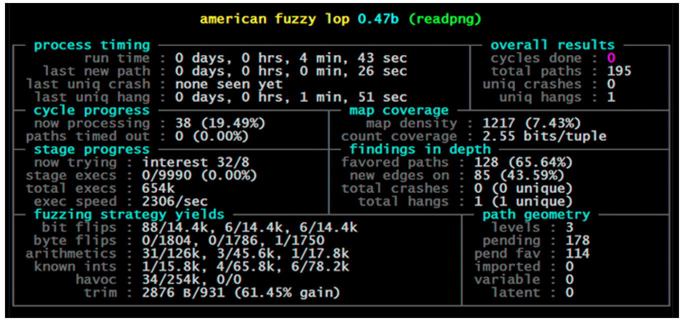
Georgia Institute of Technology

V-Fuzz: Evoluationay Fuzzing Assisted By Vulnerability Prediction

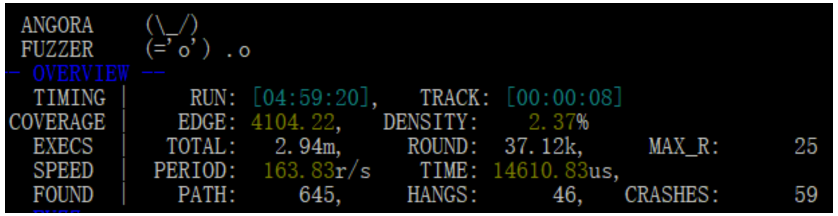
Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen
Qinchen Gu, Chunming Wu, Raheem Beyah

What is fuzzing & fuzzer?

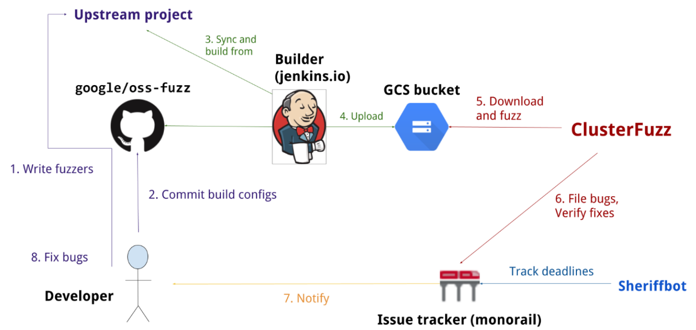
- Fuzzing is an automatic, dynamic vulnerability detection technique.
- Fuzzing detects vulnerabilities by iteratively and randomly feeding inputs to the target programs.
- Fuzzer is a tool that implements fuzzing process.



AFL



Angora



Google OSS-Fuzz

Types of fuzzers

Exploration Strategy

- Directed
  - AFLGo,
  - Hawkeye
- Coverage-based
  - AFL
  - AFLFast
  - Angora

Input Generation

- Mutation-based
  - AFL
  - AFLFast
- Grammar-based
  - Quickfuzz,
  - Peach

Target Program

- Whitebox fuzzer
  - Driller
- Greybox fuzzer
  - AFL,
  - AFLFast
- Blackbox fuzzer
  - zzuf

Coverage-based fuzzers may not so efficient

Coverage-based fuzzer is one of the most popular fuzzer.

-AFL, VUzzer, Angora, honggfuzz, t-fuzz, Driller, ...

Its goal is to cover as much as coverage of the target program as possible.

Coverage-based fuzzers may not so efficient

Coverage-based fuzzer is one of the most popular fuzzer.

-AFL, VUzzer, Angora, honggfuzz, t-fuzz, Driller, ...

Its goal is to cover as much as coverage of the target program as possible.

However, it is not efficent for detecting vulnerabilities:

1. Vulnerable code only takes a tiny fraction of the entire code.

e.g., Only 3% of the source code files in Mozilla Firefox has vulnerabilities.

2. Achieving high coverage is still very diffcult.

e.g., Driller can only generate valid inputs for 13 out of 41 CGC binaries.

Coverage-based fuzzers may not so efficient

```
#define SIZE 1000
int main(int argc, char **argv){
    unsigned char source[SIZE];
    unsigned char dest[SIZE];
    char *input=ReadData(argv[1]);
    int i;
    i=argv[2];
    /*magic byte check */
    if(input[0]!='*')
        return ERROR;
    if(input[1]==0xAB && input[2]==0xCD){
        printf("Pass␣1st␣check!\n");
        /*some nested condition*/
        if(strncmp(&input[6],"abcde",5)==0){
            printf("Pass␣2nd␣check!\n");
            /* some common codes without vulnerabilities*/
            ...
        }
        else{
            printf("Not␣pass␣the␣2nd␣check!");
        }
    }
    else{
        printf("Not␣pass␣the␣1st␣check!");
    }
    /*A buffer overflow vulnerability*/
    func_v(input, source, dest);
    return 0;
}
```

secure code

most coverage
less important

vulnerable code

less coverage
more important

Coverage-based fuzzers may not so efficient

```
#define SIZE 1000
int main(int argc, char **argv){
    unsigned char source[SIZE];
    unsigned char dest[SIZE];
    char *input=ReadData(argv[1]);
    int i;
    i=argv[2];
    /*magic byte check */
    if(input[0]!='*')
        return ERROR
    if(input[1]==0xA
        printf("Pass
        /*some nested
        if(strncmp(&
            printf("
            /* some c
            ...
        }
        else{
            printf("
        }
    }
    else{
        printf("Not pass the 1st check!
    }
    /*A buffer overflow vulnerability*/
    func_v(input, source, dest);
    return 0;
}
```

Thus, fuzzer should prioritize the potentially vulnerable code.

most coverage
less meaning

vulnerable code

less coverage
more importance

V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

Detect which are vulnerable.

Prioritize to fuzz vulnerable code.

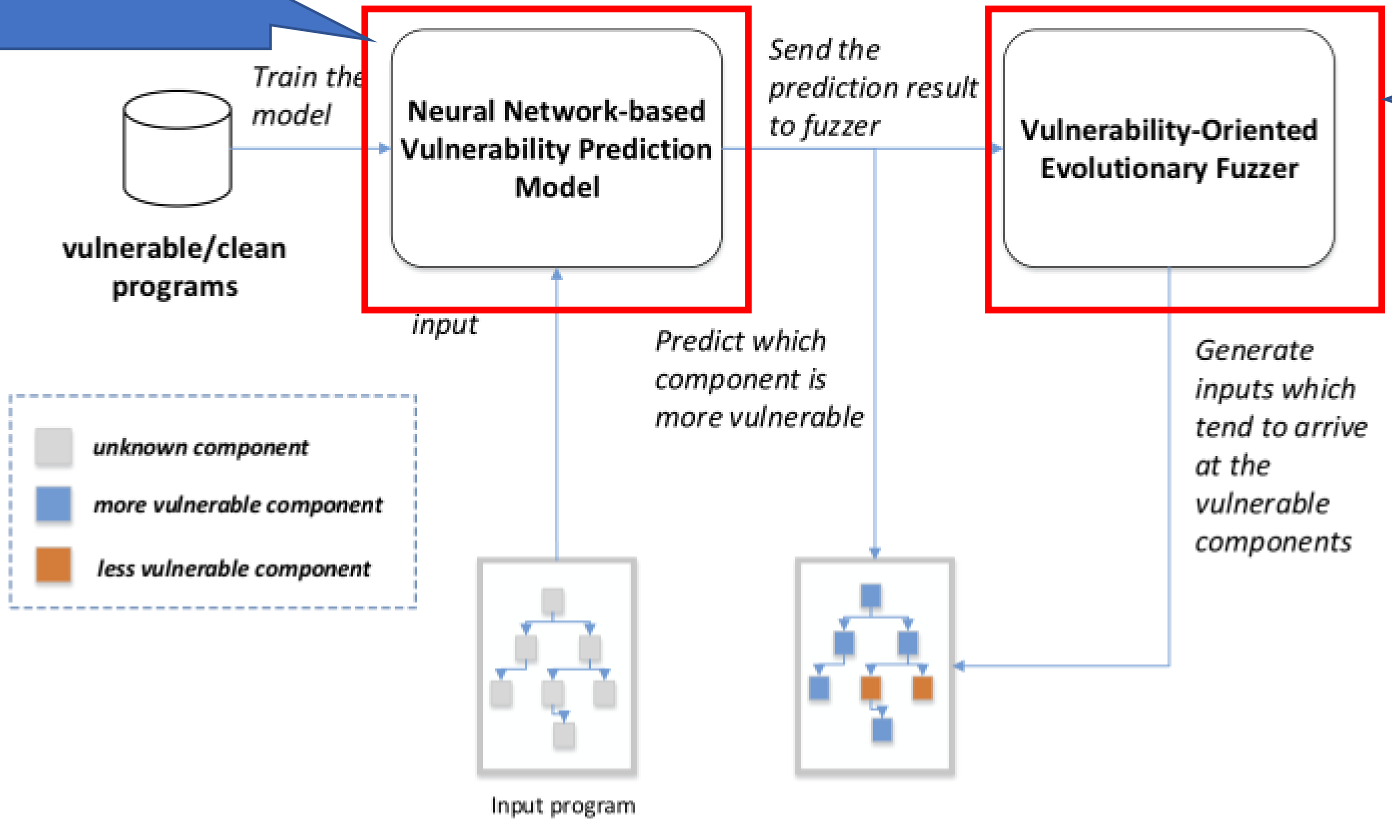


Fig. 2: The architecture of V-Fuzz.

V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

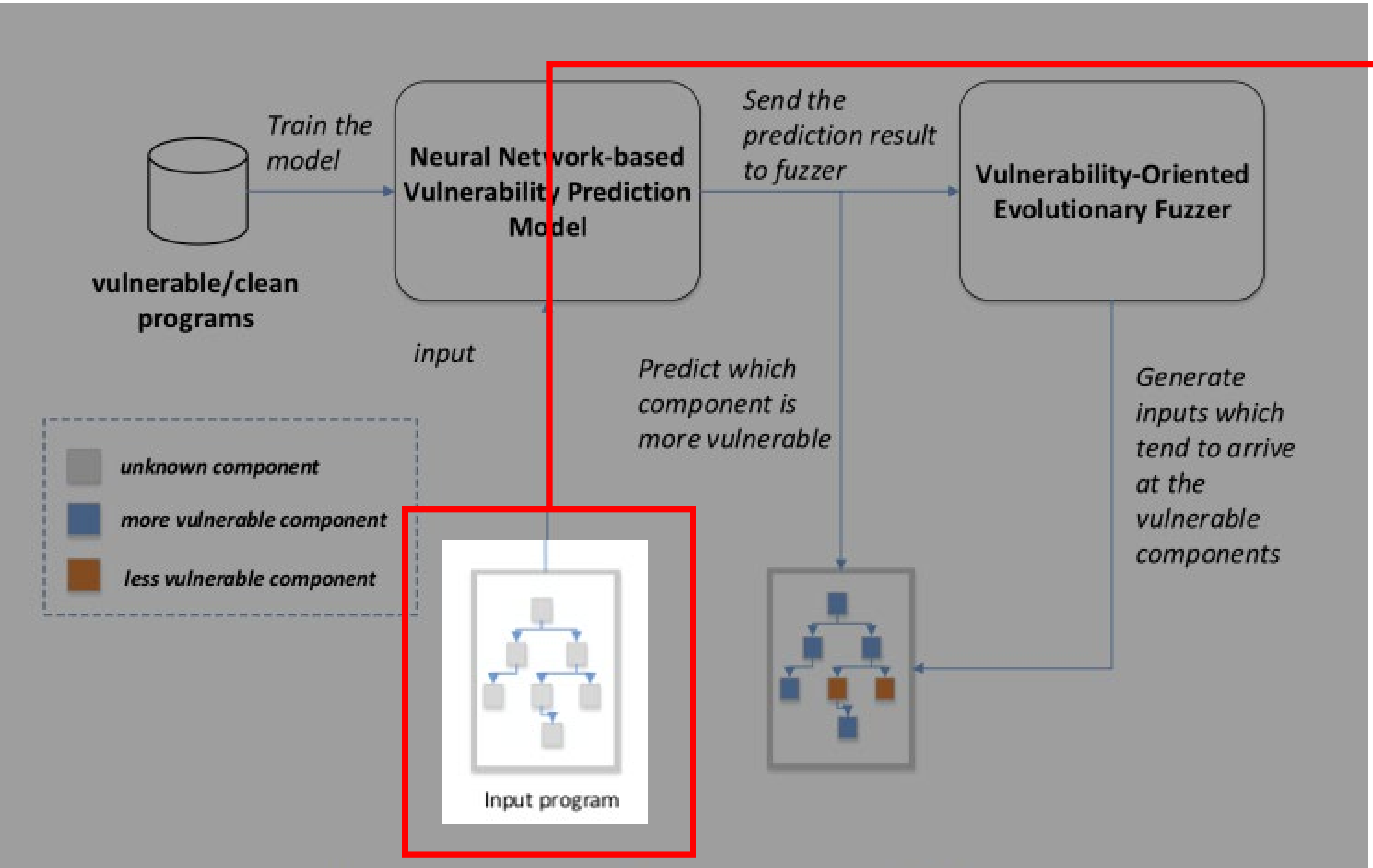
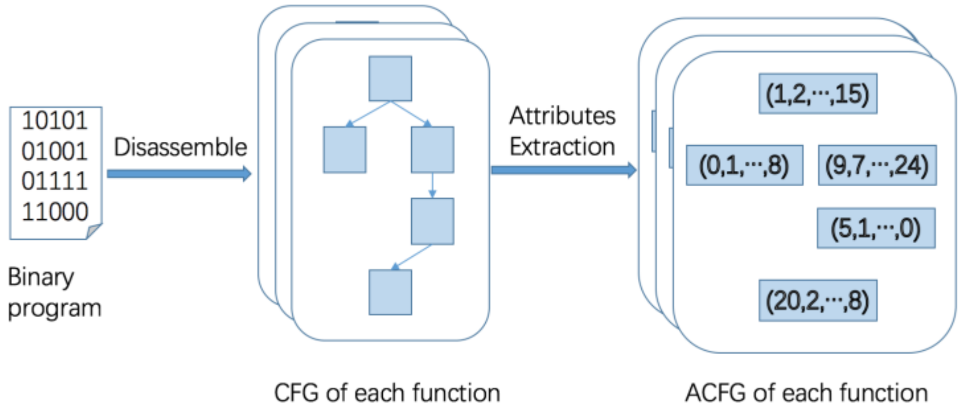


Fig. 2: The architecture of V-Fuzz.

Data preprocessing



Binary to ACFG

V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

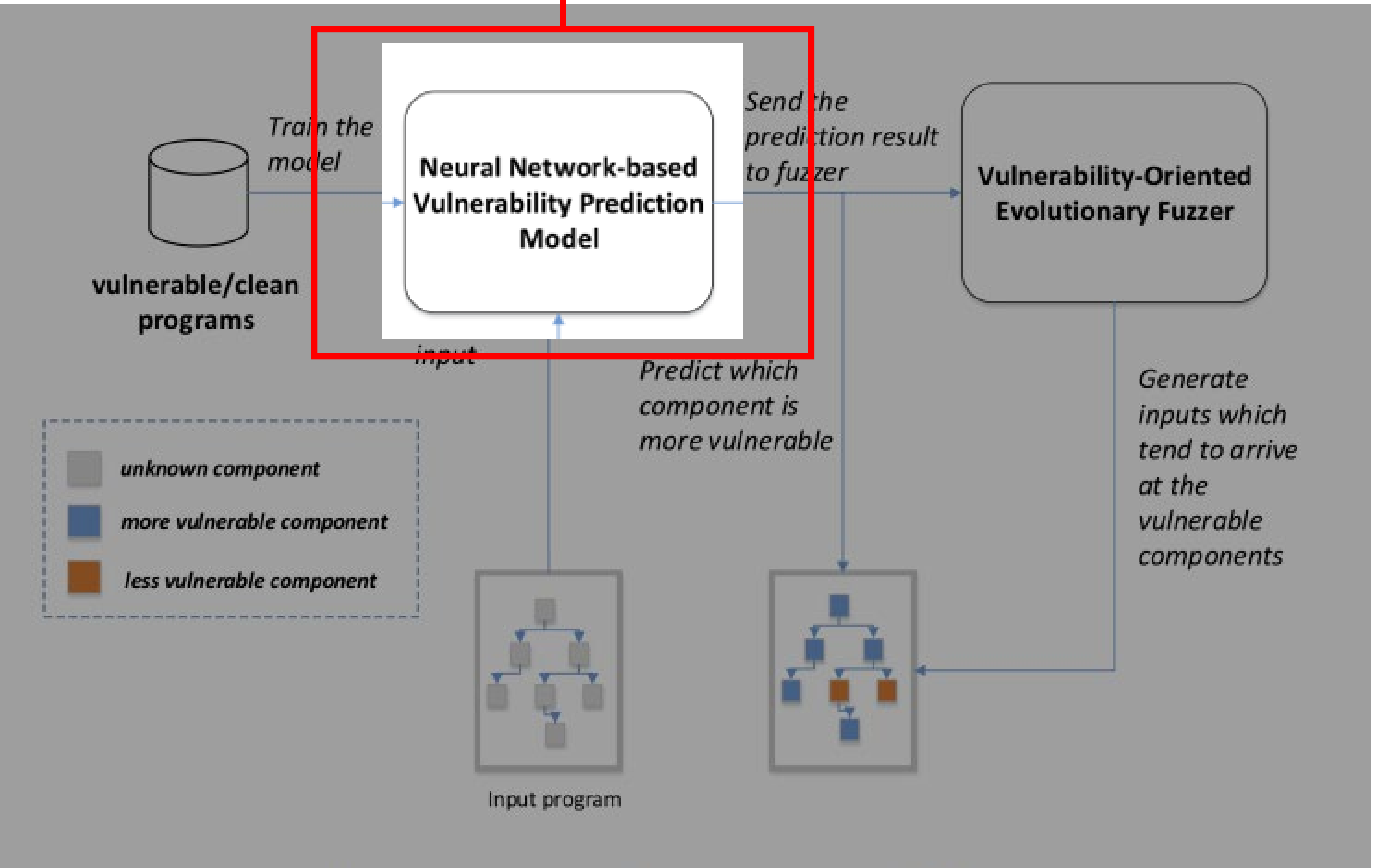
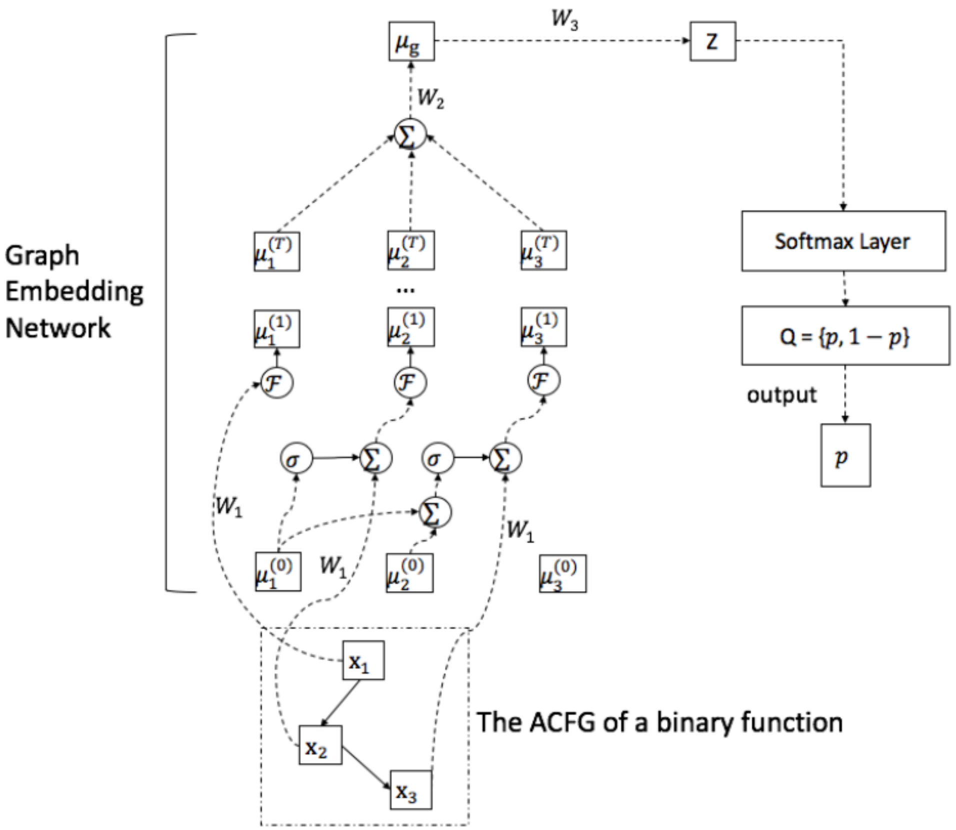


Fig. 2: The architecture of V-Fuzz.

Vulnerability Prediction Model



Output vulnerable probability

V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

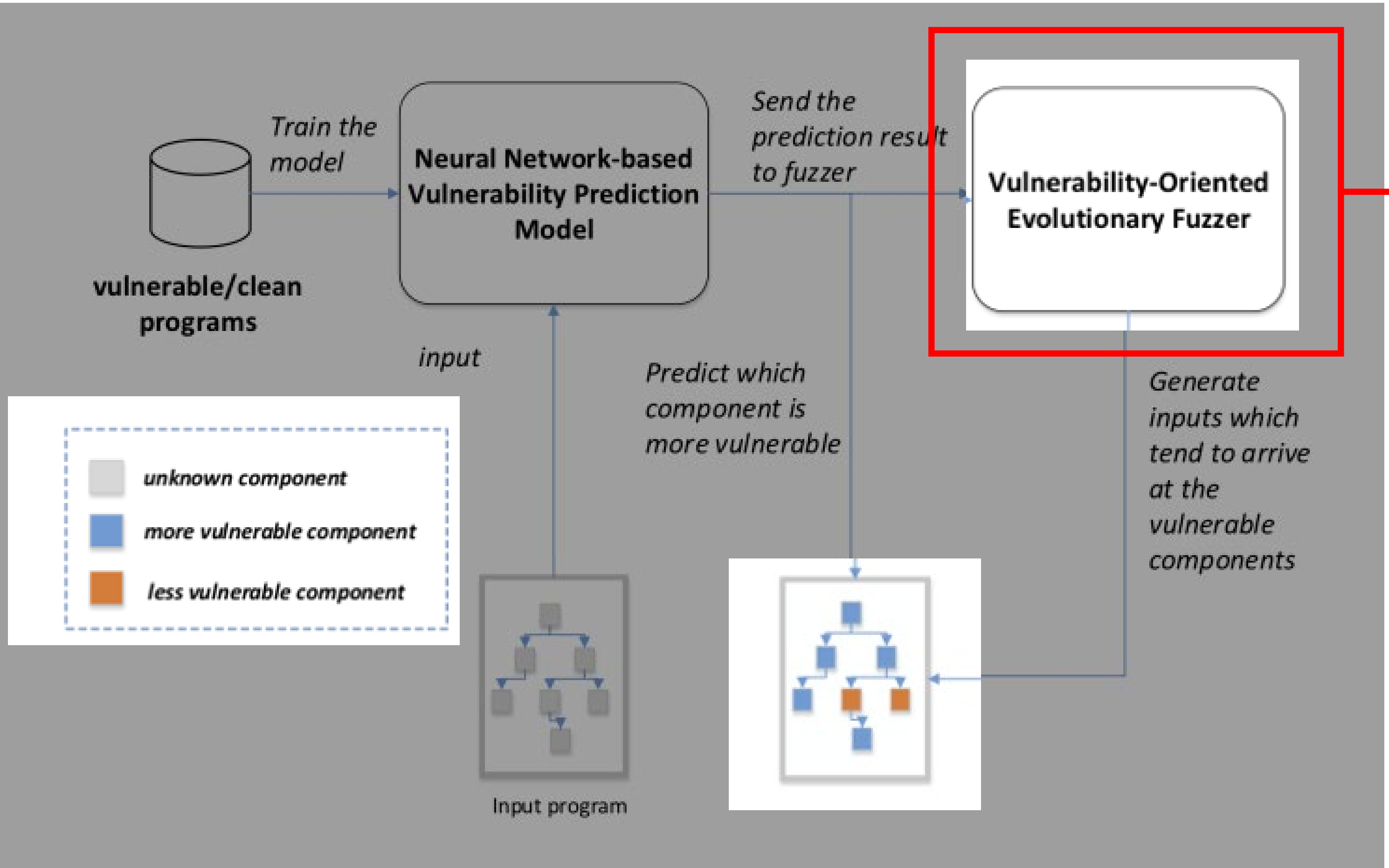


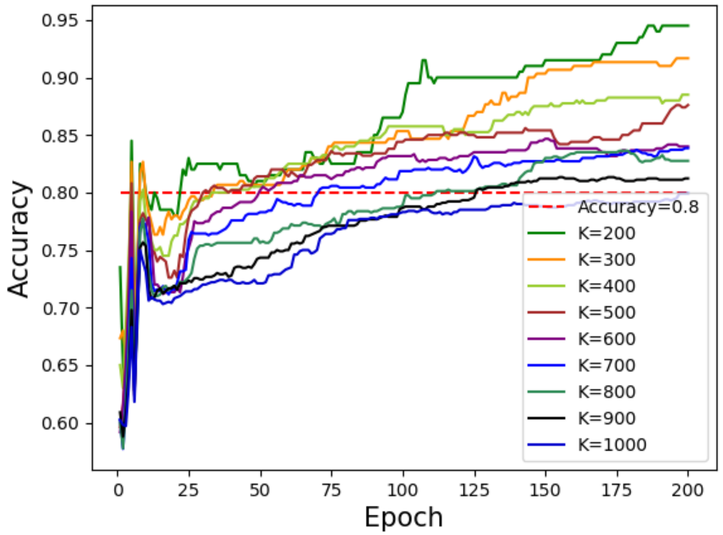
Fig. 2: The architecture of V-Fuzz.

Data preprocessing

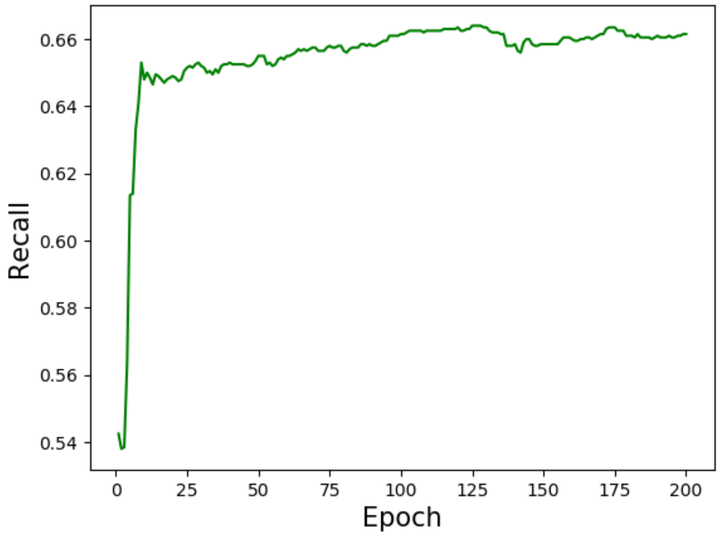
Generate inputs with tend to arrive at the vulnerable components

Experimental Results: Vulnerability Prediction

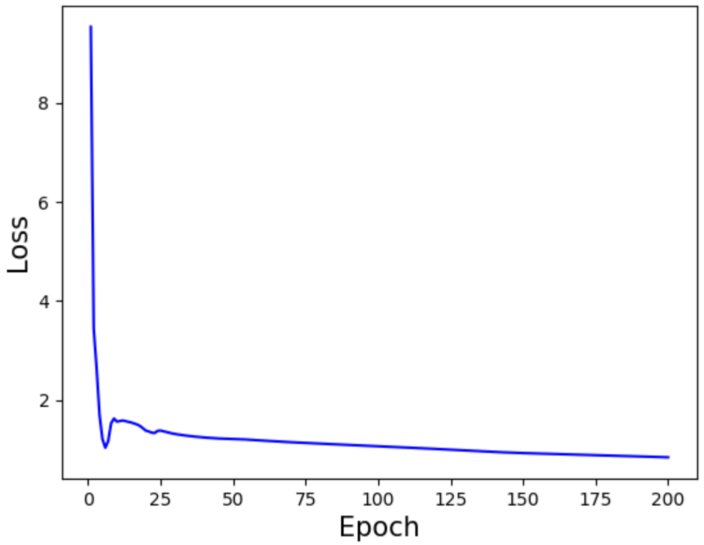
Top-K Accuracy >= 80%



Recall >= 66%

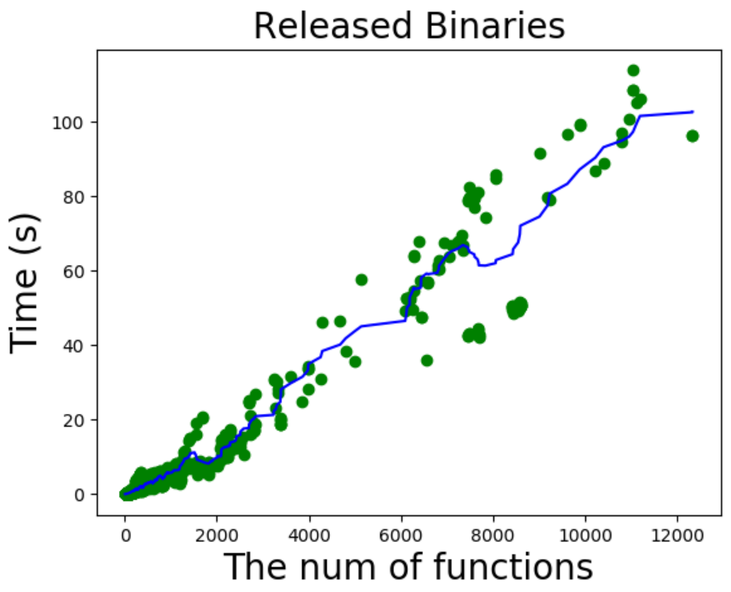


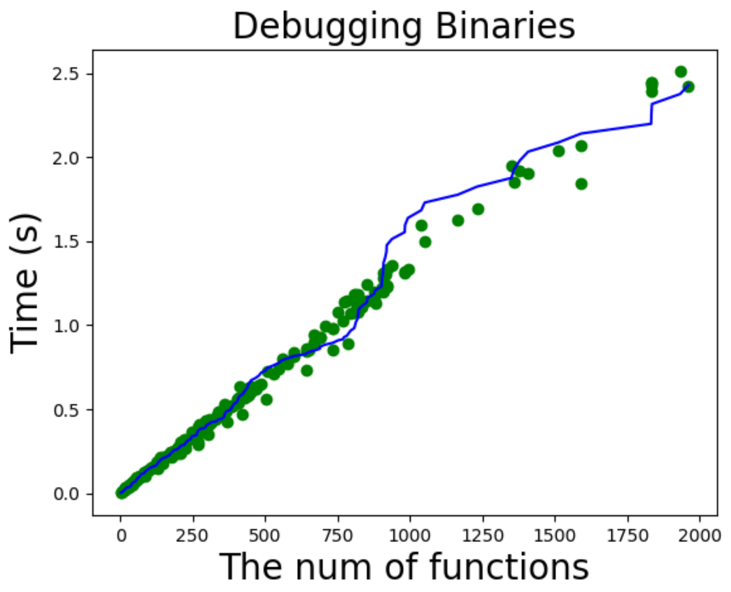
Fast Convergence



Vulnerability prediction model performs well.

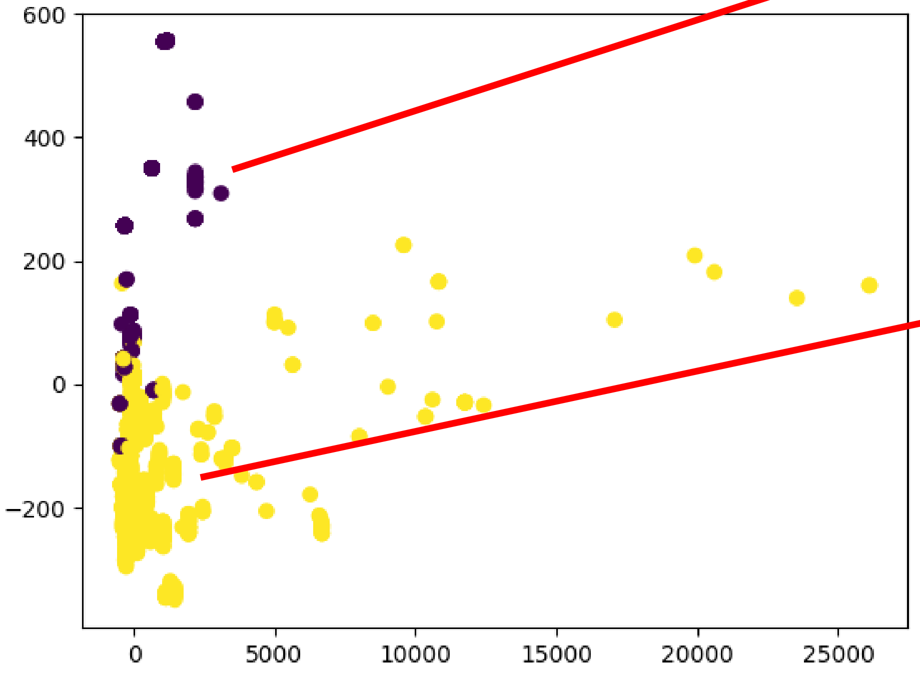
Experimental Results: ACFG Extraction Time





ACFG extraction process is fast:
Released binaries: <100 s
Debugging binaries: <2.5 s

t-SNE of graph embedding vectors

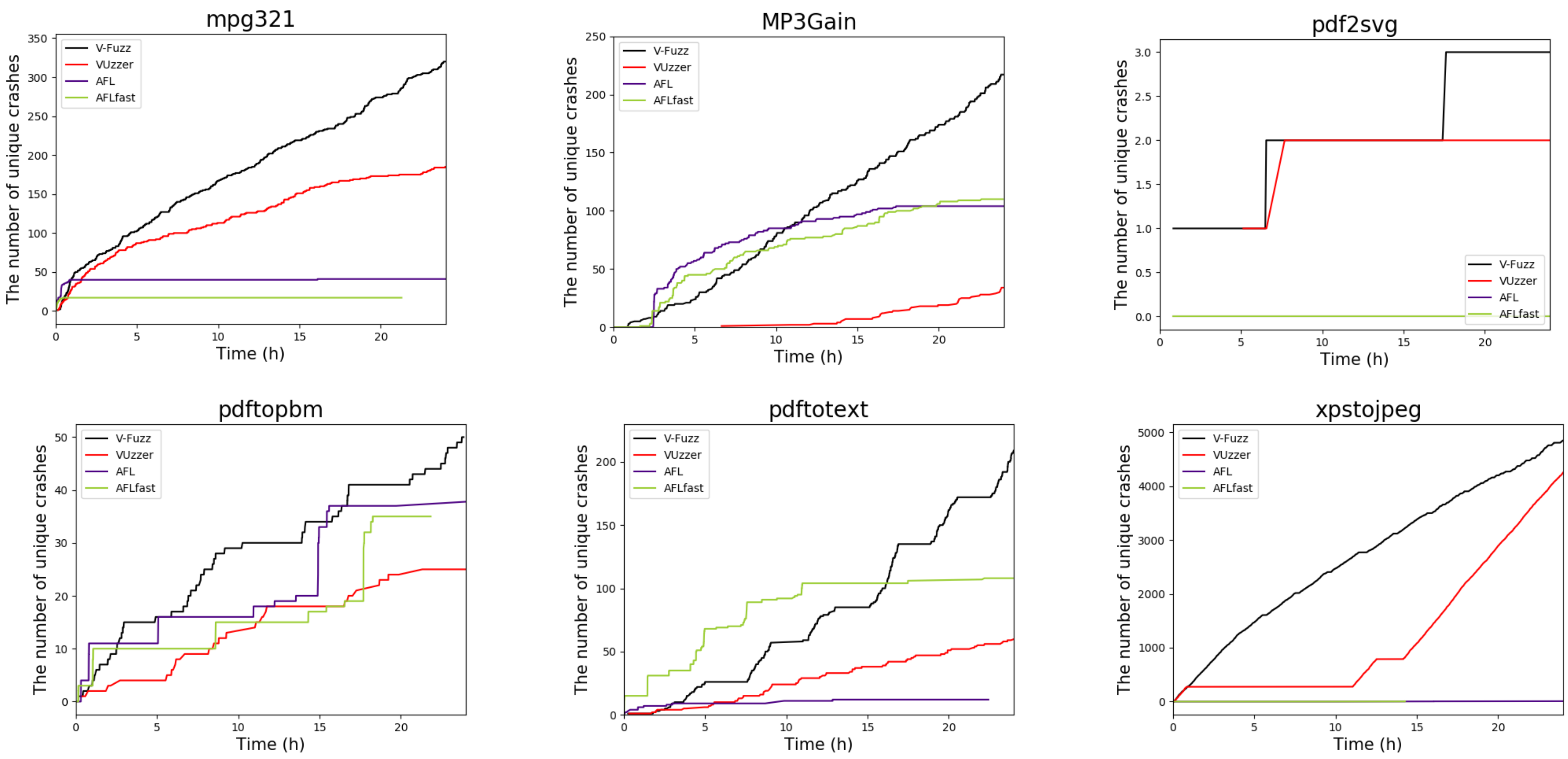


vulnerable binary functions

secure binary functions

Vulnerability prediction model can distinguish between vulnerable functions and secure functions.

Experimental Results: The number of unique crashes.



V-Fuzz finds crashes quickly than AFL, AFLFast and VUzzer.

Experimental Results: The number of unique crashes.

TABLE 6: The number of unique crashes found for 24 hours.

Application	Version	Fuzzer			
		V-Fuzz	VUzzer	AFL	AFLFast
uniq	LAVA-M	659	321	0	0
base64	LAVA-M	128	100	0	0
who	LAVA-M	117	92	0	0
pdftotext	xpdf-2.00	209	59	12	108
pdffonts	xpdf-2.00	581	367	13	0
pdftopbm	xpdf-2.00	50	25	37	35
pdf2svg + libpoppler	pdf2svg-0.2.3 libpoppler-0.24.5	3	2	0	1
MP3Gain	1.5.2	217	34	103	110
mpg321	0.3.2	321	184	40	17
xpstopng	libgxps-0.2.5	3,222	2,195	2	2
xpstops	libgxps-0.2.5	4,157	3,044	3	3
xpstojpeg	libgxps-0.2.5	4,828	4,243	4	4
cflow	1.5	1	0	0	0
Total		14,493	10,666	214	280
Average		1,114	820	16	21

V-Fuzz finds more unique crashes than AFL, AFLFast and VUzzer.

Experimental Results: CVEs

TABLE 9: The CVEs found by V-Fuzz.

Application	Version	CVE	Vulnerability Typye
pdftotext pdffonts pdftopbm	xpdf<=3.01	CVE-2007-0104	Buffer errors
mpg321	0.3.2	CVE-2017-11552	Buffer errors
MP3Gain	1.5.2	CVE-2017-14406 CVE-2017-14407 CVE-2017-14409 CVE-2017-14410 CVE-2017-12912	NULL pointer dereference Stack-based buffer over-read Buffer overflow Buffer over-read Buffer errors
libgxps	<=0.3.0	CVE-2018-10767 (new) CVE-2018-10733 (new)	Buffer errors Stack-based buffer over-read
libpoppler	0.24.5	CVE-2018-10768 (new)	NULL pointer dereference

V-Fuzz detects three new CVEs.

Conclusion

- **In this paper, we design and implement V-Fuzz, an evolutionary fuzzer assisted by vulnerability prediction.**
- **We design and implement a vulnerability prediction model based on graph embedding network that can predict the vulnerable probabilities for binary functions.**
- **Compared with several state-of-the-art fuzzers, V-Fuzz can find more vulnerabilities quickly.**
- **V-Fuzz has discovered 10 CVEs, and 3 of them are 0-day vulnerabilities.**