VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search

NDSS 2023

Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang,
Wei Xie, Xu Zhou, Danjun Liu, Kai Lu

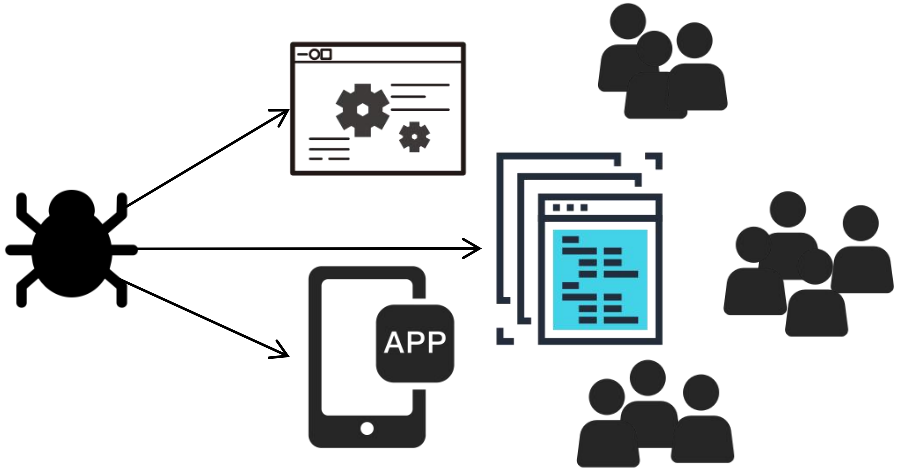
National University of Defense Technology

Contact:zh.luo@nudt.edu.cn

CONTENTS

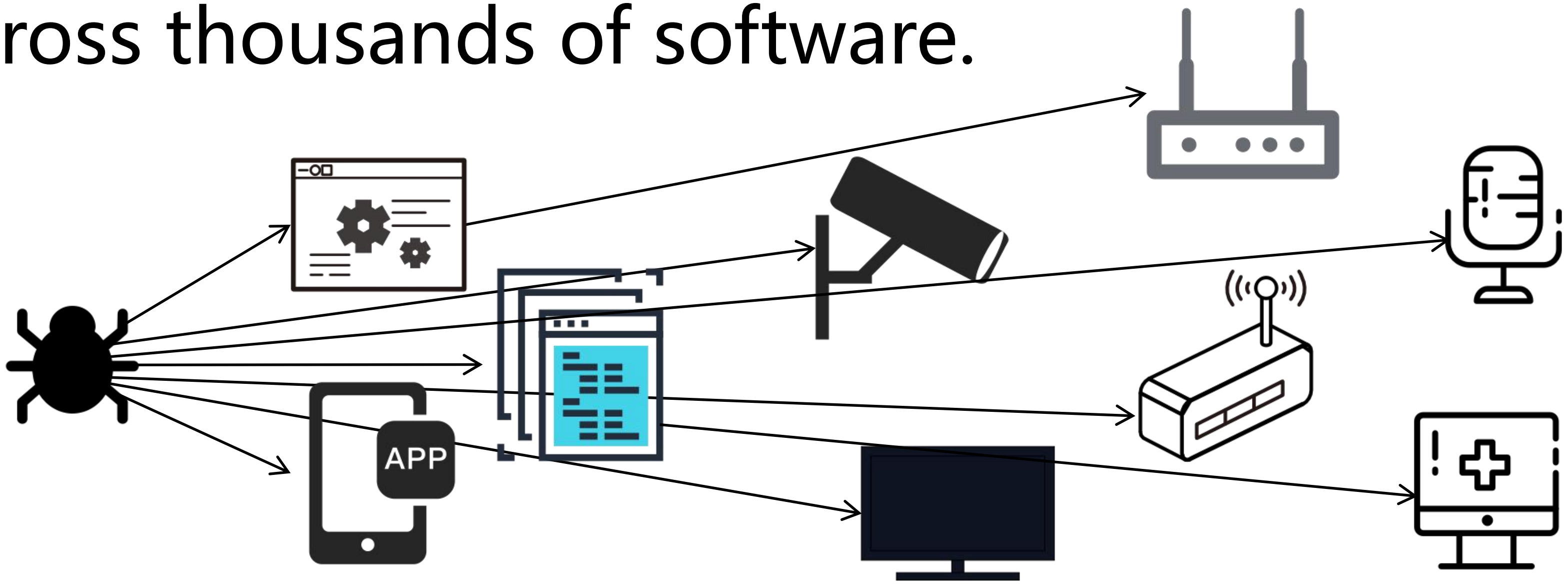
1.Background

- **Code reuse is widespread in software development.**
  - Third-party libraries are reused without secuity audit.
  - A single vulnerability in the open-source code may spread across thousands of software.



1.Background

- **Code reuse is widespread in software development.**
  - Third-party libraries are reused without secuity audit.
  - A single vulnerability in the open-source code may spread across thousands of software.



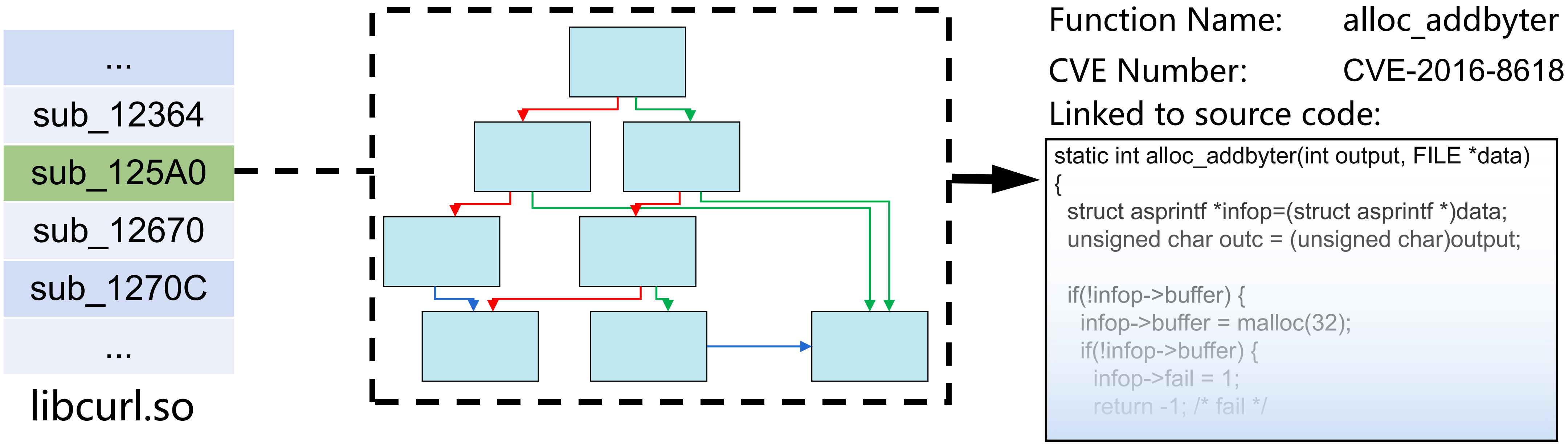
  - With the wide deployment of the IoT, the harms of code reuse are magnified.

1.Background

- **IoT firmware images and software:**
  - Firmware and software are usually only available in binaries.
    - No function names.
    - No symbol information
  - Vulnerabilities from the same source code may differ in various binary files.
- **Binary code search becomes an active research for seeking vulnerabilities hidden in firmware and software.**

1.Background

- **Binary Code Search:**
  - Find similar or homologous binary functions.
    - Given a binary file, the binary code search compares its functions with all functions in the vulnerability repository based on function similarity.



1.Background

- **However**
  - Firmware and software are compiled with various compilation settings.
  - This makes functions from the same source code may differ in binary format.

For example:

```
base64_encode_alloc (…){
  size_t outlen = 1 + BASE64_LENGTH (inlen);
  if (inlen > outlen){
      *out = NULL;
      return 0;
  }
  *out = malloc (outlen);
  if (!*out) return outlen;
  base64_encode (in, inlen, *out, outlen);
  return outlen - 1;
}
```

Coreutils base64.c

```
PUSH    {R4-R8,LR}
MOV     R6, R2
ADD     R4, R1, #2
LDR     R3, 0xAAAAAAAB
UMULL   R2, R4, R3, R4
MOV     R4, R4,LSR#1
MOV     R4, R4,LSL#2
ADD     R8, R4, #1
CMP     R1, R8
MOVHI   R4, #0
STRHI   R4, [R6]
BLS     loc_1208C
•••
```

ARM, GCC, O1

```
push  r15
push  r14
push  r13
push  r12
push  rbx
mov   r12, rdx
lea   rax, [rsi+2]
mov   rcx, 0AAAAAAAAAAAAAAABh
mul   rcx
mov   rbx, rdx
add   rbx, rdx
and   rbx, 0FFFFFFFFFFFFFFFCh
•••
```

x86, Clang, O3

```
addiu   $sp, -0x28
sw      $ra, 0x20+var_s4($sp)
sw      $fp, 0x20+var_s0($sp)
move    $fp,$sp
li      $gp, 0x4260D0
sw      $gp, 0x20+var_10($sp)
sw      $a0, 0x20+arg_0($fp)
sw      $a1, 0x20+arg_4($fp)
sw      $a2, 0x20+arg_8($fp)
lw      $v0, 0x20+arg_4($fp)
addiu   $v1,$v0, 2
li      $v0, 0xAAAAAAAB
•••
```

MIPS, GCC, O0

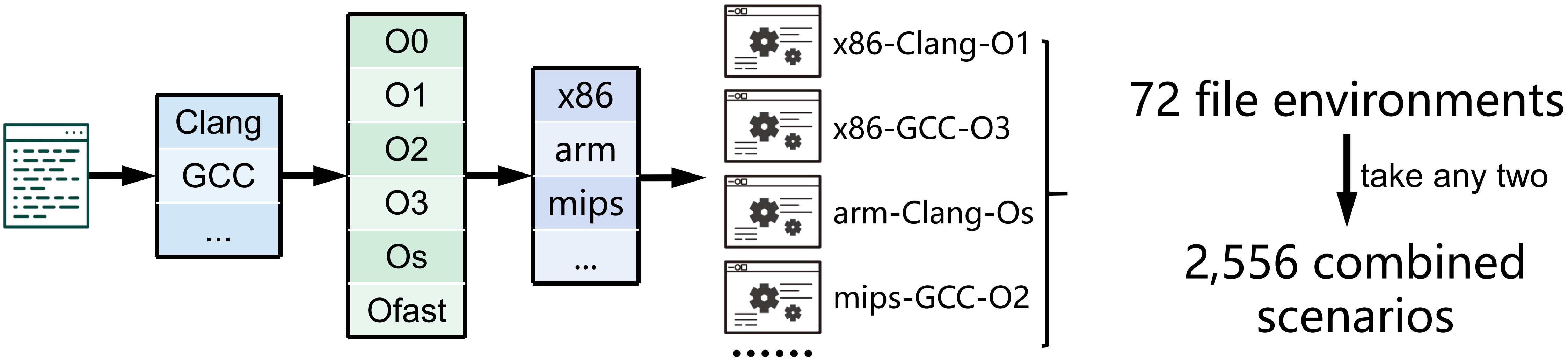
2.Motivation

- **Challenge 1:**
  - Require binary code search methods robust across ISAs.
    - Different registers
    - Different opcodes
  - Mono-architecture methods are not suitable
    - e.g., Asm2Vec (S&P 2019), DeepBinDiff (NDSS 2020), PalmTree (CCS 2021).
  - Out-of-vocabulary (OOV) issues, e.g.,
    - SAFE (DIMVA 2019) uses a normalization.
      - Still many OOV issues
    - InnerEYE (NDSS 2019) uses a neural machine translation.
      - When unseen ISA binary file comes, they become weak.

2.Motivation

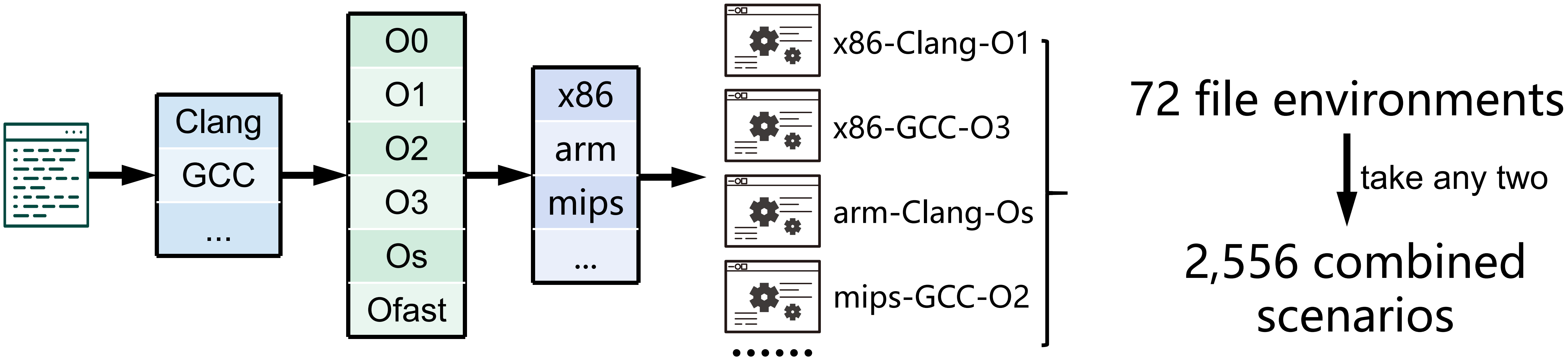
- **Challenge 2:**
  - Binaries are compiled with various compilation settings.
    - Different compilers (e.g., Clang and GCC).
    - Different optimization levels (e.g., O0, O1, O2, O3, Os, Ofast)
    - Different architectures (e.g., *x86*, *arm*, and *mips*).

File environment: the combination of {compiler, architecture, optimization level}.



2.Motivation

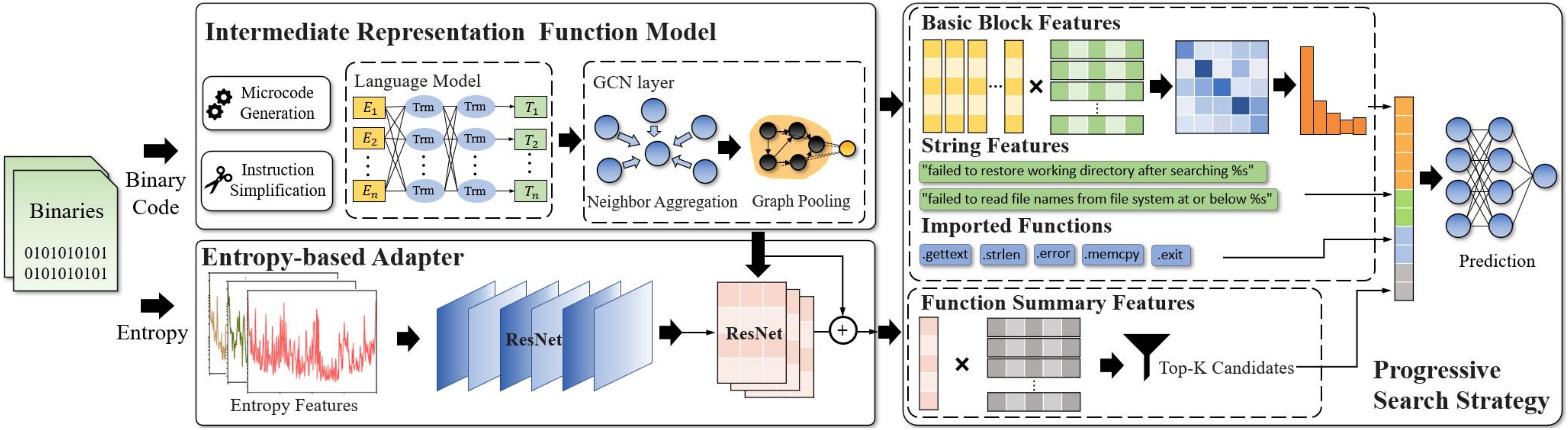
- **Challenge 2:**
  - Binaries are compiled with various compilation settings.



  - Trex, Gemini, and jTrans try to use a single deep learning model to build a robust model for this complex problem.
    - Robust against one or several specific scenarios possible
    - Robust against 2,556 scenarios complicated

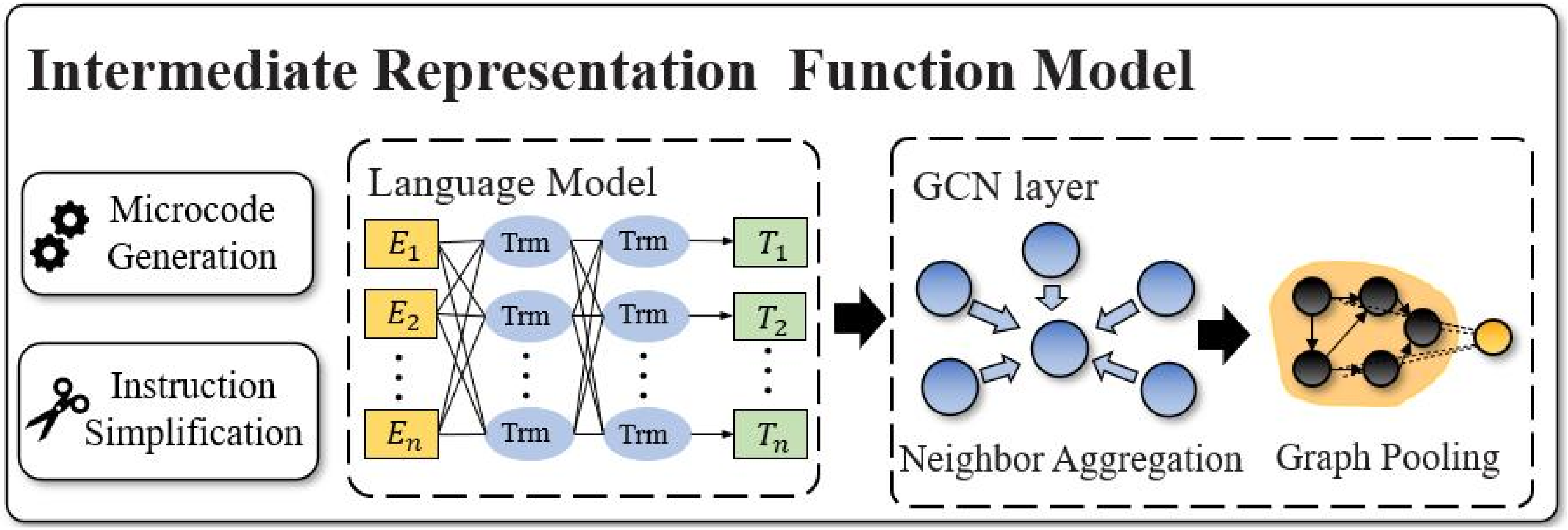
3.Design

- **To solve these Challenges, we propose a cross-architecture binary code search approach named VulHawk:**
  - Intermediate representation function model (IRFM) Challenge 1
  - Entropy-based adapter Challenge 2
  - Progressive search strategy



3.Design

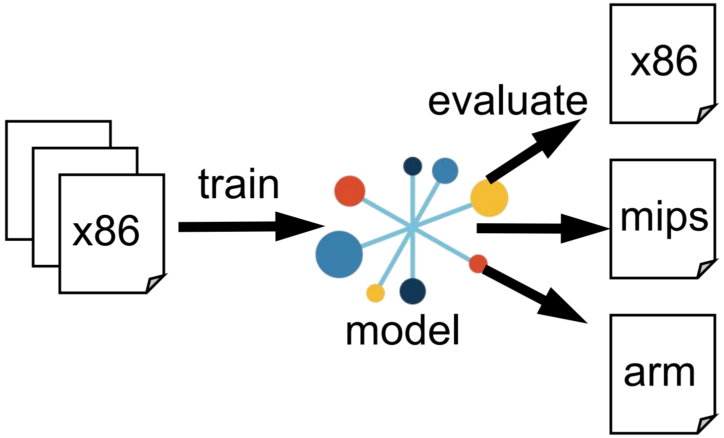
- **Intermediate Representation Function Model (IRFM)**
  - Purpose: generate function semantic embeddings.
- Microcode Generation
- Instruction Simplification
- Language Model
- GCN layer



3.Design

- **Intermediate Representation Function Model (IRFM)**
  - Microcode Generation:
    - Lift binary code to an architecture-agnostic IR (Microcode)
    - Microcode groups any instructions from different ISAs into 73 opcodes and 16 types of operands.
    - This can mitigate the impacts of instruction type differences. Challenge 1
    - It allows our model to be trained from one ISA and to search functions in multiple ISAs.

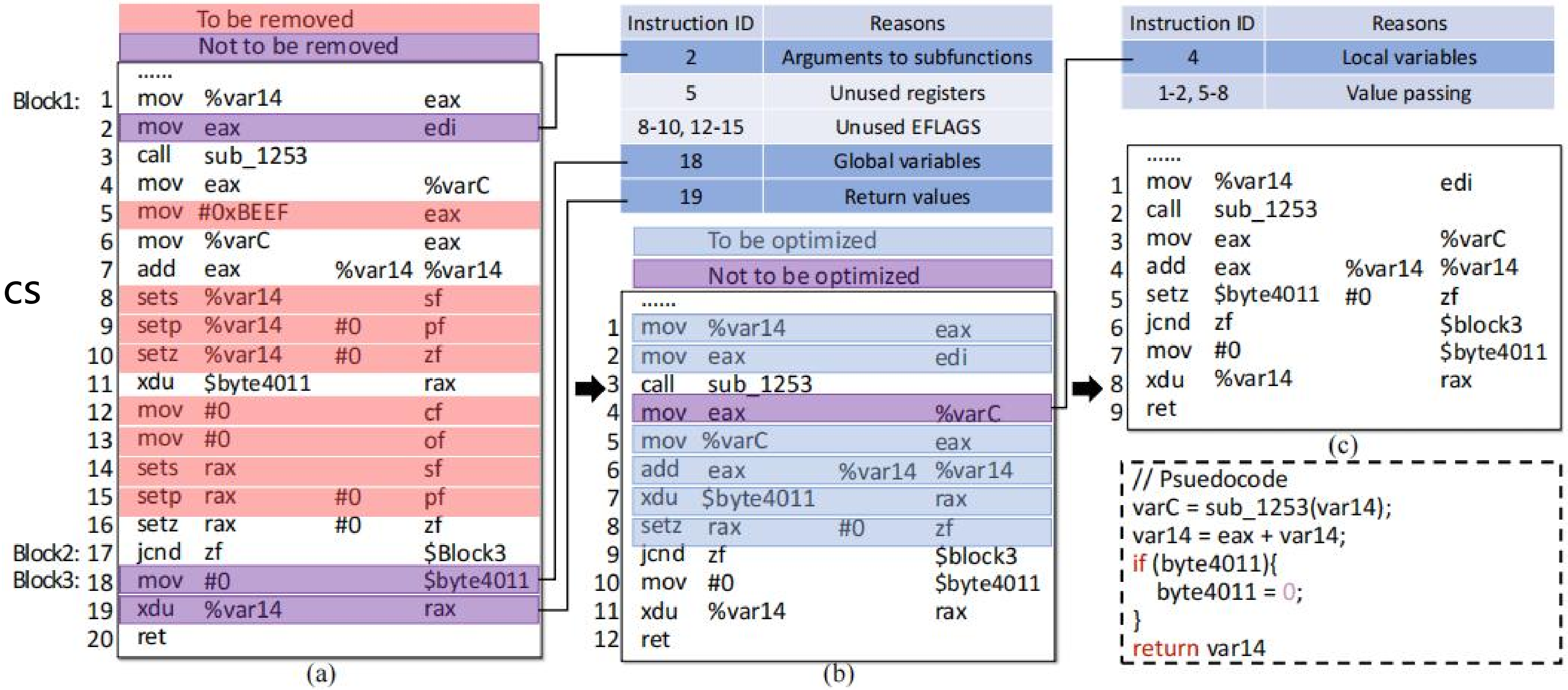
Microcode	# of types	Type list
opcode	73	nop, stx, ldx, ldc, mov, neg, lnot, bnot, xds, xdu, low, high, add, sub, mul, udiv, sdiv, umod, or, and, xor, smod, shl, shr, sar, cfadd, ofadd, cfshl, cfshr, sets, seto, setp, setnz, setz, setae, setb, seta, setbe, setg, setge, setl, setle, jcnd, jnz, jz, jae, jb, ja, jbe, jg, jge, jl, jle, jtbl, ijmp, goto, call, icall, ret, push, pop, und, ext, f2i, f2u, i2f, u2f, f2f, fneg, fadd, fsub, fmul, fdiv.
operand	16	mop_z, mop_r, mop_n, mop_str, mop_d, mop_S, mop_v, mop_b, mop_f, mop_l, mop_a, mop_h, mop_c, mop_fn, mop_p, mop_sc



- **Intermediate Representation Function Model (IRFM)**
  - Instruction Simplification:
    - Consider the used EFLAGS.
      - EFLAGS are assigned by instructions, which control the basic block conditional jumps.
    - Prune redundant instructions and preserve important semantics

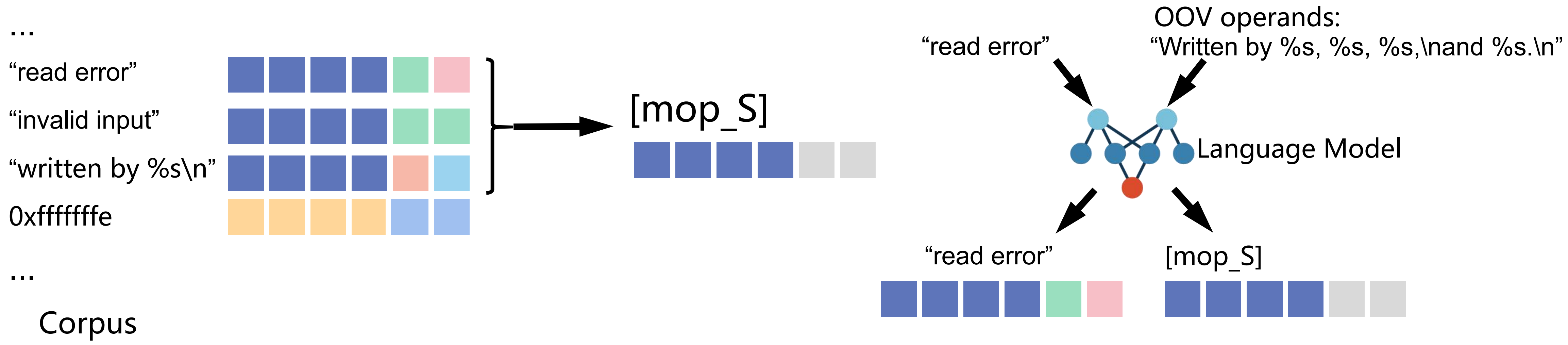
Redundant instructions:

1. occupy the limited input positions
2. reduce the weight of the main semantics
3. increase the difference of similar code



3.Design

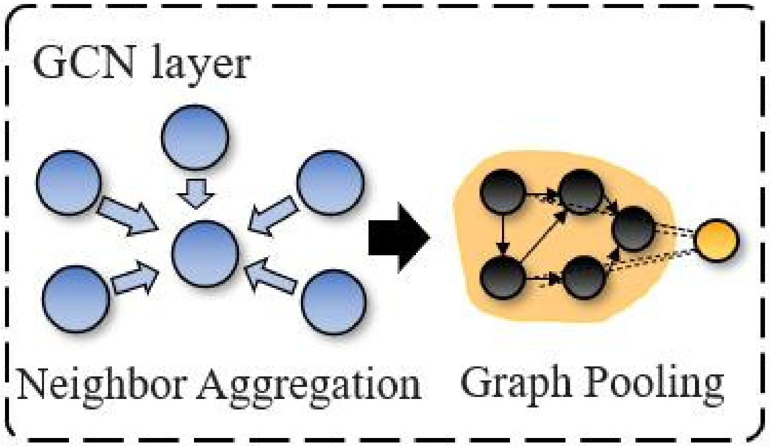
- **Intermediate Representation Function Model (IRFM)**
  - Language Model
    - Purpose: Generate basic block embeddings.
      - RoBERTa model
    - To better generate the semantics of the OOV instructions: Challenge 1
      - preserve opcodes
      - convert OOV operands into their root-operand tokens



* [mop_S] represents strings

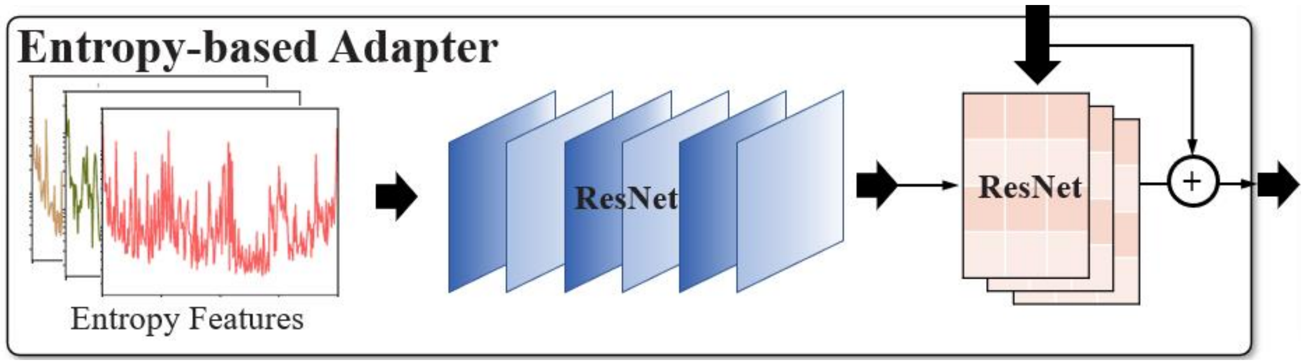
3.Design

- **Intermediate Representation Function Model (IRFM)**
  - GCN layer
    - Purpose: Generate function embeddings.
      - GCN model
      - Intergrate CFG structure and baisc block embeddings
    - We consider a CFG as a graph:
      - basic block ⟶ node
      - jump ⟶ directed edge
    - Aggregate neighbor embeddings
    - Graph pooling



3.Design

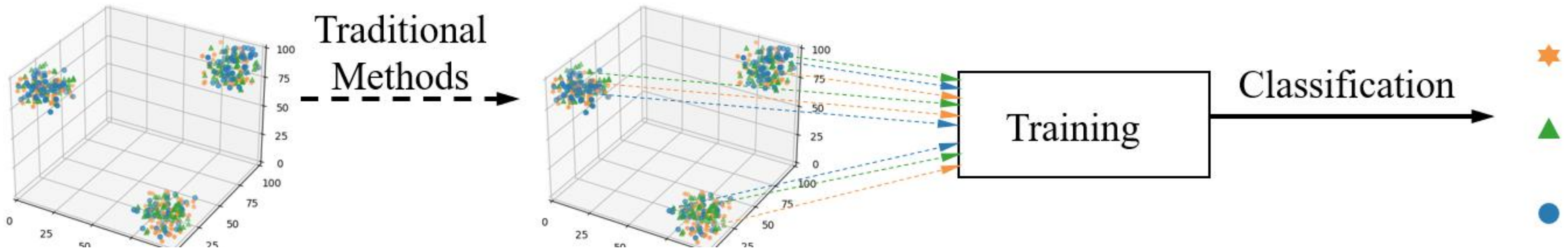
- **Entropy-based adapter** Challenge 2
  - Divide-and-conquer strategy
  - Entropy-based Binary Analysis
  - Entropy-based Adapter layer



3.Design

- **Entropy-based adapter**
  - Divide-and-conquer strategy

Matching similar functions in the embedding space:

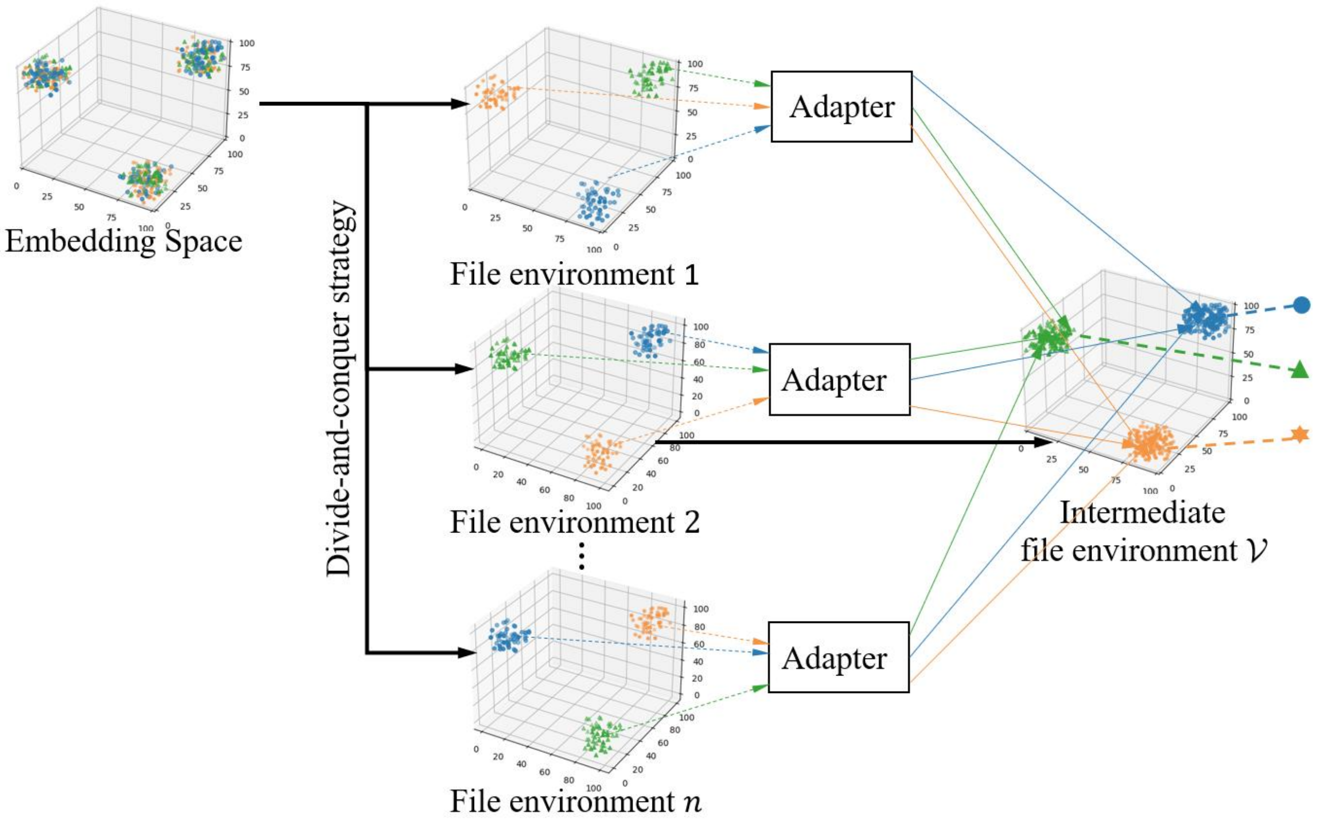


Existing methods try to use a single deep learning model to build a robust model for the similarity calculation problem with 2,556 scenarios.

e.g., the differences between O0 and O3 optimizations and the differences between GCC and Clang compilers are different.

3.Design

- **Entropy-based adapter**
  - Divide-and-conquer strategy



1. Split the mixed embedding space in to multiple sub-spaces.

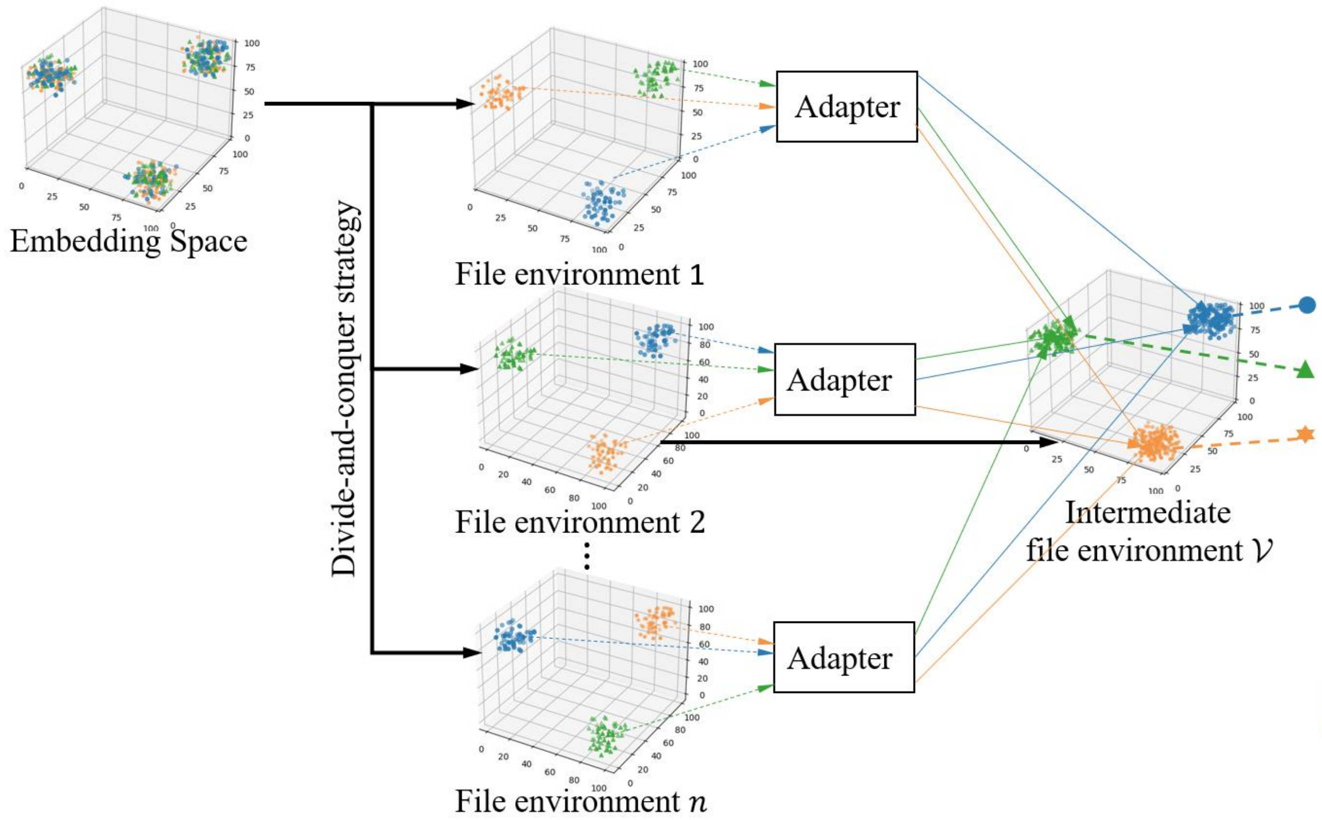
2. Select an intermediate environment $\mathcal{V}$.

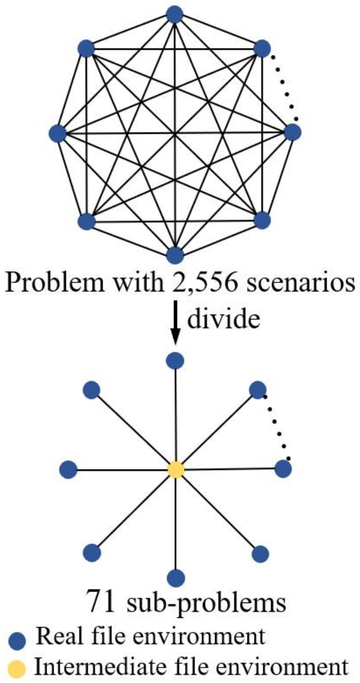
3. Divide the similarity problem among 72 file environments with 2,556 scenarios into 71 sub-problems.

4. Use adapters to transfer embeddings from different file environments into the same file environment $\mathcal{V}$ for similarity calculation.

3.Design

- **Entropy-based adapter**
  - Divide-and-conquer strategy





1. Split the mixed embedding space in to multiple sub-spaces.

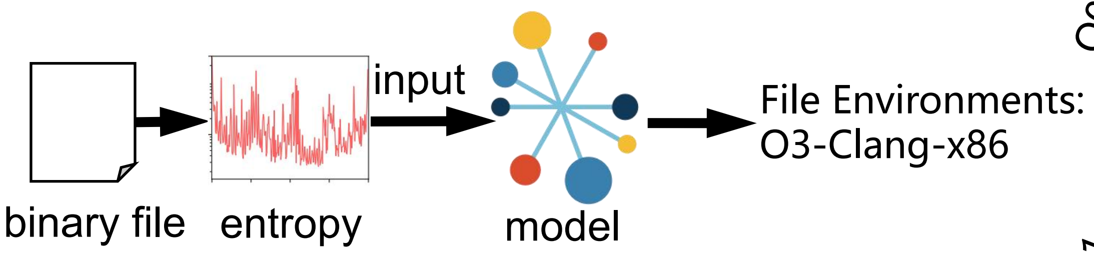
2. Select an intermediate environment $\mathcal{V}$.

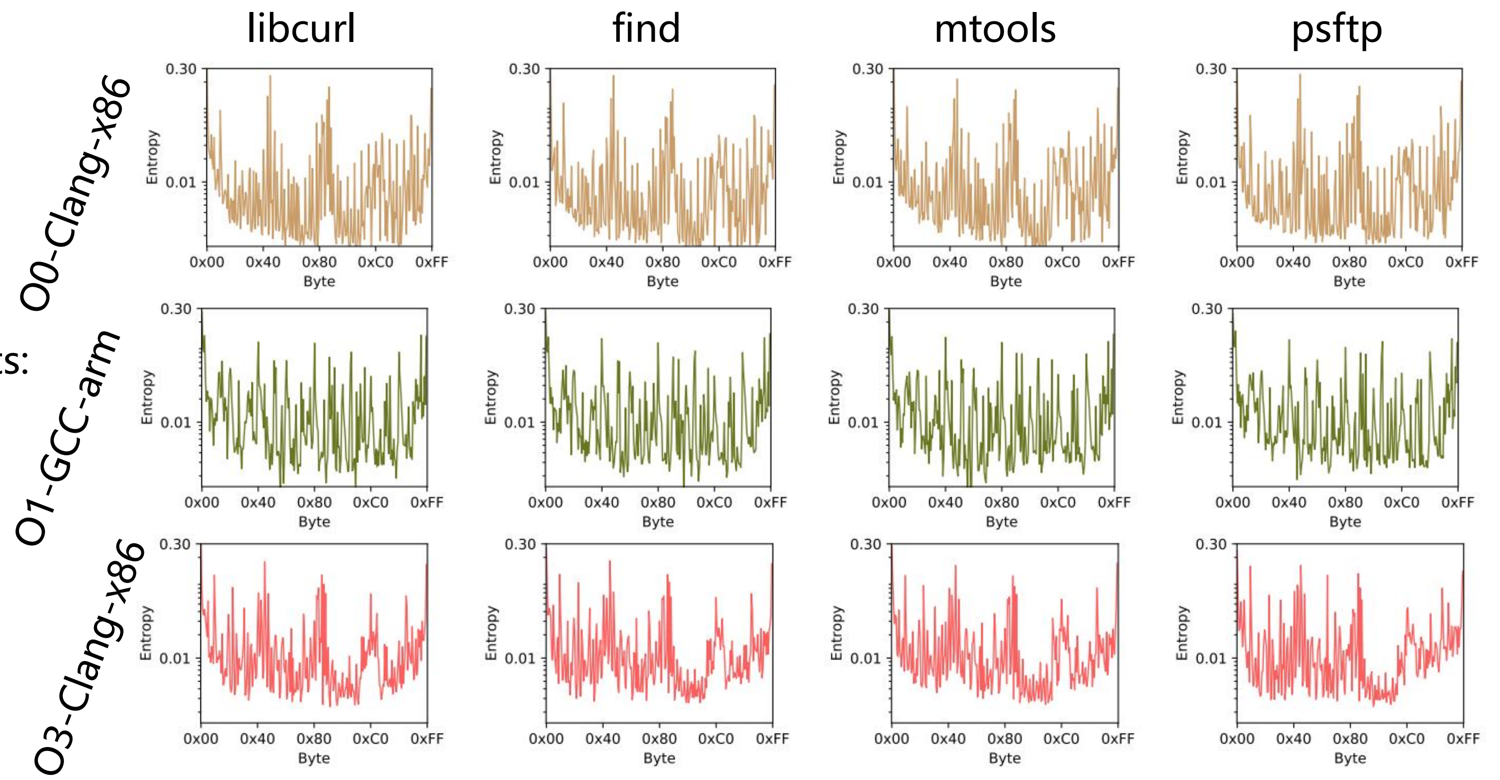
3. Divide the similarity problem among 72 file environments with 2,556 scenarios into 71 sub-problems.

4. Use adapters to transfer embeddings from different file environments into the same file environment $\mathcal{V}$ for similarity calculation.

3.Design

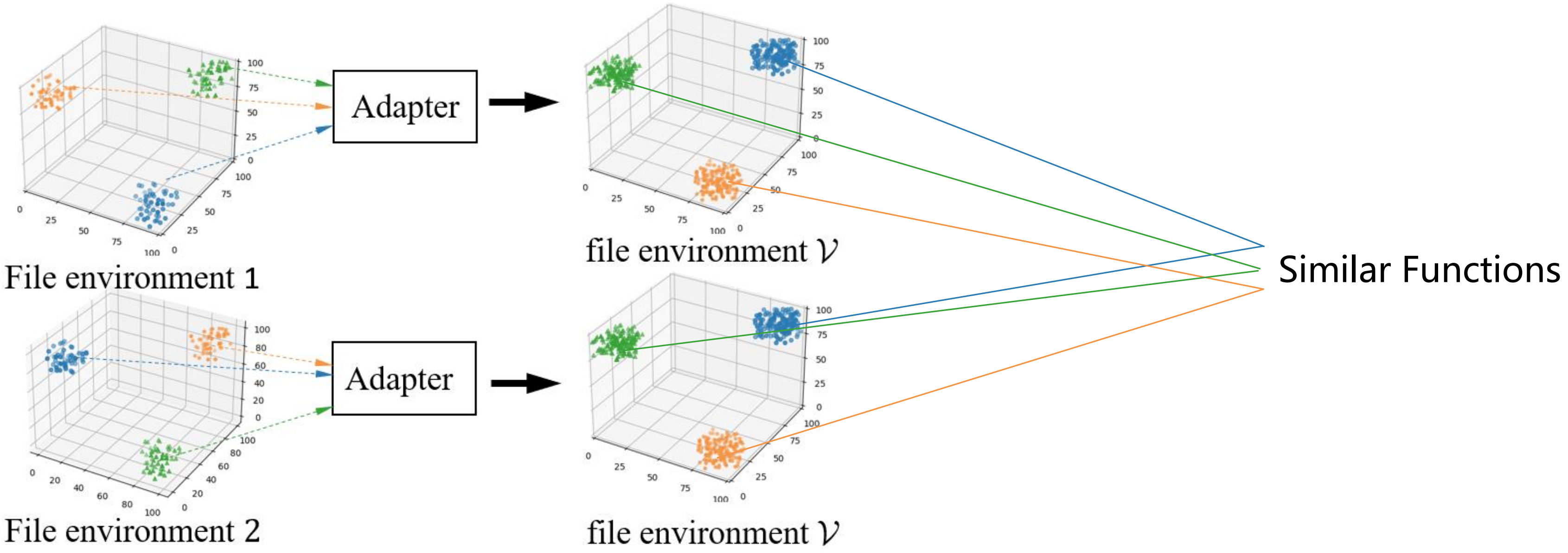
- **Entropy-based adapter**
  - Entropy-based Binary Analysis
    - Purpose: Identify the file environments.
    - Information-theoretic perspective: the more complex, the higher the entropy
      - Entropy stream
      - ResNet





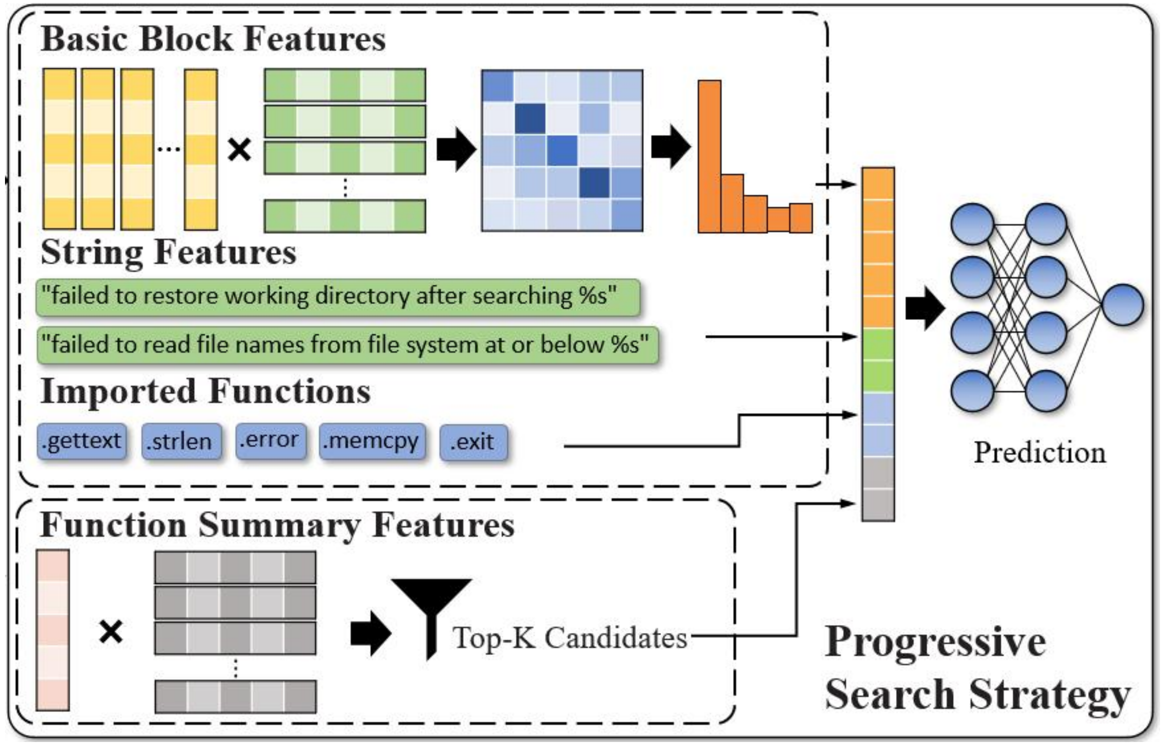
3.Design

- **Entropy-based adapter**
  - Entropy-based Adapter layer
    - Purpose: Transfer function embeddings from different file environments into the same intermediate file environment $\mathcal{V}$ to alleviate the differences caused by different file environments.
    - Adapter: ResNet model



3.Design

- **Progressive search strategy**
  - Motivation: function embeddings are coarse-grained, and fine-grained matching is time-consuming.
  - Two-step strategy
    - Function Embedding Search
      - Euclidean distance similarity
    - Similarity Calibration
      - Basic block embeddings
      - Strings
      - Imported functions
      - Function similarity



4.Evaluations

- Benchmarks:
  - 10 popular projects
  - 596,099 binary functions
  - 7 tasks

Task	XO	XA	XC	XO+XA	XO+XC	XA+XC	XO+XA+XC
compiler	×	×	◯	×	◯	◯	◯
architecture	×	◯	×	◯	×	◯	◯
optimization	◯	×	×	◯	◯	×	◯

◯ represents the function pairs with different settings for this, while × represents the function pairs with the same settings for this.

- Baselines:
  - PalmTree, SAFE, Asm2Vec, Asteria, Trex, BinDiff, GMN
  - and
    - VulHawk: the original VulHawk.
    - VulHawk-ES: replaces the entropy-based adapter with neural networks and does not use the similarity calibration.
    - VulHawk-S: VulHawk without the similarity calibration

4.Evaluations

- Research questions
  - RQ1: Given two binary functions, can VulHawk determine whether they are similar?
  - RQ2: Can VulHawk be used for searching one function in a large function repository?
  - RQ3: Can VulHawk identify how many functions are similar from two binaries?
  - RQ4: Can VulHawk detect 1-day vulnerabilities in the real world?

4.Evaluations

- Results of one-to-one comparison
  Given two functions from various file environments, the models determine their similarity
  - The AUCs of VulHawk are higher than other baselines.
  - In the cross-architecture task, VulHawk achieves the highest AUC.

	Balanced Set							Unbalanced Set						
	XC	XO	XA	XC+XO	XO+XA	XC+XA	XC+XO+XA	XC	XO	XA	XC+XO	XO+XA	XC+XA	XC+XO+XA
Asm2Vec*	0.796	0.854	-	0.861	-	-	-	0.803	0.830	-	0.864	-	-	-
Asteria	0.904	0.924	0.951	0.879	0.950	0.933	0.877	0.905	0.933	0.956	0.870	0.950	0.935	0.892
PalmTree*	0.965	0.973	-	0.952	-	-	-	0.965	0.969	-	0.948	-	-	-
GMN	0.769	0.780	0.865	0.711	0.726	0.775	0.717	0.773	0.783	0.870	0.718	0.740	0.775	0.723
SAFE	0.980	0.983	0.509	0.975	0.505	0.513	0.515	0.979	0.984	0.504	0.975	0.500	0.512	0.509
Trex	0.981	0.965	0.947	0.963	0.901	0.928	0.883	0.984	0.962	0.946	0.957	0.896	0.937	0.879
VulHawk	**0.993**	**0.990**	**0.998**	**0.990**	**0.992**	**0.994**	**0.988**	**0.996**	**0.988**	**0.998**	**0.991**	**0.993**	**0.995**	**0.987**
VulHawk-ES	0.971	0.979	0.989	0.962	0.963	0.979	0.966	0.974	0.979	0.987	0.963	0.966	0.980	0.961
VulHawk-S	0.978	0.983	0.992	0.971	0.972	0.983	0.973	0.980	0.985	0.990	0.971	0.974	0.982	0.968

* PalmTree and Asm2Vec do not support cross-architecture tasks.

4.Evaluations

- Answer to RQ1
  - VulHawk determines the similarity of two binary functions with high performance, and ranks the first in 7 tasks of one-to-one comparison.

4.Evaluations

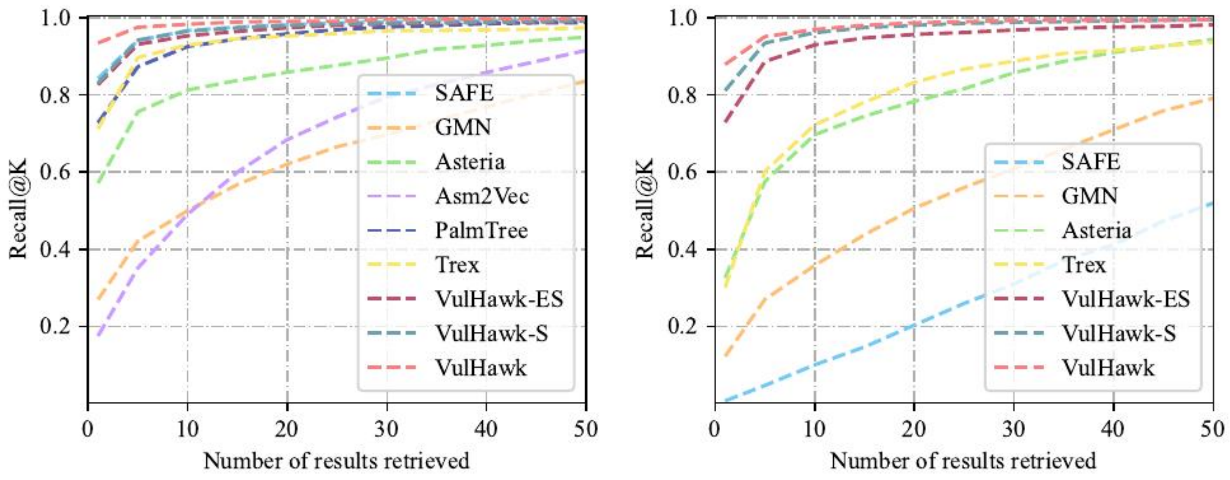
- Research questions
  - RQ1: Given two binary functions, can VulHawk determine whether they are similar?
  - RQ2: Can VulHawk be used for searching one function in a large function repository?
  - RQ3: Can VulHawk identify how many functions are similar from two binaries?
  - RQ4: Can VulHawk detect 1-day vulnerabilities in the real world?

4.Evaluations

- Results of one-to-many search

Given a function, the models retrieve top-K candidate functions from the repository (1:100).

- VulHawk outperforms the other baselines and achieves the best recall@1 of 0.935 in the XO task and 0.879 in the XC+XO+XA task.



(a) XO (b) XC+XO+XA

4.Evaluations

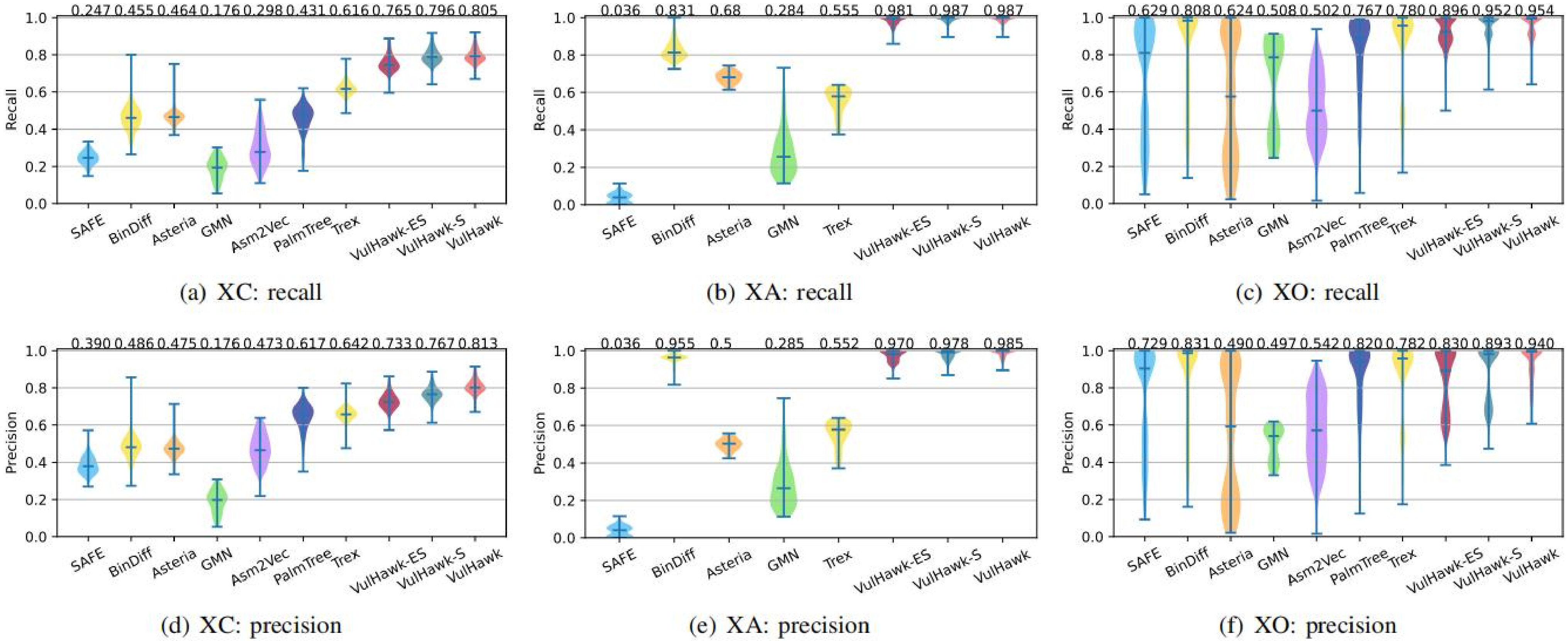
- Answer to RQ2
  - VulHawk can retrieve the best candidates accurately in a large function repository.

4.Evaluations

- Research questions
  - RQ1: Given two binary functions, can VulHawk determine whether they are similar?
  - RQ2: Can VulHawk be used for searching one function in a large function repository?
  - RQ3: Can VulHawk identify how many functions are similar from two binaries?
  - RQ4: Can VulHawk detect 1-day vulnerabilities in the real world?

4.Evaluations

- Results of many-to-many matching
  - Given two binaries, the models give how many functions are similar.
    - VulHawk outperforms the other baselines, and its probability distributions of recall and precision are more concentrated.



4.Evaluations

- Answer to RQ3
  - VulHawk can be used to match similar functions between two binaries, and it outperforms the state-of-the-art methods in many-to-many matching.

4.Evaluations

- Research questions
  - RQ1: Given two binary functions, can VulHawk determine whether they are similar?
  - RQ2: Can VulHawk be used for searching one function in a large function repository?
  - RQ3: Can VulHawk identify how many functions are similar from two binaries?
  - RQ4: Can VulHawk detect 1-day vulnerabilities in the real world?

4.Evaluations

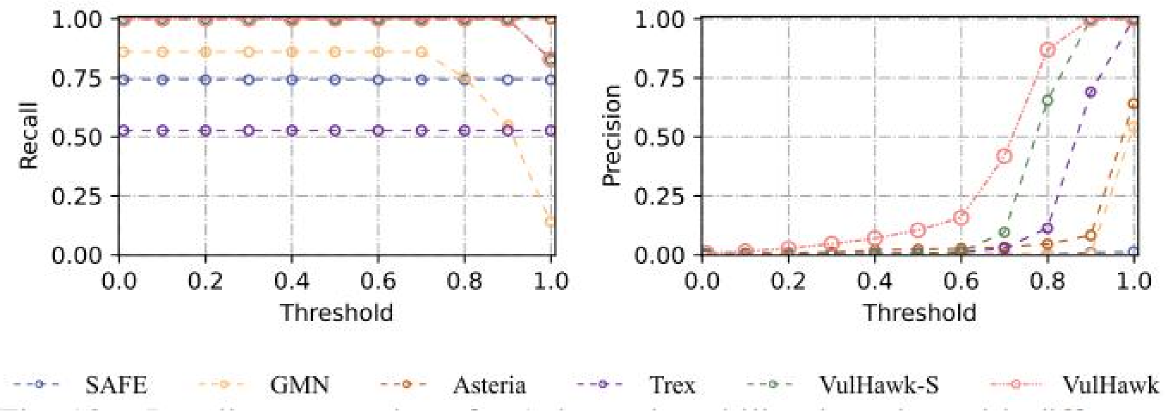
- Results of 1-day Vulnerability Detection from Firmware
  - The repository contains vulnerable functions and their patched functions of 12 relevant CVEs.
  - The ground truth includes 93 related vulnerable functions and 119 related patched functions.

Results of Vulnerability Detection

#	CVE	Confirmed #	VulHawk	Trex	SAFE	GMN	Asteria
1	2015-0286	3	3;0;0*	0;0;3	0;0;3	3;215;0	3;0;0
2	2015-1789	3	3;0;0	0;0;3	0;0;3	2;766;1	3;0;0
3	2016-0797	8	8;0;0	0;0;8	0;0;8	8;2073;0	8;0;0
4	2016-0798	4	4;0;0	0;0;4	0;0;4	4;287;0	4;0;0
5	2016-2176	4	4;0;0	0;0;4	3;0;1	4;335;0	4;0;0
6	2016-2182	14	14;0;0	9;0;5	12;0;2	3;7814;11	14;0;0
7	2016-6303	17	17;0;0	13;0;4	17;4459;0	17;1802;0	17;0;0
8	2016-8618	10	10;0;0	9;0;1	9;0;1	9;6520;1	10;0;0
9	2016-8622	10	10;0;0	9;0;1	10;753;0	10;9084;0	10;4;0
10	2018-1000301	9	9;0;0	4;0;5	9;0;0	9;4801;0	9;27;0
11	2021-22924	10	10;0;0	4;0;6	9;0;1	10;2264;0	10;2;0
12	2021-23840	1	1;0;0	1;0;0	0;0;1	1;381;0	1;19;0
	Total	93	**93;0;0**	49;0;44	69;5212;24	80;36342;13	93;52;0

* "3;0;0" represents VulHawk detects three true positives, zero false positives, and zero false negatives.

False Positive Analysis

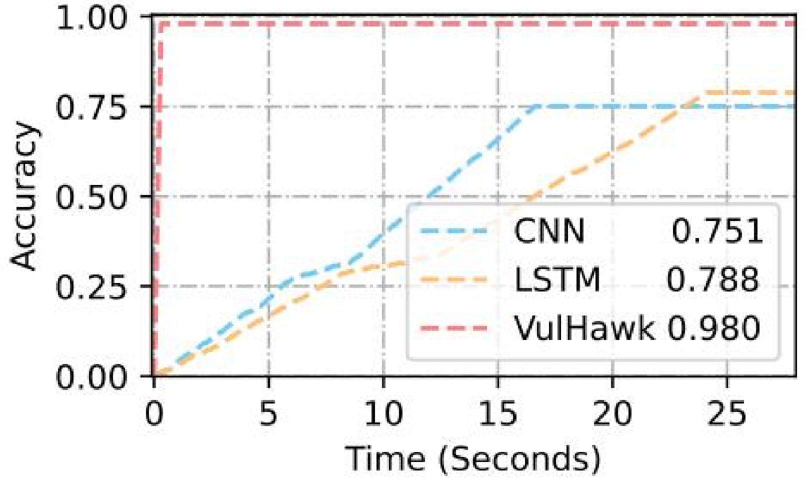


4.Evaluations

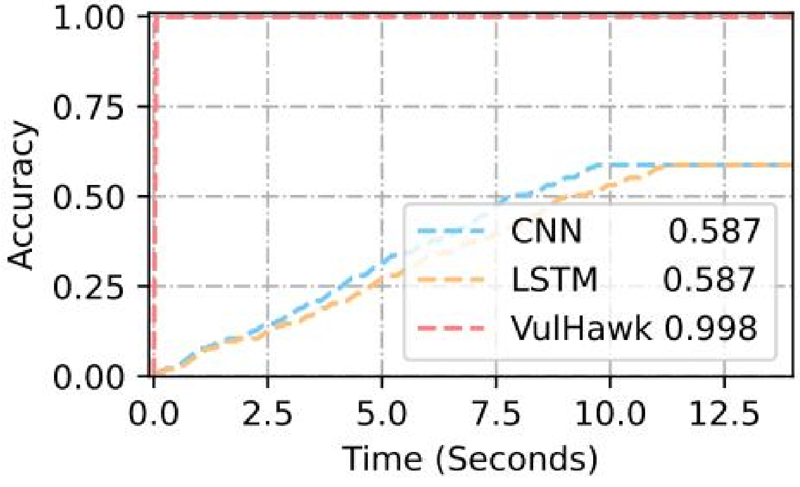
- Answer to RQ4
  - VulHawk can distinguish vulnerable functions and their patched functions and detect 1-day vulnerabilities with high performance over the baselines in the real world.

4.Evaluations

- Evaluation of file environment identification
  - Faster
  - More accurate
  - More stable



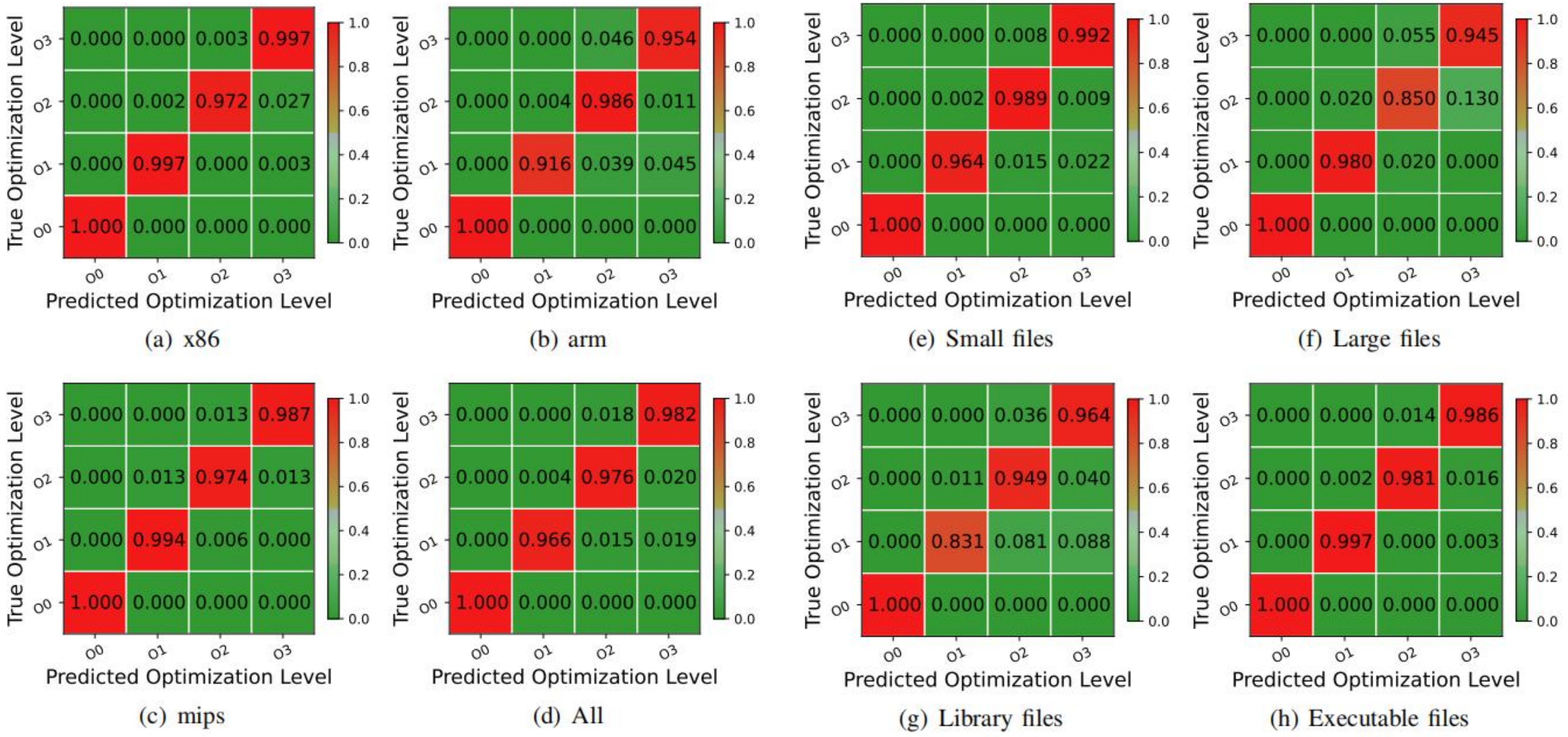
(a) optimization levels



(b) compilers

4.Evaluations

- Evaluation of file environment identification
  - Architectures
  - File sizes
  - File types



(a) x86 (b) arm (e) Small files (f) Large files

(c) mips (d) All (g) Library files (h) Executable files

5.Conclution

- We propose VulHawk: cross-architecture binary code search for vulnerability detection
  - Propose an IFRM to resolve Challenge 1.
  - Use a **divide-and-conquer** strategy and **Entropy-based adapters** to resolve Challenge 2.
  - Propose a progressive search strategy to boost the performance and reduce false positives.
  - We implement prototype VulHawk.
  - The evaluation shows the performance of VulHawk.

Q & A

Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang,
Wei Xie, Xu Zhou, Danjun Liu, Kai Lu
National University of Defense Technology
Contact:zh.luo@nudt.edu.cn