Vestige: Identifying Binary Code Provenance for Vulnerability Detection

Yuede Ji Lei Cui H. Howie Huang

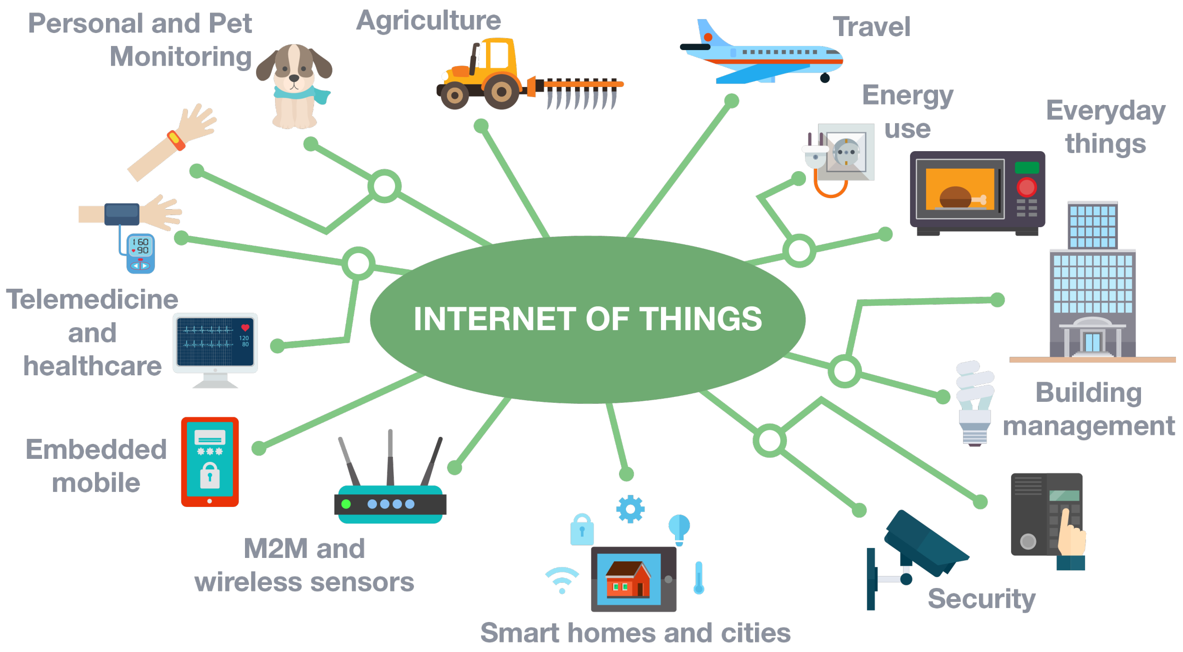
George Washington University

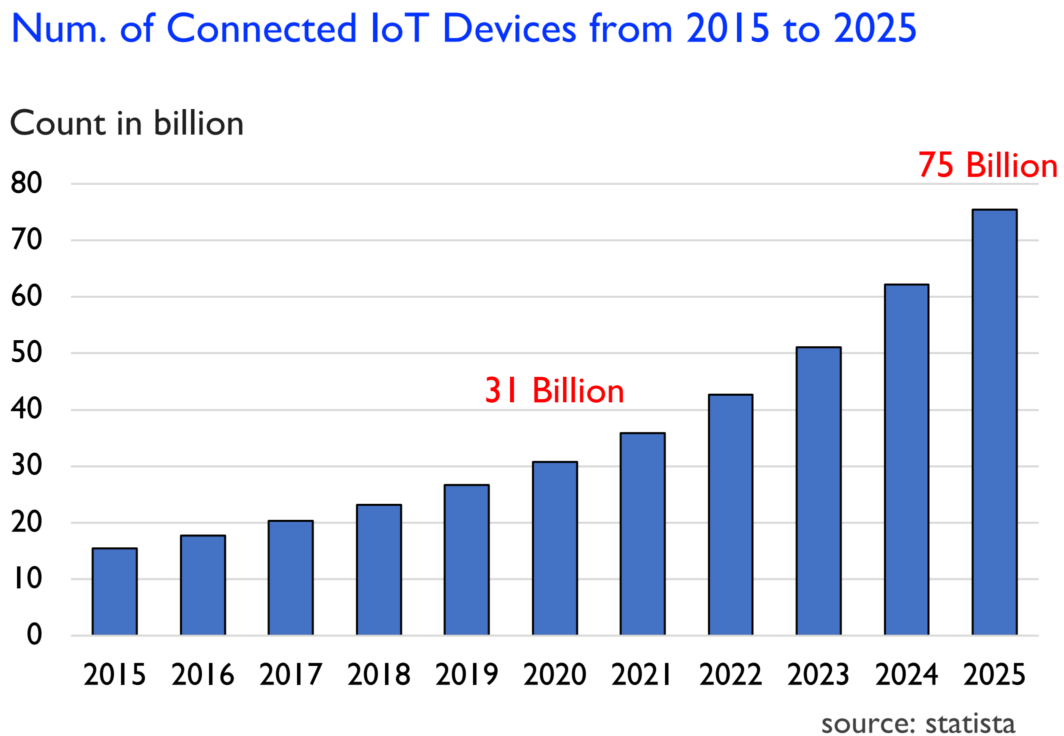
June 24th, 2021

Binary Code is Prevalent

- Software vendors usually do not share the source code.

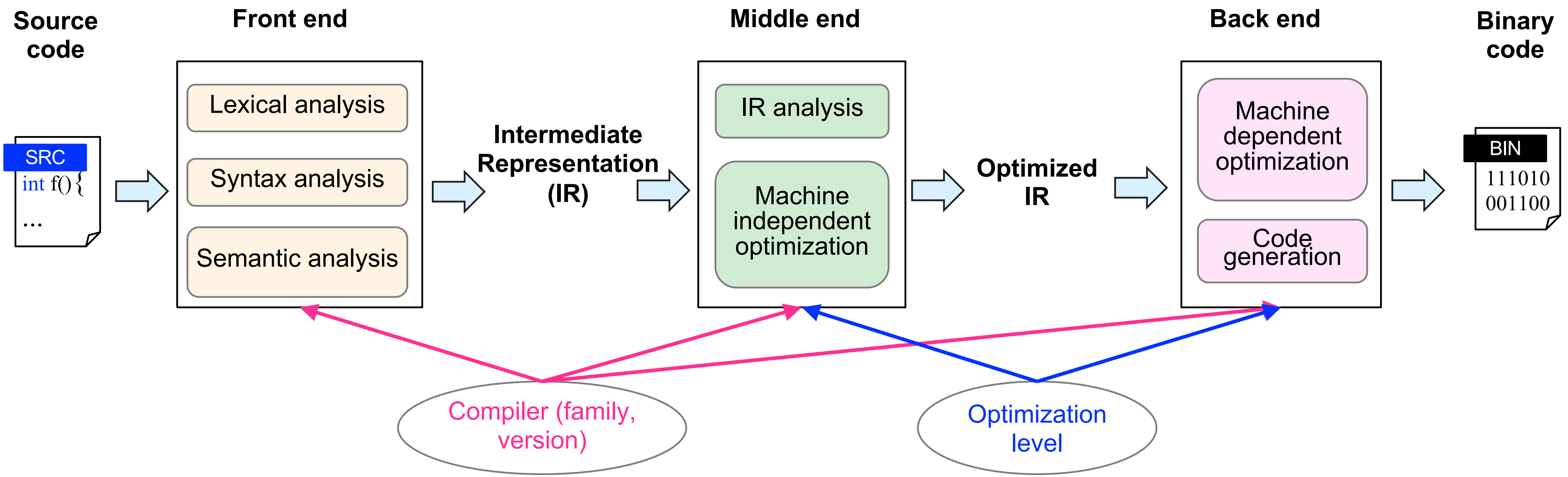
- A significant number of binaries are running in the wild, e.g., firmware of IoT devices.





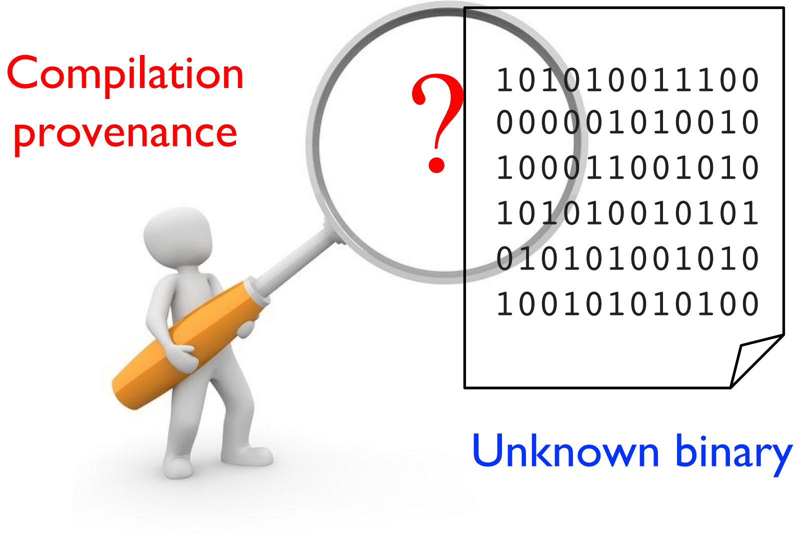
Binary Code Compilation Provenance

- Compilation *provenance*: the *compiler* and *compilation configurations.*
- Used in malware analysis, code vulnerability detection, and code authorship identification.



Research Problem and Existing Solutions

- Research Problem
  - Identify the compilation provenance of an unknown binary.
  - The provenance is regarded as a 3-tuple, <*compiler family*, *version*, *optimization level*>.



- Existing Solutions *[Rosenblum et al. ISSTA'11], [Massarelli et al. BAR'19], [Rahimian et al. DFRWS'15]*
  - Convert the provenance identification problem to a classification problem.
  - Extract features (patterns) from binary code.
    - Instructions features, control flow graphs.
  - Build a machine learning-based prediction model.

Motivation

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main() {
    int a, b;
    scanf(“%d%d”, &a, &b);
    int c = sum(a, b);
    printf(“c = %d\n”, c);
    return 0;
}
```

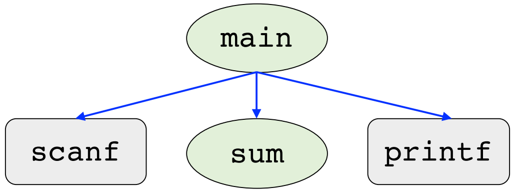
(a) An example source code with inter overflow

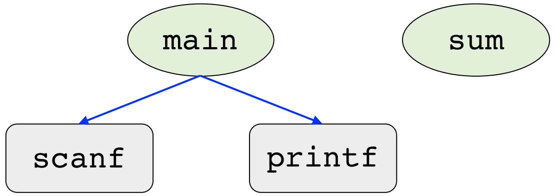
```
<sum>:
push    ebp
mov     ebp, esp
mov     eax, dword ptr [ebp+0Ch]
mov     edx, dword ptr [ebp+8]
add     eax, edx
pop     ebp
ret

<main>:
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
lea     eax, [esp+18h]
mov     [esp+8], eax
lea     eax, [esp+14h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDD
call    ___isoc99_scanf
mov     edx, [esp+18h]
mov     eax, [esp+14h]
mov     [esp+4], edx
mov     [esp], eax
call    sum
mov     [esp+1Ch], eax
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset format
call    _printf
mov     eax, 0
leave
ret
```

(b) Assembly code of GCC-4.8.4-O0

```
<sum>:
mov     eax, dword ptr [esp+4]
add     eax, dword ptr [esp+8]
ret

<main>:
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
lea     eax, [esp+1Ch]
mov     [esp+8], eax
lea     eax, [esp+18h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDD
call    ___isoc99_scanf
mov     eax, [esp+1Ch]
add     eax, [esp+18h]
mov     dword ptr [esp+4], offset aCD
mov     dword ptr [esp], 1
mov     [esp+8], eax
call    ___printf_chk
xor     eax, eax
leave
ret
```

(c) Assembly code of GCC-4.8.4-O2

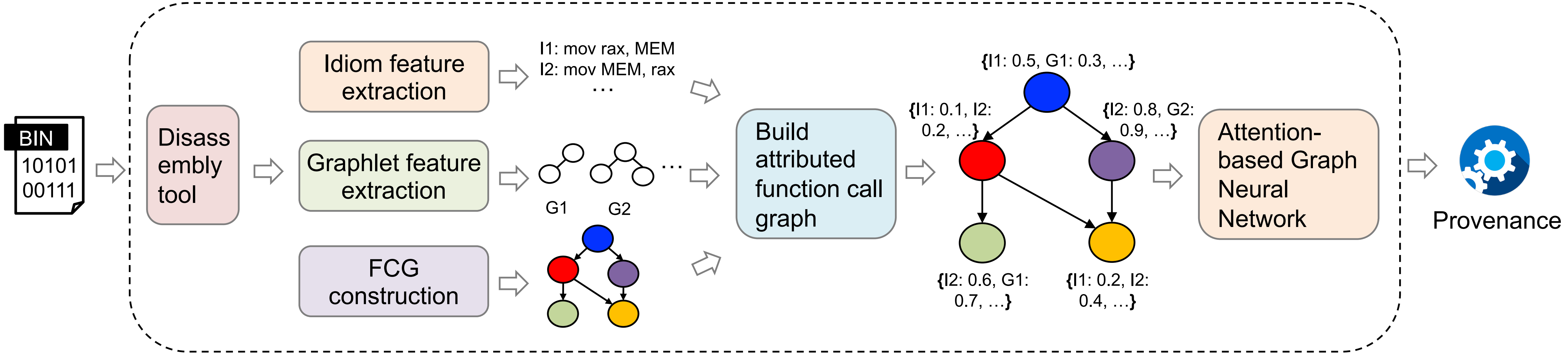




- *GCC-4.8.4-O0* v.s. *GCC-4.8.4-O2*
  - *Control flow graphs* are same
  - *Instructions* have minor differences
  - *Function call graphs* are obviously different

Overview of Vestige

- Vestige: a graph neural network-based binary code provenance identification method.
  - Represent a binary code as *attributed function call graph (ACFG).*
  - Use *graph attention network* to learn a provenance prediction model.



Attributed Function Call Graph

```
...
do {
      tbio = BIO_pop(f);
      BIO_free(f);
      f = tbio;
} while (f != upto)
...
```

Source code fragment of CVE-2015-1792

```
loc_816C6F2:
mov     [esp+1Ch+var_1C],
        ebx
call    BIO_pop
mov     [esp+1Ch+var_1C],
        ebx
```

```
loc_816C6F0:
...
```

Provenance 1: *GCC-4.8.4-O2*

```
loc_824A343:
mov     eax, [esp+1Ch+arg_0]
mov     [esp+1Ch+var_1C],
         eax
call    BIO_pop
mov     [esp+1Ch+var_1C],
         eax
...
```

Provenance 2: *LLVM-5.0-O0*

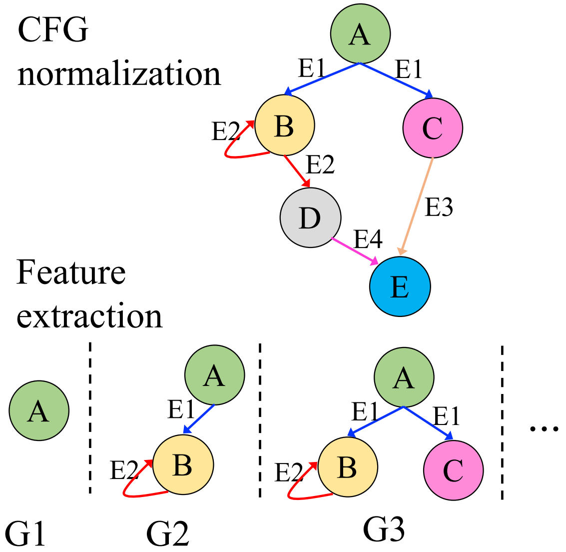
Instruction normalization

```
mov     rax, MEM
mov     MEM, rax
call    rip
mov     MEM, rax
...
```

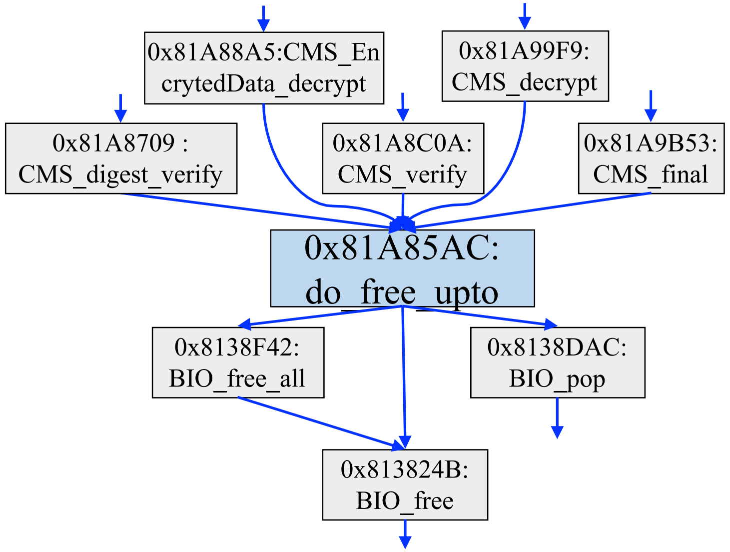
Feature extraction

```
I1: mov rax, MEM
I2: mov rax, MEM | mov MEM, rax
I3: mov rax, MEM | * | mov MEM, rax
...
```

Idiom features



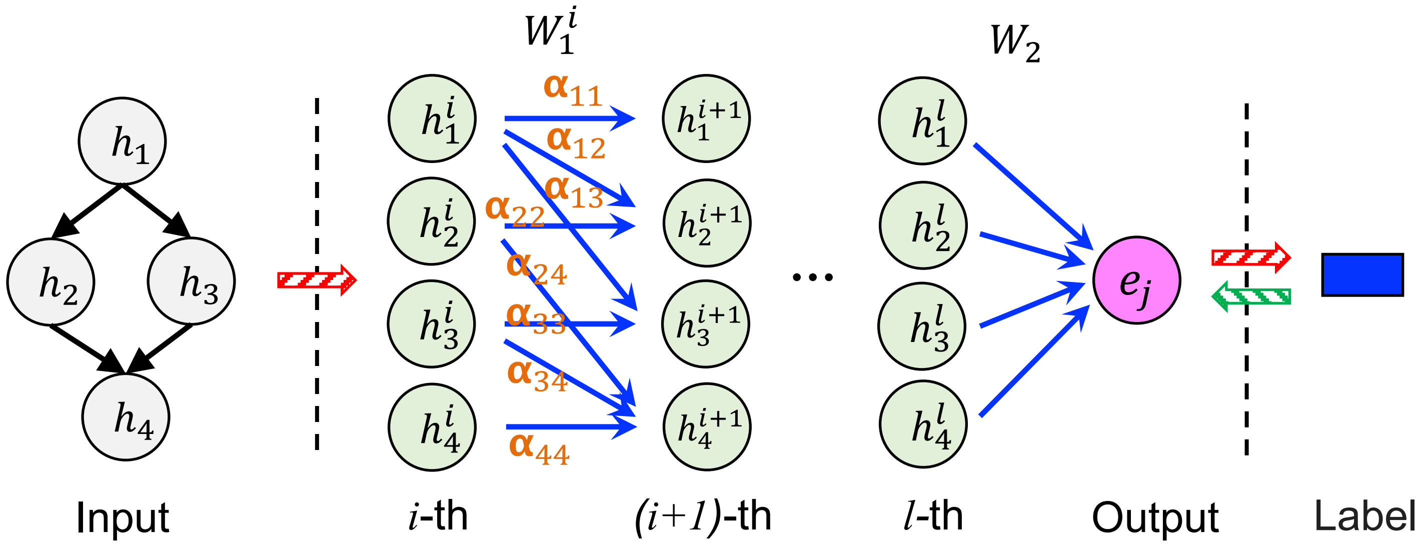
Graphlet features



Function call graph

Attributed Graph Embedding

- Graph Embedding
  - Graph attention network (GAT) *[Veličković et al. ICLR'18]*
  - Objective function: $h_v^{i+1} = \sigma\left(\alpha_{vv} h_v^i + \sum_{u \in N(v)} \alpha_{vu} W_1^i h_u^i\right)$



Experiment

- Dataset

Dataset	Software	Compiler	Optimization	# Binaries
I (Baseline)	Bash-4.3, Diffutiils-3.3, Grep-2.16, Tar-1.27.1, Wget-1.15	GCC-{4.6.4, 4.8.4, 5.4.1}, LLVM-{3.3, 3.5, 5.0}	O0 – O3	336
II (Code similarity dataset)	SNNS-4.2, PostgreSQL- 7.2, Binutils-{2.25, 2.30}, Coreutils-{8.21, 8.29}			6,168
III (Vulnerability dataset)	OpenSSL-{0.9.7f, 1.0.1f, 1.0.1n}			24

- Evaluation Metrics
  - Accuracy
  - Top-k hit rate

Accuracy of Provenance Identification

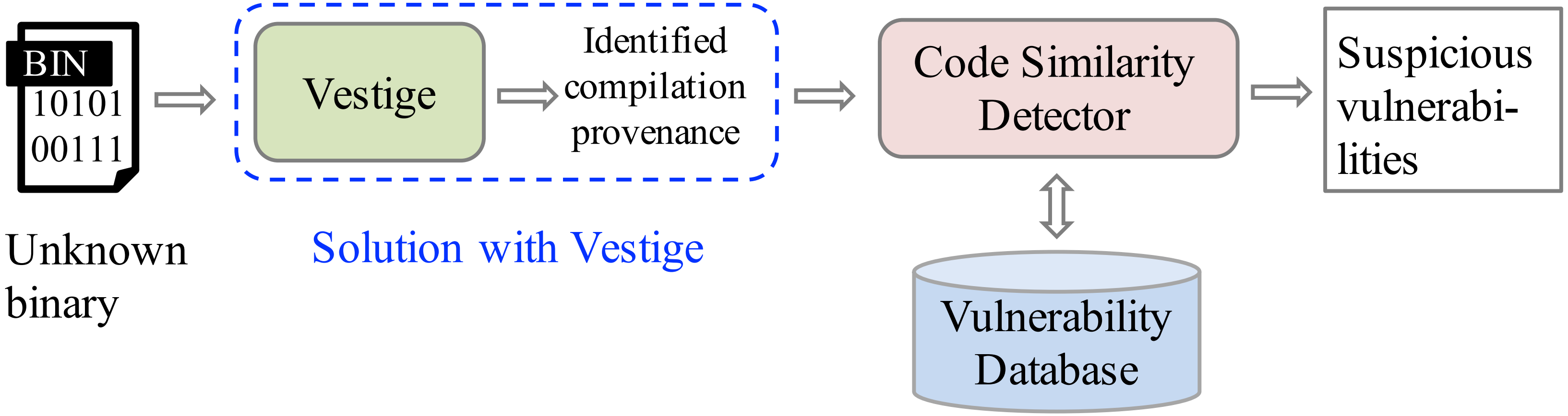
- Setting
  - 6,504 binaries from dataset I and II.
  - 10-fold cross validation.

- Compared Works
  - Origin *[Rosenblum et al. ISSTA'11]*
  - Vestige-S2V: a different graph neural network, structure2vec, on AFCG.

	Origin	**Vestige-S2V**	**Vestige-GAT**
Optimization level	92.2%	98.7%	**99%**
Compiler version	96.8%	95.5%	**97.9%**
Compiler family	**99.5%**	**99.5%**	**99.5%**
Overall	90.2%	93.3%	**96.1%**

Case Study #1: Code Similarity Detection

- Two-step Solution with Vestige
  - First step: identify compilation provenance of the unknown binary code.
  - Second step: compare the unknown binary code with vulnerable code sharing the same compilation provenance.
- Code Similarity Detector
  - BGM [*Bipartite Graph Matching*]
  - Genius [*Qian et al. CCS'16*]
  - Gemini [*Xu et al. CCS'17*]



Case Study #1: Code Similarity Detection

- Dataset II
  - 6K+ binaries
  - 24 compilation provenances
- Training and testing use different software
- Search 1,000 functions

	Software	Compilation Provenance	# Binaries
Train	SNNS-4.2, PostgreSQL-7.2	Compiler: GCC-{4.6.4, 4.8.4, 5.4.1}, LLVM-{3.3, 3.5, 5.0} Optimization: O0 – O3	600
Test	Binutils-{2.25, 2.30}, Coreutils-{8.21, 8.29}		5,568
Total	-	24	6,168

	Top-1		**Top-5**	
	Original	Original + Vestige	Original	Original + Vestige
BGM	29%	56% (+**27%**)	45%	89% (+**44%**)
Genius	51%	64% (+**13%**)	69%	91% (+**22%**)
Gemini	66%	87% (+**21%**)	77%	94% (+**17%**)

Case Study #2: Vulnerability Detection

- OpenSSL dataset (III)
  - 20 vulnerable functions
  - Query against 24 variants
- Top-1 hit rate
- Result
  - BGM: 33% → 49% (16%)
  - Genius: 39% → 58% (19%)
  - Gemini: 50% → 76% (26%)

CVE	Query	BGM:+Vestige	Genius:+Vestige	Gemini:+Vestige
2015-0209	1.0.1f	46:67	50:71	50:88
2014-0195	1.0.1f	33:42	42:58	54:92
2016-2106	1.0.1f	58:63	58:63	63:83
2012-0027	1.0.1f	42:58	58:67	63:88
2014-3513	1.0.1f	46:71	50:83	67:92
2015-1791	1.0.1f	50:67	50:83	71:96
2015-3196	1.0.1f	42:67	38:75	58:92
2014-3567	1.0.1f	22:56	33:67	50:79
2016-0797	1.0.1n	21:41	25:41	38:83
2016-2180	1.0.1n	25:42	29:83	46:95
2016-2105	0.9.7f	58:67	47:58	58:83
2016-2176	1.0.1n	38:42	38:46	50:67
2016-2109	0.9.7f	10:29	30:38	40:54
2015-3195	0.9.7f	25:42	50:63	58:83
2016-2182	0.9.7f	25:42	33:50	46:58
2016-2178	0.9.7f	13:25	19:42	25:63
2015-0292	0.9.7f	37:42	46:54	50:67
2016-2105	0.9.7f	58:63	58:63	67:71
2016-2842	1.0.1n	5:25	10:33	19:50
2016-0705	1.0.1n	13:21	17:21	19:42
Average	-	33:49	39:58	50:76

Summary

- Key Takeaway:
  - *Identifying compilation provenance* can effectively improve code similarity detection.
  - The *attributed function call graph* (AFCG) is an efficient representation for binary code analysis.

- Impact:
  - Improve *top-1* hit rate of recent code vulnerability detection methods by up to 26%.

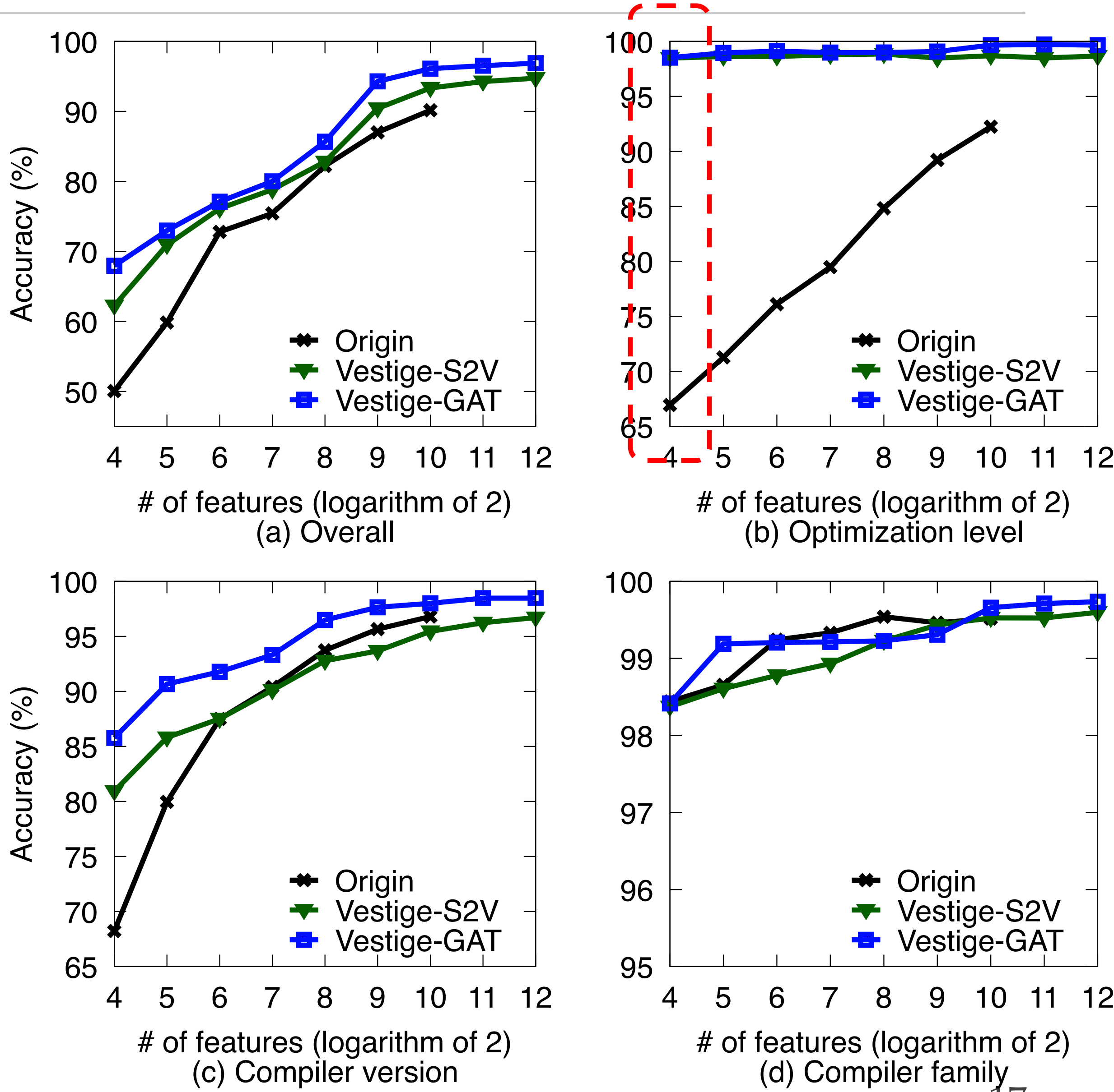
Thank You

Contact us: yuedeji@gwu.edu, leicui@gwu.edu, howie@gwu.edu

Backup Slides

Accuracy of Provenance Identification

- Accuracies against number of features
  - Function call graph is efficient for optmization level



Sensitivity Study