BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network

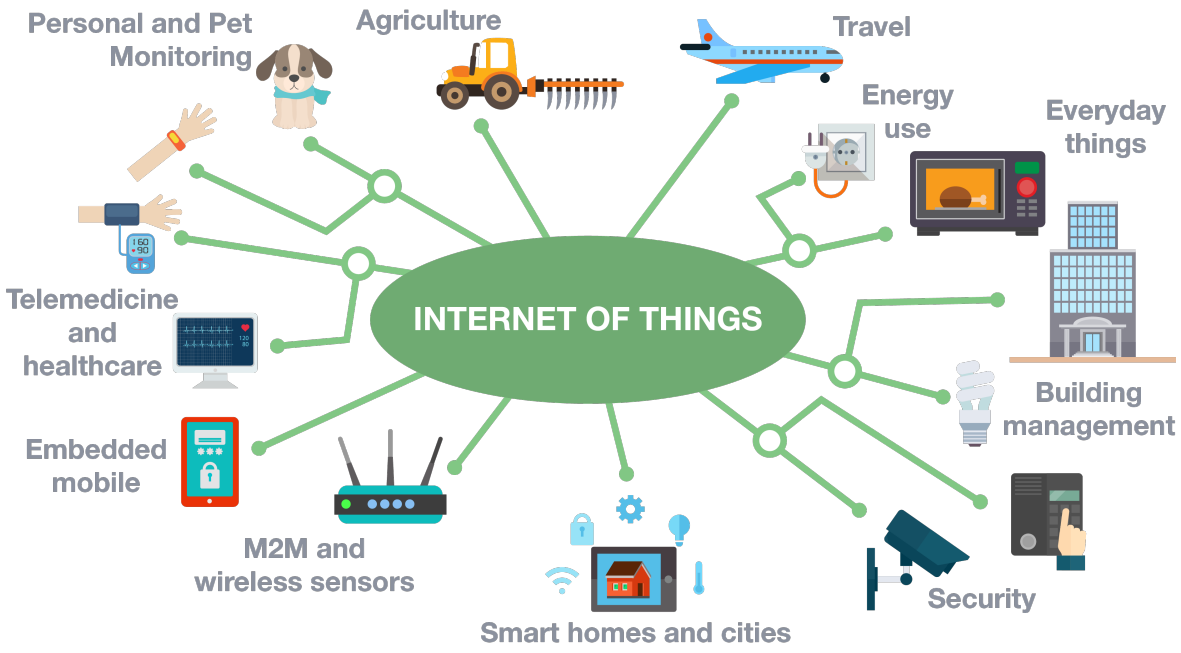
Yuede Ji Lei Cui H. Howie Huang

George Washington University

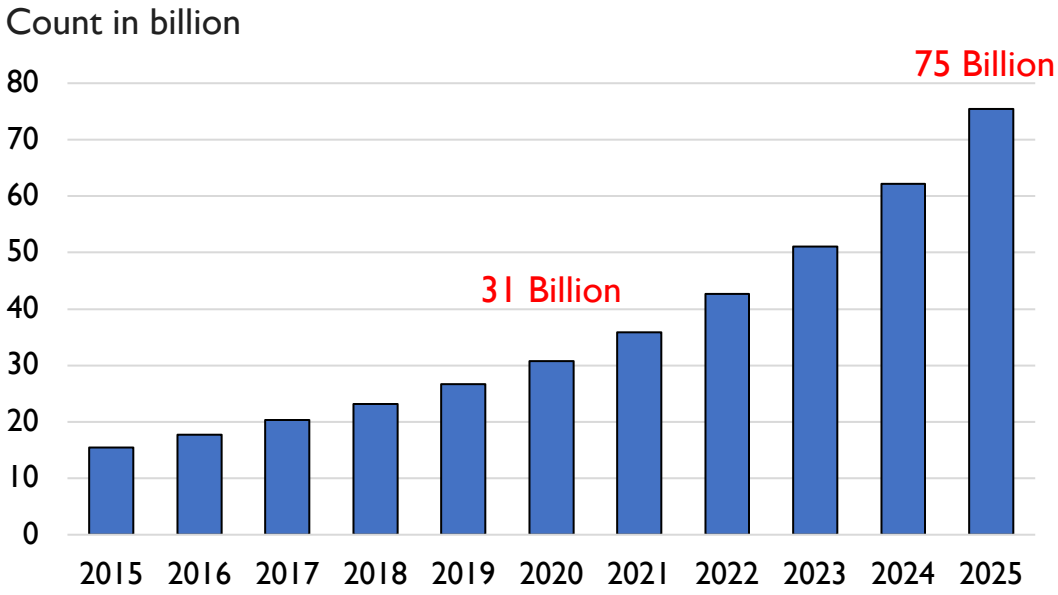
AsiaCCS 2021

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.



Num. of Connected IoT Devices from 2015 to 2025



source: statista

What is Vulnerability?

- A vulnerability is a *weakness* (usually a *bug*) that can be exploited by the attacker to perform unauthorized activities.

- Real-world impact
  - *WannaCry* ransomware infects 200,000 computers across 150 countries.
  - Exploits CVE-2017-0144 vulnerability in Windows' Sever Message Block protocol.

```
1  int square_sum(int a)
2  {
3      int result = 0;
4      int i;
5      for(i=1; i<a; ++i)
6      {
7          result += i * i;
8      }
9      return result;
10 }
```

Integer overflow

Research Problem

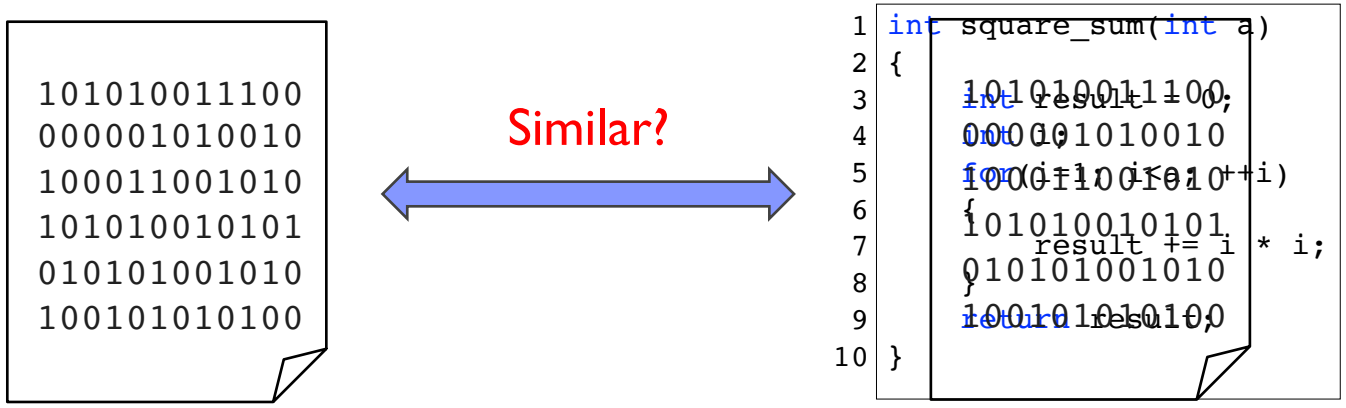
- Motivation
  - Open-source libraries are widely used in binaries, e.g., OpenSSL.
  - Up to *80%* of the binaries in IoT firmware still use third-party libraries with *already discovered vulnerabilities.*

- Research Problem
  - How to scalably check whether the already discovered vulnerabilities (*source code*) still exist in the *unknow binary code*.

[NDSS'13] Cui, Ang, Michael Costello, and Salvatore Stolfo. "When firmware modifications attack: A case study of embedded exploitation." (2013). In NDSS.

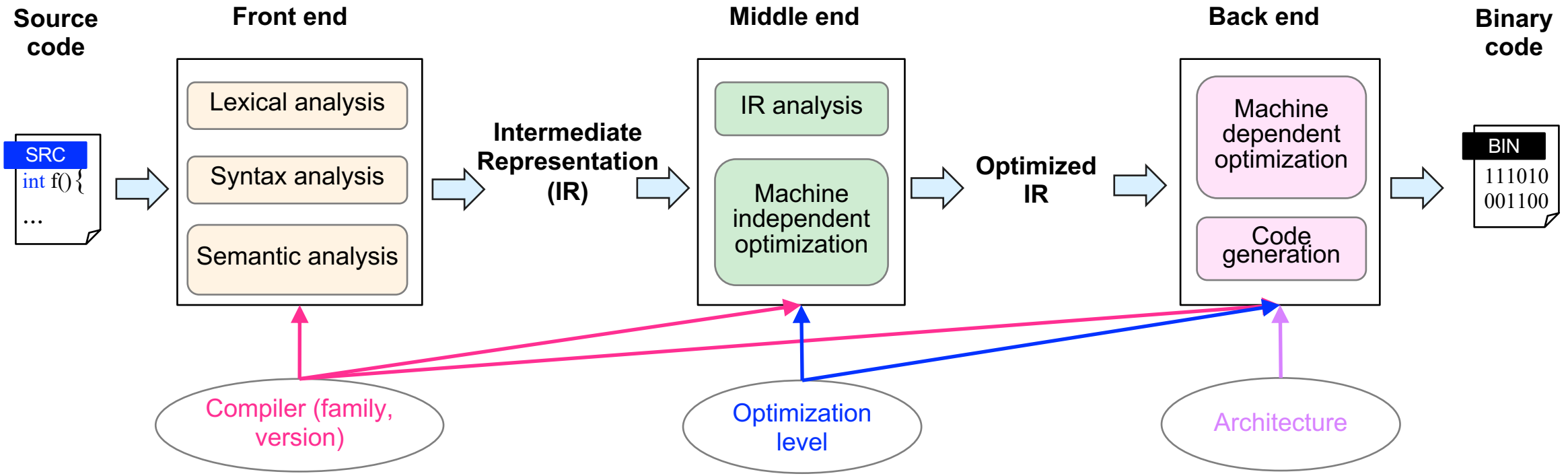
Similarity-Based Solution

- Source-Binary Code Similarity for Vulnerability Detection
  - If an *unknown code* is similar to a *discovered vulnerability template*, then the unknown code is potentially to have the same vulnerability.
- Challenge:
  - The source and binary code are *not in a canonicalized form*.
- Existing Solution:
  - Transform the problem to *binary code similarity detection* by compiling the source code to a binary.



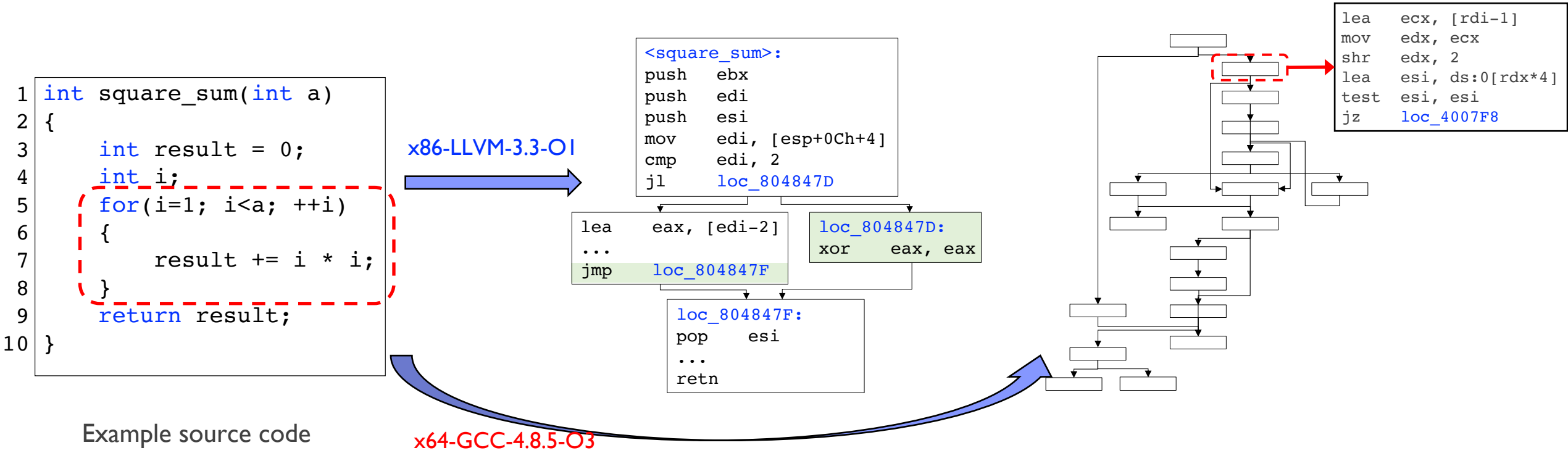
Challenge #1: Low Accuracy

- *Low accuracy* due to many combinations of compiling options
  - Compiler: GCC (204 versions), LLVM (55 versions), etc.
  - Optimization level: *O-{0, 1, 2, 3, s, fast, g}*
  - Architecture: x86, ARM, MIPS, etc.



Challenge #1: Low Accuracy

- *Low accuracy* due to many combinations of compiling options
  - Compiler: GCC (204 versions), LLVM (55 versions), etc.
  - Optimization level: *O-{0, 1, 2, 3, s, fast, g}*
  - Architecture: x86, ARM, MIPS, etc.



Example source code

Challenge #2: Low Coverage of Similar Code Types

- Three types of similar code
  - Type-1: Difference in the white space, blank line, layout, and comment.
  - Type-2: Difference in identifiers, literals, and data types.
  - Type-3: Difference in statements, including changed, added, or deleted.
- *Low coverage* as prior works mainly focus on Type-1/2 similar code
  - Type-3 contributed to over *50%* of all vulnerabilities discovered by code similarity [*Islam et al. ESEM'17*]

```
1  int square_sum(int a)
2  {
3      int result = 0;
4      int i;
5      for(i=1; i<a; ++i)
6      {
7          result += i * i;
8      }
9      return result;
10 }
```

Example source code

```
1  int square_sum(int a)
2  {
3      int result = 0;
4      int i;
5      for(i=1; i<a; ++i)
6      {
7          result += i * i;
8          // weakness
9      }
10     return result;
11 }
```

Type-1

```
1  int square_sum(int a)
2  {
3      long long int s = 0;
4      int i;
5      for(i=1; i<a; ++i)
6      {
7          s += i * i;
8      }
9      return result;
10 }
```

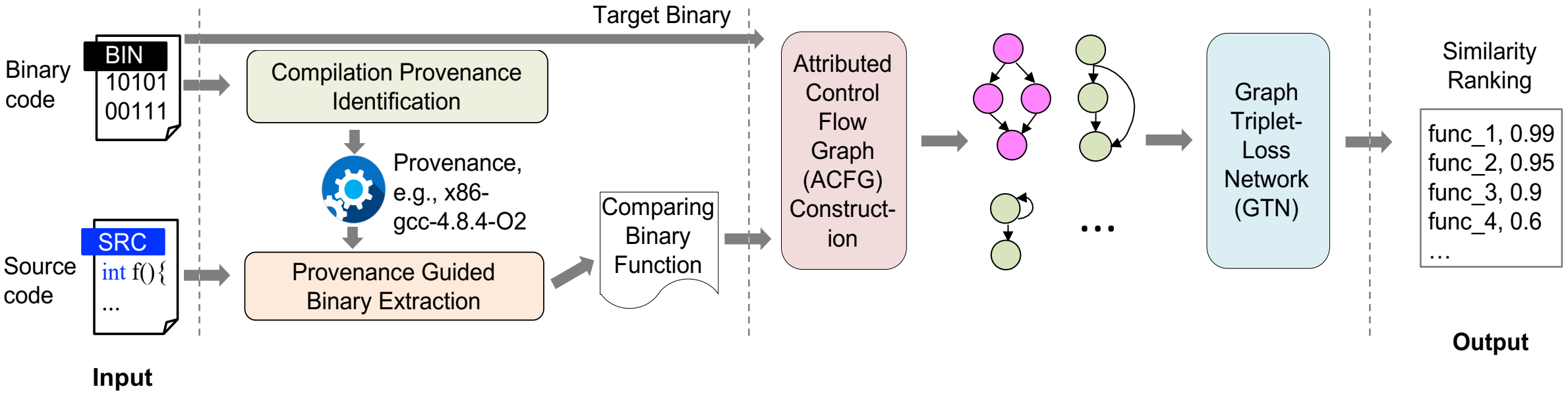
Type-2

```
1  int square_sum(int a)
2  {
3      if (a < 1)
4          return 0;
5      int result = 0;
6      int i;
7      for(i=1; i<a; ++i)
8      {
9          result += i * i;
10     }
11     return result;
12 }
```

Type-3

Overview of BugGraph

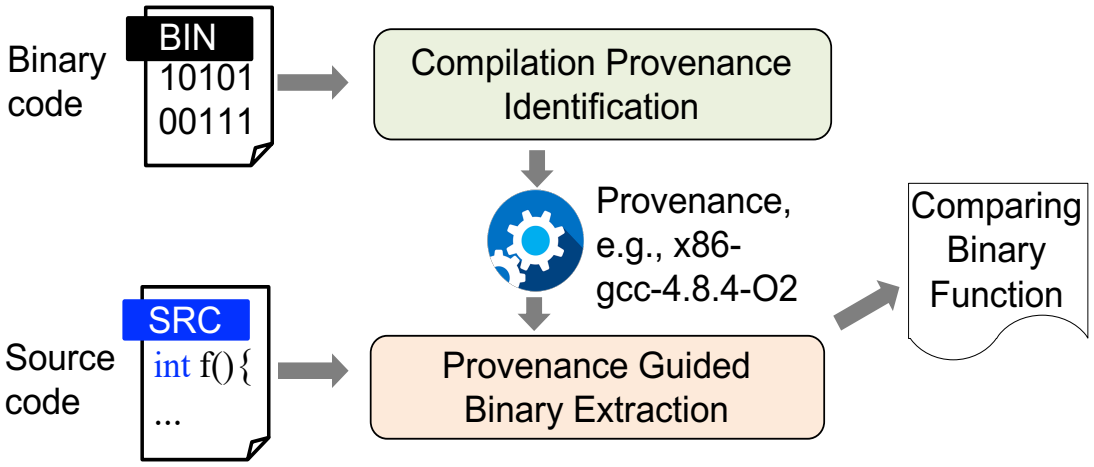
- BugGraph: a two-step source-binary code similarity detection system
  - Step 1: Source binary canonicalization
    - *First work* to *identify compilation provenance* for source-binary code similarity detection.
  - Step 2: Code similarity computation
    - *Ranking-based* graph triplet-loss network to better detect type-3 similar code.



Step 1: Source Binary Canonicalization

- Key Insight:
  - *Provenance*-guided source binary canonicalization

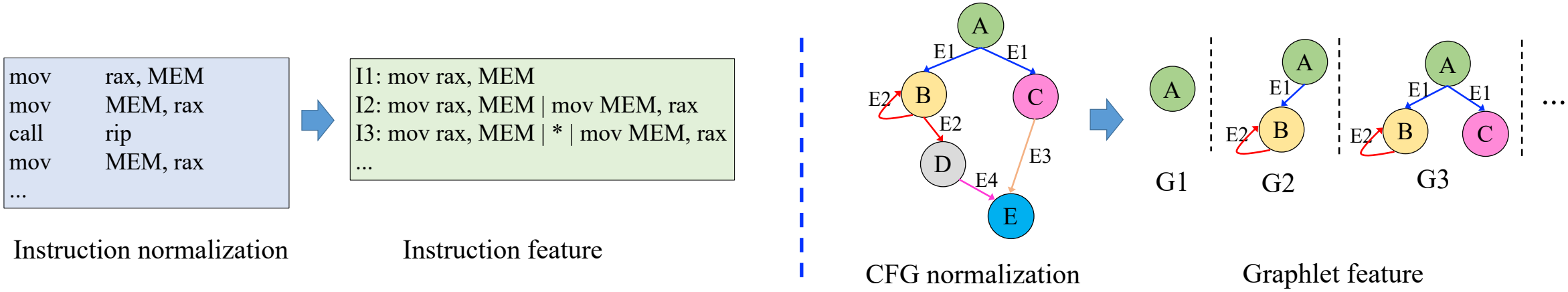
- Compilation Provenance
  - *<architecture, compiler family, compiler version, optimization level>*



Step 1: Source Binary Canonicalization

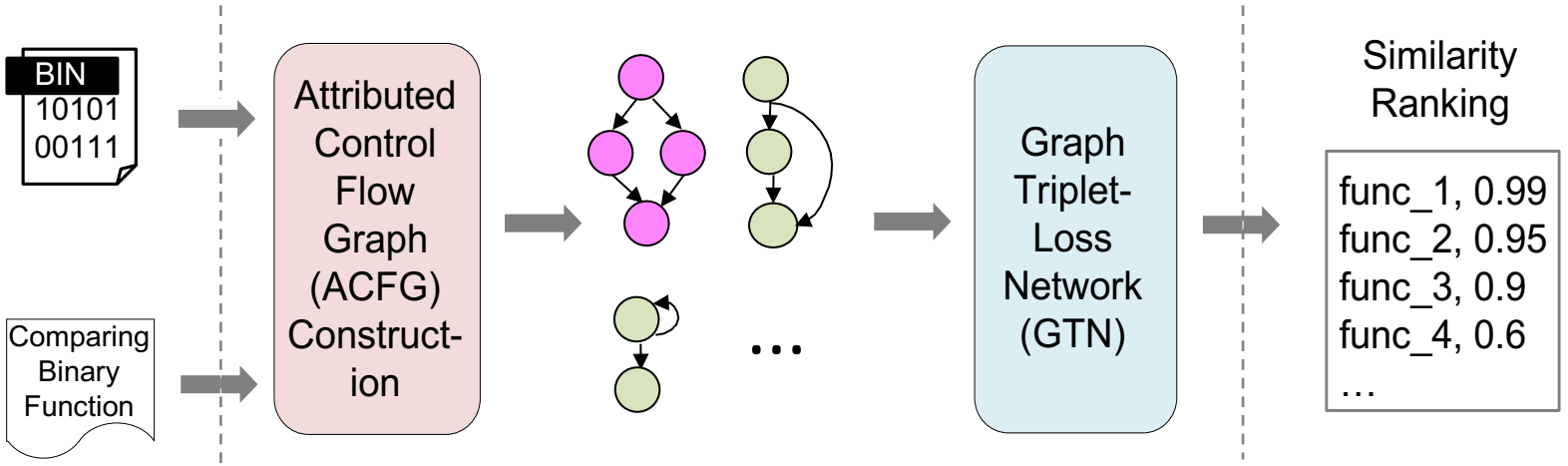
Step 1: Source Binary Canonicalization

- Architecture: *file* command
- Compiler family, version, optimization level.
  - Customized from *Origin [Rosenblum et al. ISSTA'11]*
  - *Classification problem* by taking the compilation provenance as label.
  - Extract code pattern as feature and build a machine learning model.



Step 2: Code Similarity Computation

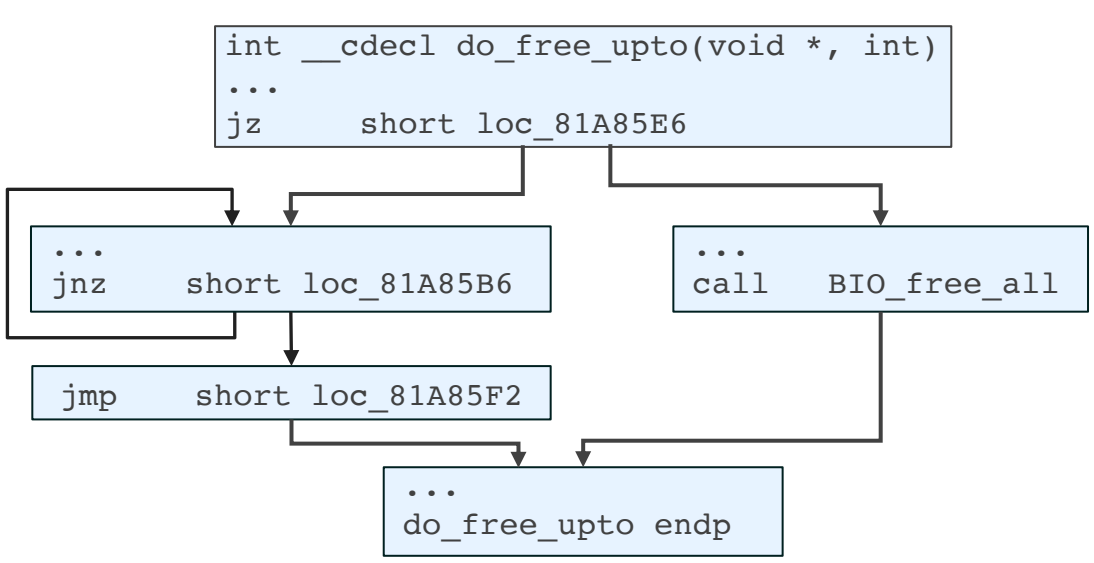
- Key Insight:
  - Represent binary code with *attributed control flow graph*.
  - Use ranking-based *graph triplet-loss network* to identify different types of similar code.



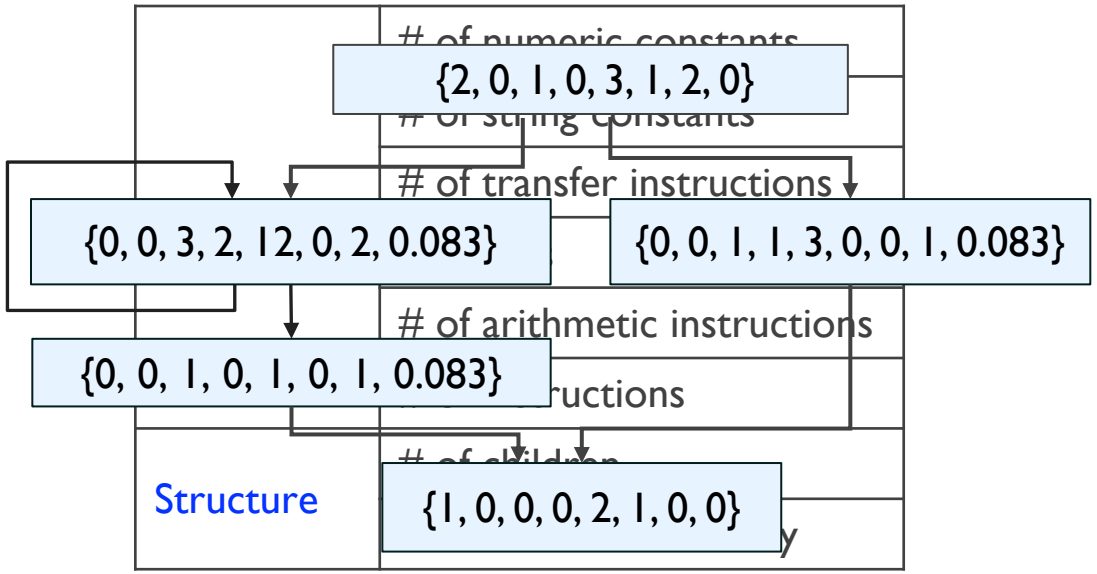
Binary Code to Attributed Graph

- Attributed control flow graph
  - Control flow graph: essential code structure feature
  - Attributes: syntax features

- The problem has become *attributed graph similarity* problem.



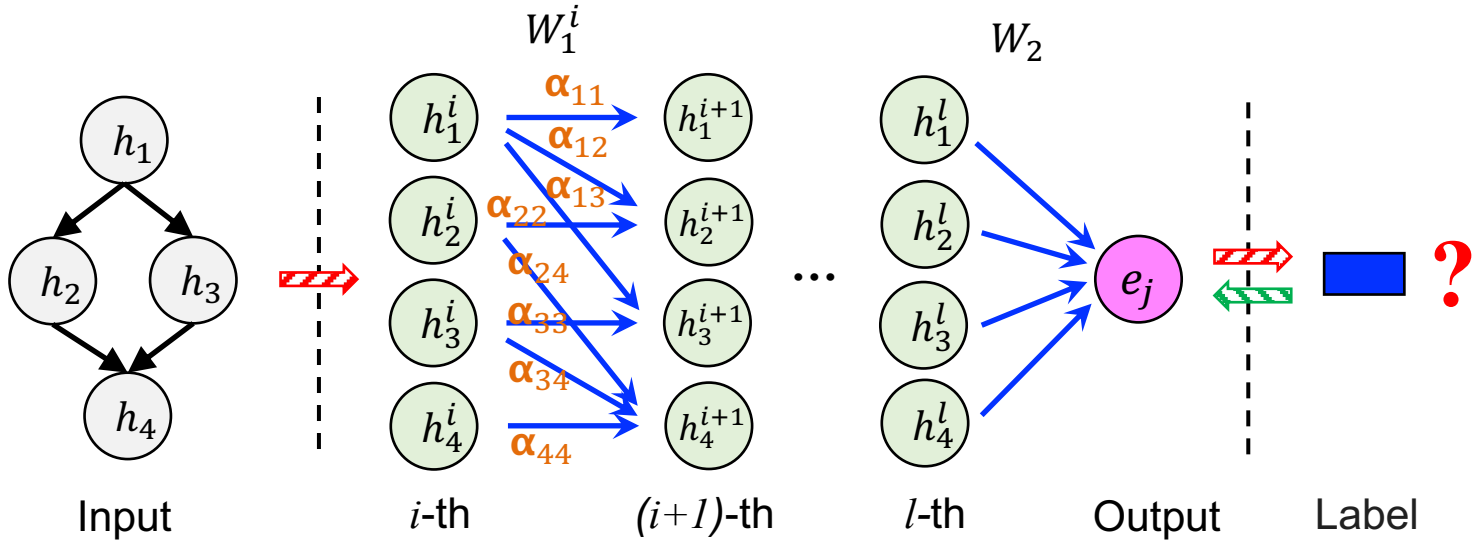
Assembly code of CVE-2015-1792 compiled with x86-GCC-4.8.4-O0



[CCS'16] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In CCS 2016.

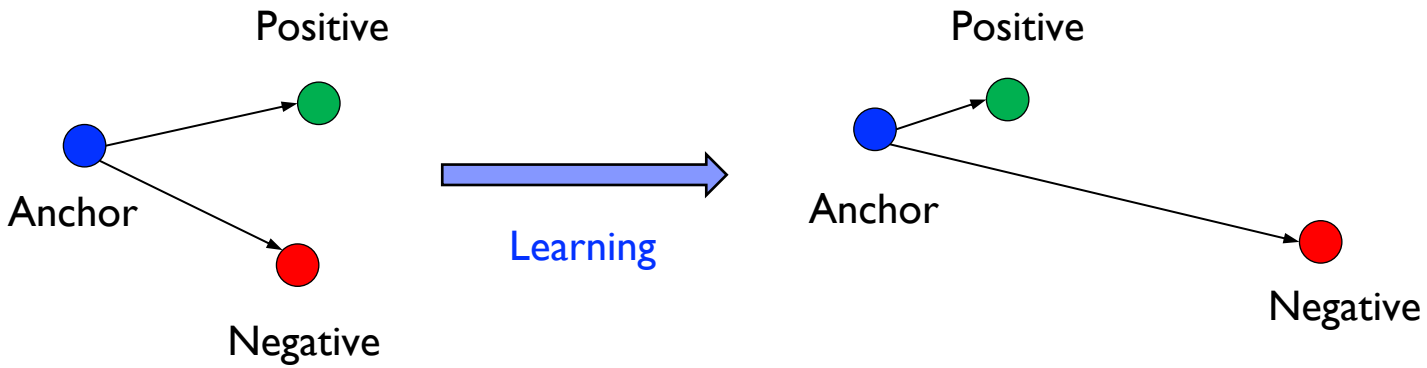
Attributed Graph Embedding

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



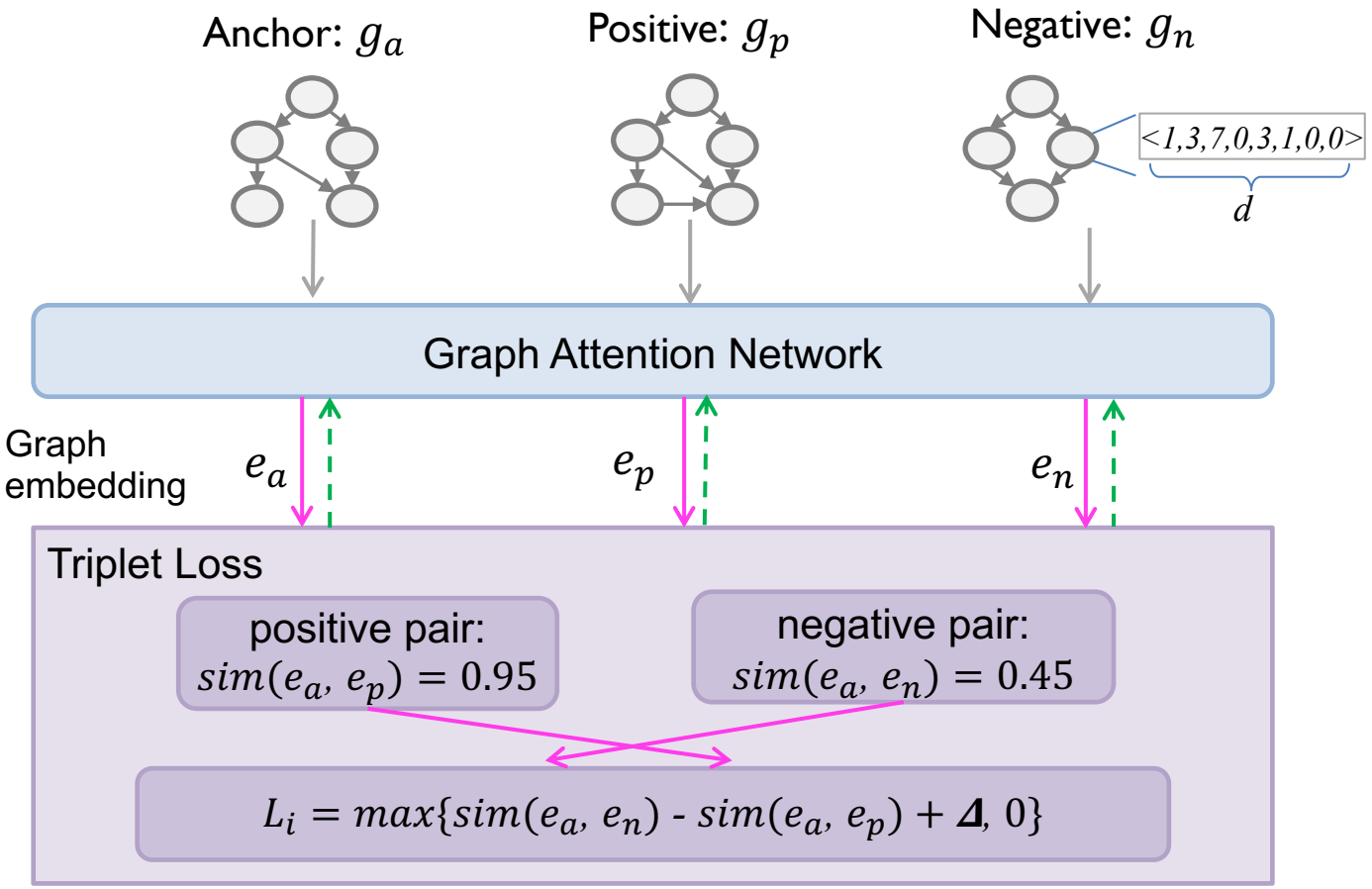
Ranking-Based Triplet Loss Network

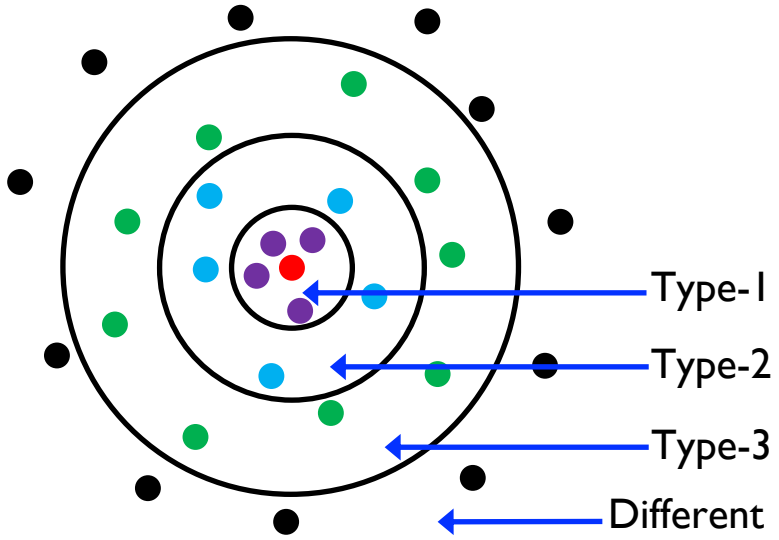
- Triplet Loss Network
  - Triplet: <*anchor, positive, negative*>
  - *Similarity label*: anchor is more similar with positive than with negative.
  - Loss function: $L = \max(d(a,p) - d(a,n) + margin, 0)$



Graph Triplet-Loss Network for Code Similarity

- Ranking similar code:
  - Type-1 > Type-2 > Type-3 > Different





Experiment

	Software	Compilation Provenance	Similarity Types	# Binaries	# Functions
Train	SNNS-4.2, PostgreSQL-7.2	Architecture: x86 Compiler: GCC-{4.6.4, 4.8.4, 5.4.1}, LLVM-{3.3, 3.5, 5.0} Optimization: O0 – O3	Type-1/2/3	600	493,841
Test	Binutils-{2.25, 2.30}, Coreutils-{8.21, 8.29}		Type-1/2/3	5,568	2,648,627
Total	-	24	Type-1/2/3	6,168	3,142,468

- Dataset
  - 6K+ binaries, 3M+ functions
  - 24 compilation provenances

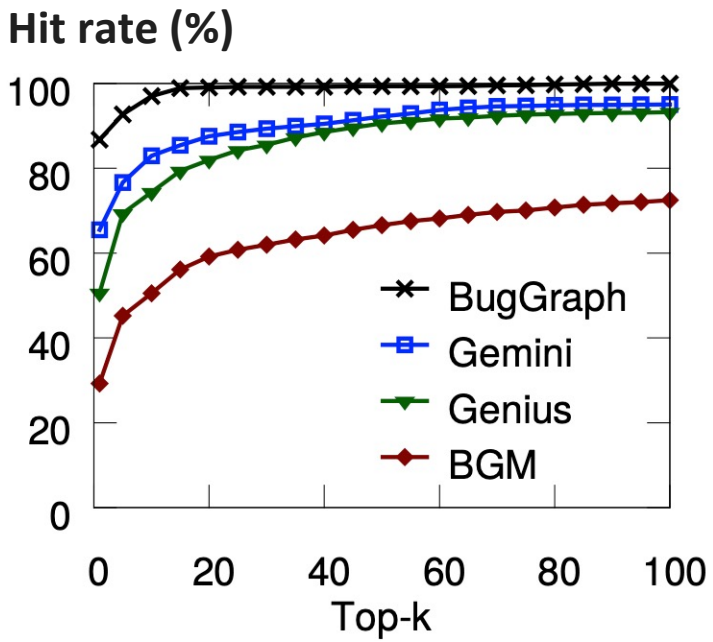
- Training and testing use different software
- Search 3,000 Type-1/2/3 similar code (1,000 for each)
- Report the *top-k hit rate*, the smaller k the more important

Source-Binary Code Similarity Test

- Compilation provenance accuracy:
  - 82% overall accuracy
  - 100%, 100% for architecture, compiler family
  - 96%, 84% for version, and optimization

- Type-1 similar code:
  - E.g., Top-5 hit rate
    - BugGraph, 93%
    - Gemini *[Xu et al. CCS'17]*, 77%
    - Genius *[Qian et al. CCS'16]*, 69%
    - BGM *[Bipartite Graph Matching]*, 45%

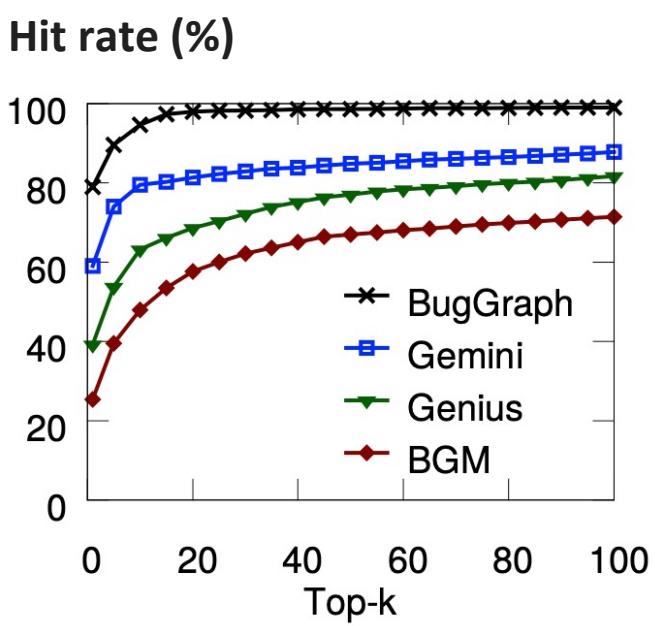
Top-k Hit Rate of Type-1 Similar



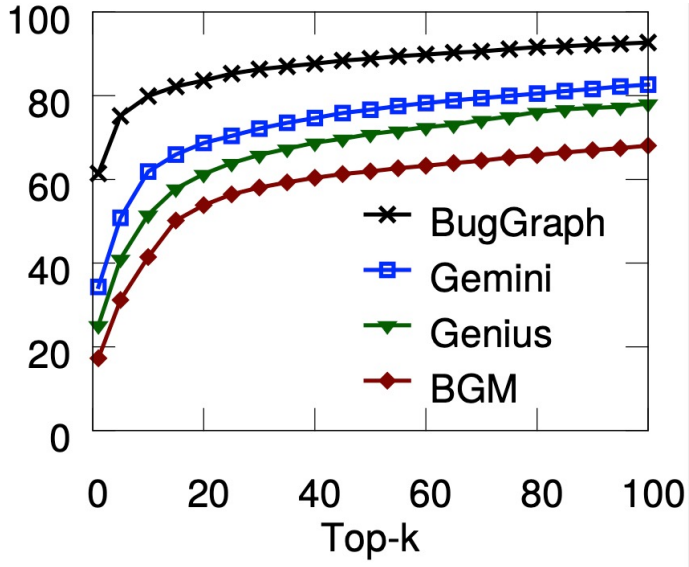
Source-Binary Code Similarity Test

- Type-2 Similar Code (top-5 hit rate)
  - BugGraph, 90%
  - Gemini, 74%
  - Genius, 54%
  - BGM, 40%
- Type-3 Similar Code (top-5 hit rate)
  - BugGraph, 75%
  - Gemini, 51%
  - Genius, 41%
  - BGM, 31%

Top-k Hit Rate of Type-2 Similar



Top-k Hit Rate of Type-3 Similar



Detecting Vulnerabilities from Firmware

- Tested six firmware of routers.
- Vulnerability database: 218 known vulnerable code
- Findings:
  - Identified *140* vulnerable code from 42 unique CVEs.
  - Old vulnerabilities still exist.

CVE	# Appear	Vulnerability type	CVE	# Appear	Vulnerability type	CVE	# Appear	Vulnerability type
2016-6303	5	out-of-bounds write	2016-0702	5	side-channel attack	2015-0206	4	Allow DoS attack
2016-6302	5	Allow DoS attack	2016-0701	2	Miss required crypto	2015-0205	4	Allows remote access
2016-2842	5	Out-of-bounds write	2015-3197	3	Man-in-the-middle	2015-0204	4	Downgrade attacks
2016-2182	5	Out-of-bounds write	2015-1794	2	segmentation fault	2014-8176	4	DoS overflow
2016-2180	5	Out-of-bounds read	2015-1792	4	Allows DoS attack	2014-5139	3	NULL pointer dereference
2016-2178	5	Side-channel attack	2015-1791	4	Double free	2014-3572	4	Downgrade attacks
2016-2176	2	buffer over-read	2015-1790	4	NULL pointer dereference	2014-3567	4	Allow DoS attack
2016-2109	2	memory consumption	2015-1789	4	Out-of-bounds read	2014-3511	1	Man-in-the-middle
2016-2105	2	heap memory corruption	2015-1788	4	Allow DoS attack	2014-3508	1	information leakage
2016-0799	5	Out-of-bounds read	2015-0292	4	Integer underflow	2014-3470	1	NULL pointer dereference
2016-0797	3	Integer overflow	2015-0288	4	NULL pointer dereference	2014-0221	1	Allow DoS attack
2016-0705	5	Double free	2015-0287	4	invalid write operation	2014-0198	1	NULL pointer dereference
2016-0704	4	Information leakage	2015-0286	4	invalid read operation	2014-0195	1	Buffer overflow
2016-0703	1	Man-in-the-middle	2015-0209	4	Use-after-free	2013-6449	1	Daemon crash

Summary of Applying Graph to Cybersecurity

- **Key Takeaway:**
  - *Identifying compilation provenance* can effectively improve code similarity detection.
  - *Ranking-based* graph triplet-loss network can cover different types of similar code.

- **Impact**:
  - Identified *140* vulnerable code from 6 firmware.

Thank You

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