Software Security Analysis from Automation to Intelligence

Yulei Sui

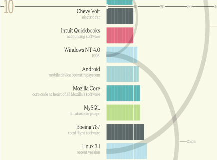
`http://yuleisui.github.io`

School of Computer Science
University of Technology Sydney, Australia

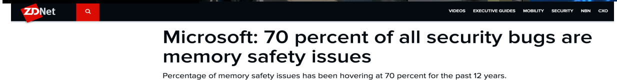
August 4, 2021

Modern System Software

Extremely large and complex but error-prone



More Complex!



Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x0000000A (0x0227001d, 0x00000002, 0x00000000, 0x804eba3a)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```

Memory Leaks

Buffer Overflows

Null Pointers

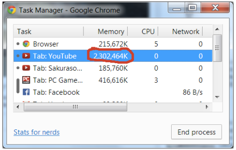
Use-After-Frees

Data-races

More Buggy!

Modern System Software

Extremely large and complex but error-prone



Task	Memory	CPU	Network
Browser	215,672K	5	0
Tab: YouTube	2,302,464K	0	0
Tab: Sakuraso...	185,760K	0	0
Tab: PC Game...	416,616K	3	0
Tab: Facebook			86 B/s

memory leaks

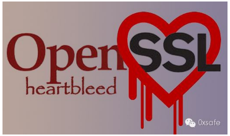
massive leaks over 2GB on a single browser tab

use-after-free

exploit price up to $100k per bug in Chrome

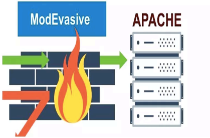
data race

11 civilians died



buffer overflow

66% websites affected



null pointer

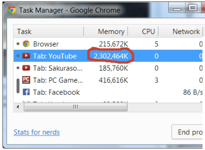
denial of service affecting millions of servers worldwide

uninitialized variables

password leakage via *tar* on Solaris OS

Modern System Software

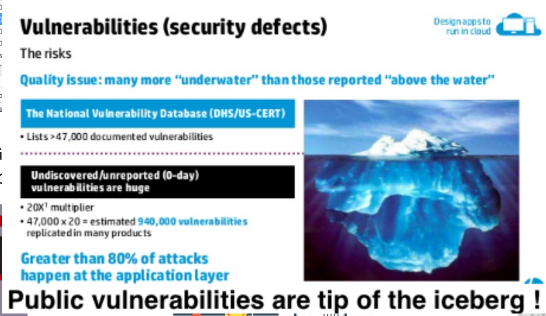
Extremely large and complex but error-prone



Task	Memory	CPU	Network
Browser	215,672K	5	0
Tab: YouTube	2,302,464K	0	0
Tab: Sakuraso...	185,760K	0	0
Tab: PC Game...	416,616K	3	0
Tab: Facebook			86 B/s

memory leaks

massive leaks over 2G on a single browser tab



Vulnerabilities (security defects)

The risks

Quality issue: many more "underwater" than those reported "above the water"

The National Vulnerability Database (DHS/US-CERT)

- Lists >47,000 documented vulnerabilities

Undiscovered/unreported (0-day) vulnerabilities are huge

- 20X^1 multiplier
- 47,000 x 20 = estimated 940,000 vulnerabilities replicated in many products

Greater than 80% of attacks happen at the application layer

Public vulnerabilities are tip of the iceberg !

data race

11 civilians died

buffer overflow

66% websites affected

null pointer

denial of service affecting millions of servers worldwide

uninitialized variables

password leakage via *tar* on Solaris OS

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
    - Foundation: SVF - value-flow analysis framework
    - Key features: sparse and on-demand analysis
    - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
    - Case 1: Boosting the performance of existing detectors
    - Case 2: Rapid prototyping via code embedding

Memory Leak

- A dynamically allocated object is not freed along some execution path of a program
- A major concern of long running server applications due to gradual loss of available memory.

```
/∗ CVE−2012−0817 allows remote attackers to cause a denial of service through adversarial connection requests.∗/
/∗ Samba −−libads/ldap.c:ads_leave_realm ∗/.

host = memAlloc(hostname);
 ...
 if (...) {...; return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.

```

```
/∗ A memory leak in Php−5.5.11 ∗/

for (...) {
    char∗ buf = readBuffer();
     if (condition)
       printf (buf);
    else
      continue;  // buf is leaked in else branch
    freeBuf(buf);
}
```

Buffer Overflow

- Attempt to put more data in a buffer than it can hold.
- Program crashes, undefined behavior or zero-day exploit[1].

```
/* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/

  void verfiyPassword(){
    char buff[15]; int pass = 0;
    printf ("\n Enter the password : \n");
    gets(buff);

    if (strcmp(buff, "thegeekstuff")){     // return non−zero if the two strings do not match
        printf ("\n Wrong Password \n");
    }
    else{                                   // return zero if two strings matched or a buffer overrun
        printf ("\n Correct Password \n");
        pass = 1;
    }
    if (pass)
        printf ("\n Root privileges given to the user \n");
  }

```

---

[1] Heartbleed, a well-known vulnerability in OpenSSL is also caused by buffer overflow (It took more than 2 years to discover and fix it since first patch, and over 500,000 websites were affected). Vulnerability is exploited when more data can be read than should be allowed.

Uninitialized Variable

- Stack variables in C and C++ are not initialized by default.
- Undefined behavior or denial of service via memory corruption

```
/* An uninitialized variable vulnerability simplified from gnuplot (CVE-2017-9670) */

void load(){
        switch (ctl) {
                case -1:
                        xN = 0; yN = 0;
                        break;
                case 0:
                        xN = i; yN = -i;
                        break;
                case 1:
                        xN = i + NEXT_SZ; yN = i - NEXT_SZ;
                        break;
                default:
                        xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
                        break;
        }
                plot(xN, yN);
}


```

Use-After-Free

- Attempt to access memory after it has been freed.
- Program crashes, undefined behavior or zero-day exploit.

```
/*  CVE−2015−6125 and CVE−2018−12377 with similar heap use after free patterns*/

char* msg = memAlloc(...);
...
if (err) {
        abrt = 1;
        ....
        free(msg); // the memory is released when an error occurs at server
}
...
if (abrt) {
        ...
        logError("operation aborted before commit", msg); // try to access released heap variable,
                                                          // causing either crash or writing confidential data
}
```

Data Race

- A data race occurs when two threads access the same memory concurrently and at least one of the accesses is for writing.
- Program crashes, undefined behavior and zero-day exploit.

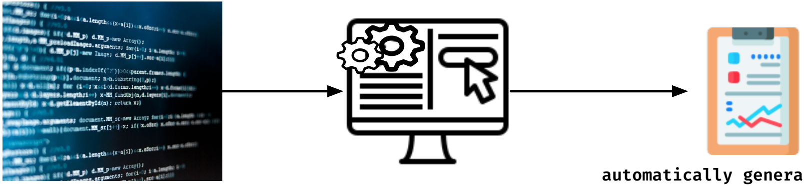
```
typedef std :: map<std::string, u32_int> map_t;

void *balance_Inquire(void *p) {
  map_t& m = *(map_t*)p;
  m["client"] = amount;     // map m is written in thread t
  return 0;
}

int main() {
  map_t m;
  pthread_t t;
  pthread_create(&t, 0, &balance_Inquire, &m);
  printf ("client =%d\n", m["client"]);          // map m is read in thread main
  pthread_join(t, 0);
}

```

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case 1: Boosting the performance of existing detectors
  - Case 2: Rapid prototyping via code embedding

What is Software/Program Analysis

- Software Analysis a.k.a Program analysis is the process of automatically analyzing the **behavior of computer programs** such as correctness, robustness, safety and security.
- Program analysis is to develop algorithms and tools which can **analyze other programs**



automatically generated report

What is Software/Program Analysis

- Software Analysis a.k.a Program analysis is the process of automatically analyzing the **behavior of computer programs** such as correctness, robustness, safety and security.
- Program analysis is to develop algorithms and tools which can **analyze other programs**
- Applications of program analysis
  - **Compiler optimizations**: transforming the source code to minimize a program's execution time, memory footprint, storage size, and power consumption
  - **Bug finding**: Identify the program or system that cause failure or produce an unexpected result
  - **Security vulnerability assessment**: Protect private users' data in databases
  - **Automatic Parallel Computation**: Guarantee the safe execution in different iterations on parallel calculations

Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it – inspecting source code by examining all possible program paths*
  - + Pin-point problems at source code level.
  - + Catch bugs at early the stage of the software development cycle.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it – inspecting source code by examining all possible program paths*
  - + Pin-point problems at source code level.
  - + Catch bugs at early the stage of the software development cycle.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

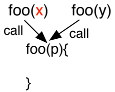
Levels of Abstractions

Assume x is a tainted value

```
p = x

p = y
```

flow-sensitivity

at which
program point
p is tainted?



context-sensitivity

under which
calling context
p is tainted?

```
if(cond)
      p = x
else
      p = y
```

path-sensitivity

along which
program path
p is tainted?

Static Analysis vs. Dynamic Analysis

Dynamic Analysis

- *Analyze a program at runtime – inspecting running program by examining some executable paths depending on specific test inputs*
  - + Identify bugs at runtime (catch it when you observe it).
  - + Zero or very low false alarm rates.
  - - Runtime overhead due to code instrumentation.
  - - May miss bugs (false negative) due to under-approximation.

Static Analysis vs. Dynamic Analysis

Dynamic Analysis

- *Analyze a program at runtime – inspecting running program by examining some executable paths depending on specific test inputs*
  - + Identify bugs at runtime (catch it when you observe it).
  - + Zero or very low false alarm rates.
  - - Runtime overhead due to code instrumentation.
  - - May miss bugs (false negative) due to under-approximation.

Instrumentations

p = x[i]

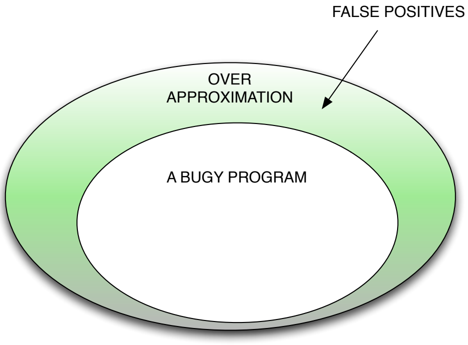
Observe_and_check (&x, i)

Check against
self-maintained
runtime meta-info

Limited Coverage

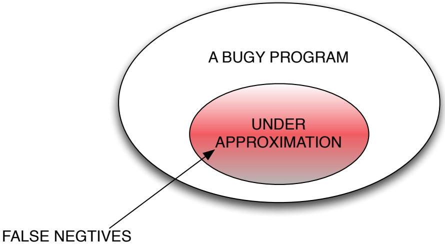
```
if(hard_to_satisfy)
        x=*p         // a null dereference
 else
        x= *q        // a safe dereference
```

Bug Detection Philosophy



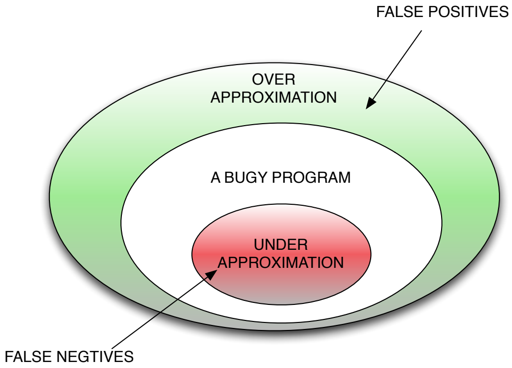
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy

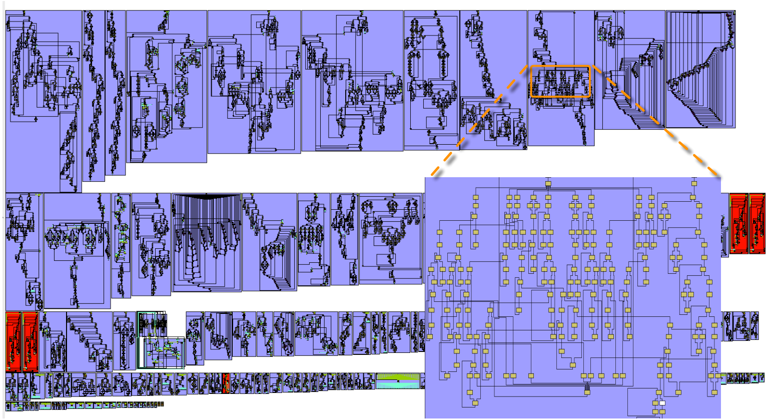


- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Some Static and Dynamic Analysis Tools

	open-source	**commercial**
static	deepsource Cppcheck sonarqube SVF-tool Splint Flawfinder FindBugs Semmle	MICRO FOCUS SYNOPSYS VERACODE DEEPCODE by snyk CodeScene™ Powered by Empear raxis information security kiuwan An Idera, Inc. Company RIPSTECH SOURCEBRELLA COVERITY BY SYNOPSYS CODESONAR
dynamic	KLEE LDRA Valgrind Iroh.js sanitizers Dmalloc Jalangi2	Trustwave SENTINEL Dynamic OWASP CHECKMARX acunetix HCL AppScan detectify

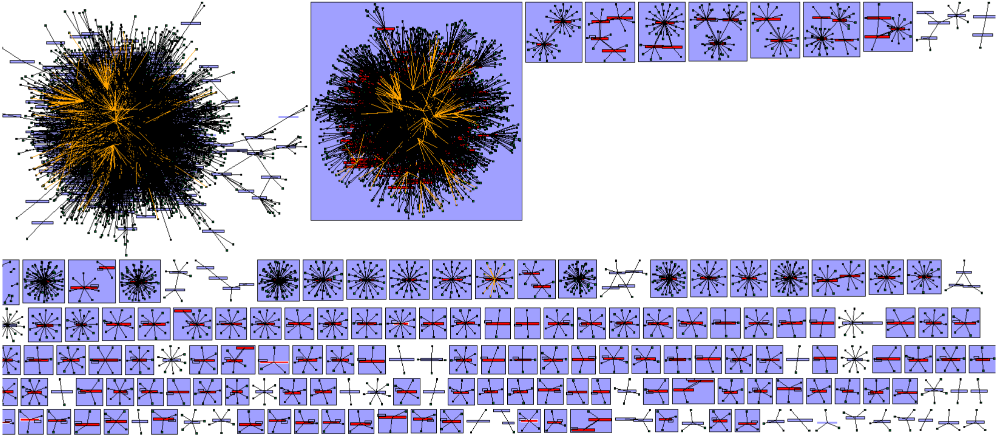
Whole-Program CFG of 300.twolf (20.5KLOC)



#functions: 194 #pointers: 20773 #loads/stores: 8657

Costly to reason about flow of values on CFGs!

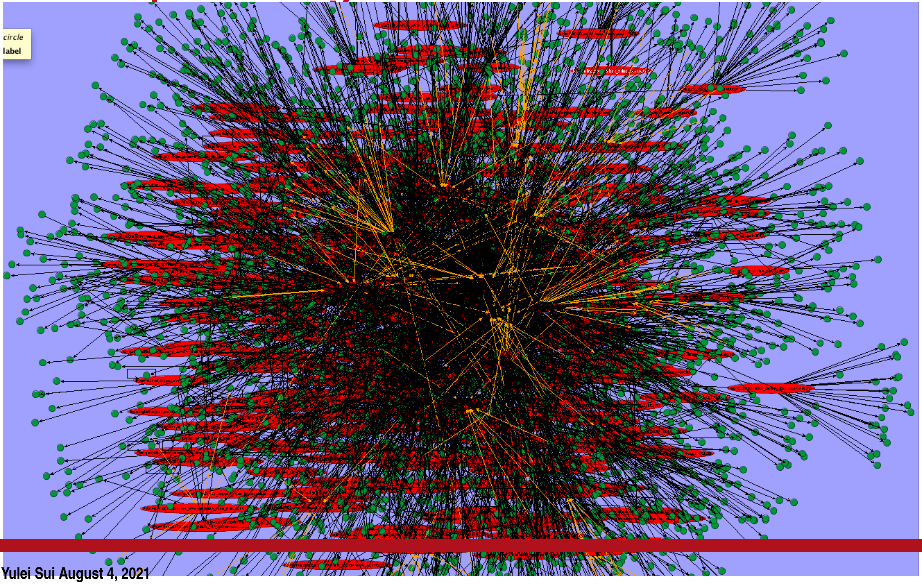
Call Graph of 176.gcc (230.5KLOC)



#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case 1: Boosting the performance of existing detectors
  - Case 2: Rapid prototyping via code embedding

SVF : Static Value-Flow Analysis

A sparse, selective and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100KLOC C/C++ code and **600+ stars with 32 contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, **2018 ICSE Distinguished Paper** , **2019 SAS Best Paper**, **2020 OOPSLA Distinguished Paper**.

SVF : Static Value-Flow Analysis

A sparse, selective and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100KLOC C/C++ code and **600+ stars with 32 contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, **2018 ICSE Distinguished Paper** , **2019 SAS Best Paper**, **2020 OOPSLA Distinguished Paper**.
- Value-Flow Analysis: resolves both control and data dependence.
  - Does the information generated at program point *A* flow to another program point *B* along some execution paths?
  - Can function *F* be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?

SVF : Static Value-Flow Analysis

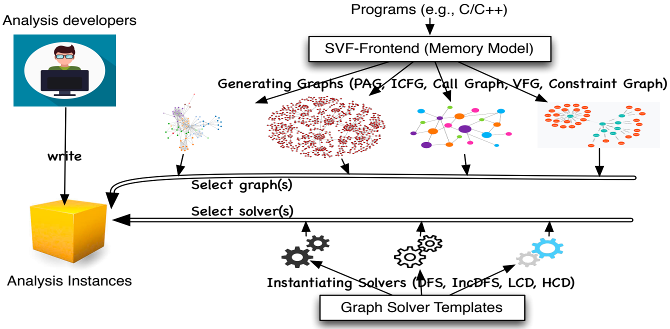
A sparse, selective and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100KLOC C/C++ code and **600+ stars with 32 contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, **2018 ICSE Distinguished Paper** , **2019 SAS Best Paper**, **2020 OOPSLA Distinguished Paper**.
- Value-Flow Analysis: resolves both control and data dependence.
  - Does the information generated at program point A flow to another program point B along some execution paths?
  - Can function F be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?
- Key features of SVF
  - Sparse: compute and maintain the data-flow facts where necessary
  - Selective : support mixed analyses for precision and efficiency trade-offs.
  - On-demand : reason about program parts based on user queries.

SVF : Static Value-Flow Analysis

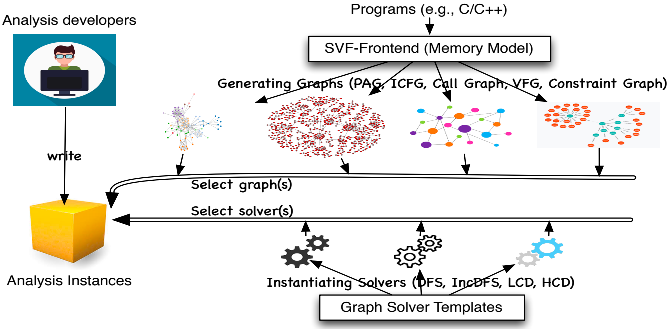
- SVF has been used and cited by researchers from leading program analysis and security groups, e.g. Chopped Symbolic Execution (from Imperial College London@ICSE'18 and @FSE'19), PinPoint (from HKUST@PLDI'18), Type-based CFI (from ACSAC'18@MIT and Northeastern University), Kernel Fuzzing (from Purdue@IEEE S&P'18), Directed Fuzzer (from NTU@CCS'18), K-Miner (from TU Darmstadt@NDSS'18), Permission Check Analysis (from Virginia Tech & Zhejiang University @USENIX Security'19), probabilistic analysis (from University of Pennsylvania @PLDI'19), and Hybrid Testing (from Northeastern University @S&P'20), and system call specialization (from Northeastern University @USENIX Security'20), and hot patch generation for kernels (from NTU @USENIX Security'20), and fuzzing for kernel file system (from Georgia Institute of Technology @S&P'20).

SVF: Design Principle



- Serving as an open-source foundation for building practical value-flow analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.

SVF: Design Principle



- Serving as an open-source foundation for building practical value-flow analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.
- Client applications:
  - Static bug detection (e.g., memory leaks, null dereferences, use-after-frees and data-races)
  - Accelerate dynamic analysis (e.g., Google's Sanitizers and AFL fuzzing)

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case 1: Boosting the performance of existing detectors
  - Case 2: Rapid prototyping via code embedding

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

```
p = & a

*p = & tainted

*p = & safe

q = *p
```

Flow-insensitive analysis

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

```
p = & a

*p = & tainted

*p = & safe

q = *p
```

p → a

a → tainted, safe

q → tainted, safe

false alarm!

Flow-insensitive analysis

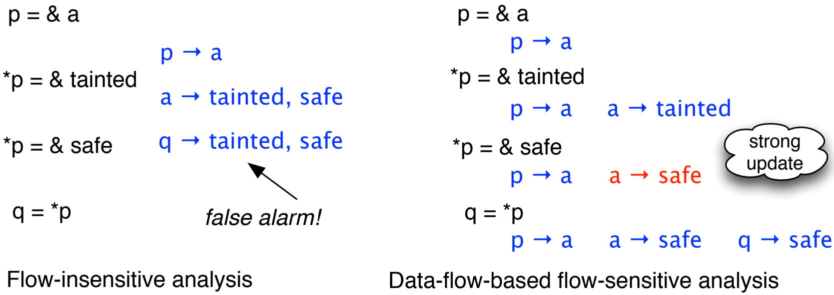
Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**



p = & a

p → a

*p = & tainted

a → tainted, safe

*p = & safe

q → tainted, safe

q = *p

false alarm!

Flow-insensitive analysis

p = & a

p → a

*p = & tainted

p → a   a → tainted

*p = & safe

p → a   a → safe

strong update

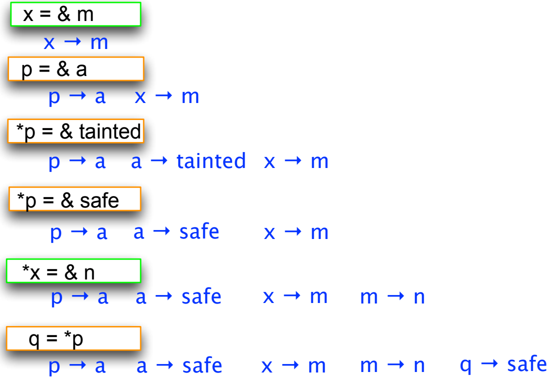
q = *p

p → a   a → safe   q → safe

Data-flow-based flow-sensitive analysis

The Data-flow-based Flow-Sensitive Analysis

- **Propagates points-to along the control-flow without knowing whether the information will be used there or not.**



x = & m

x → m

p = & a

p → a x → m

*p = & tainted

p → a a → tainted x → m

*p = & safe

p → a a → safe x → m

*x = & n

p → a a → safe x → m m → n

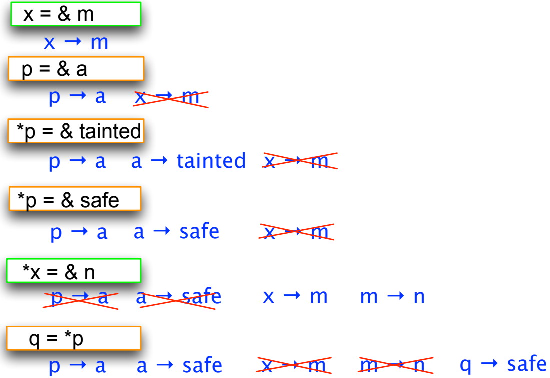
q = *p

p → a a → safe x → m m → n q → safe

Data-flow-based flow-sensitive analysis

The Data-flow-based Flow-Sensitive Analysis

- **Propagates points-to along the control-flow without knowing whether the information will be used there or not.**



x = & m

x → m

p = & a

p → a ~~x → m~~

*p = & tainted

p → a a → tainted ~~x → m~~

*p = & safe

p → a a → safe ~~x → m~~

*x = & n

~~p → a~~ ~~a → safe~~ x → m m → n

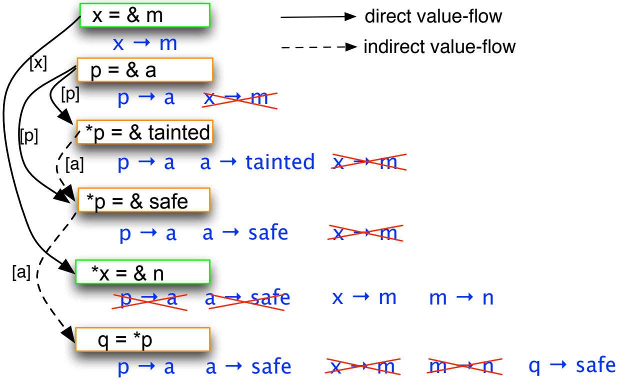
q = *p

p → a a → safe ~~x → m~~ ~~m → n~~ q → safe

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis (TSE’14, CC’16, TSE’18)

- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



direct value-flow

indirect value-flow

x = & m

x → m

p = & a

p → a ~~x → m~~

*p = & tainted

p → a a → tainted ~~x → m~~

*p = & safe

p → a a → safe ~~x → m~~

*x = & n

~~p → a~~ ~~a → safe~~ x → m m → n

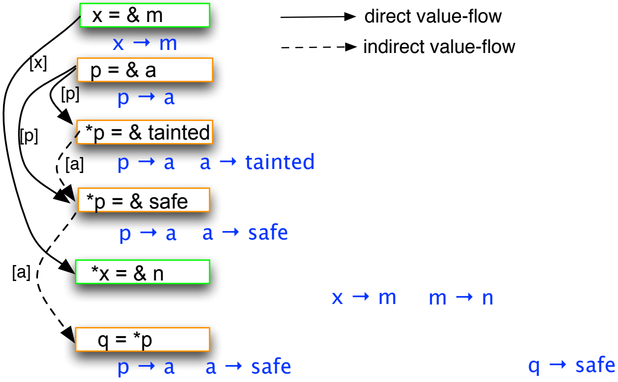
q = *p

p → a a → safe ~~x → m~~ ~~m → n~~ q → safe

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis (TSE’14, CC’16, TSE’18)

- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



Sparse flow-sensitive analysis

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case 1: Boosting the performance of existing detectors
  - Case 2: Rapid prototyping via code embedding
- **Research opportunities**

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17, ICSE'18 and ICSE'20)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17, ICSE'18 and ICSE'20)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%
- **Buffer Overflow Detection** (ISSRE'14 and TRel'16)
  - eliminate expensive runtime checks inside loops through static weakest precondition analysis
  - reduce the runtime overhead of SOFTBOUND from 77% to 47%

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17, ICSE'18 and ICSE'20)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%
- **Buffer Overflow Detection** (ISSRE'14 and TRel'16)
  - eliminate expensive runtime checks inside loops through static weakest precondition analysis
  - reduce the runtime overhead of SOFTBOUND from 77% to 47%
- **Control-Flow and Type Integrity** (ISSTA'17 and ISSRE'19)
  - pointer analysis to identify and remove spurious call targets by class hierarchy analysis to raise the bar against code reuse attacks.
  - reduce the sets of legitimate targets permitted at 20.3% of the virtual callsites in Chrome

Limitations of Conventional Program Analysis

- Performance
  - **Hard to balance between precision and scalability**
    - **False alarms** when using fast and **imprecise** Andersen's analysis, yielding 126,000 alarms for programs with 2 MLOC.
      – Imprecise handling of **complicated program features**, e.g., linked-list, loops and recursions.
    - **Long running time** when using **precise** flow- and context-sensitive analysis to analyze 2 MLOC for weeks.
      – `php-5.6.8`: **1,391 frees x 244,917 uses = 340 million pairs** with **billions** of calling contexts.

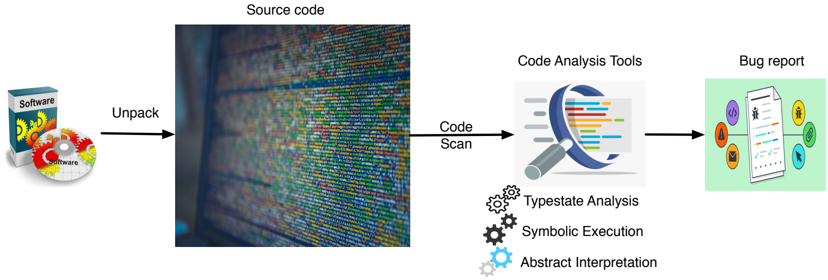
Limitations of Conventional Program Analysis

- Performance
  - **Hard to balance between precision and scalability**
    - **False alarms** when using fast and **imprecise** Andersen's analysis, yielding 126,000 alarms for programs with 2 MLOC.
      - Imprecise handling of **complicated program features**, e.g., linked-list, loops and recursions.
    - **Long running time** when using **precise** flow- and context-sensitive analysis to analyze 2 MLOC for weeks.
      - `php-5.6.8`: **1,391 frees x 244,917 uses = 340 million pairs** with **billions** of calling contexts.
- Applicability
  - **Lack of an unified approach to recognizing a wide variety of vulnerabilities**
    - Bugs may behave very differently and often **not simply manifest as memory errors or crashes** (e.g., misuse of APIs and inconsistent business logic) .
    - **Detecting each type of bugs needs to write their own detectors**, which relies on **domain experts** to define specific detection strategies.
    - Combination knowledge of programming language theories and **extensive engineering efforts**.

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case 1: Boost the performance of existing detectors
  - Case 2: Rapid prototyping via code embedding

Static Program Analysis for Bug Detection



Source code

Unpack

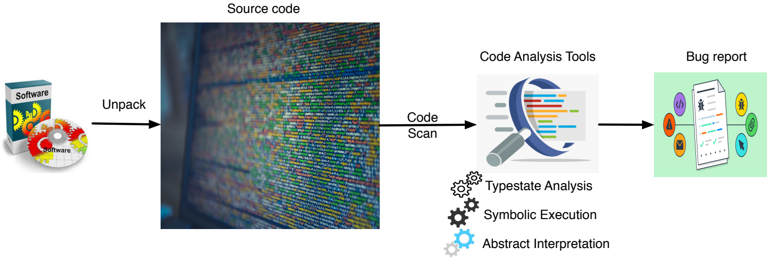
Code Scan

Code Analysis Tools

Bug report

- Typestate Analysis
- Symbolic Execution
- Abstract Interpretation

Static Program Analysis for Bug Detection



Source code

Unpack

Code Scan

Code Analysis Tools

Bug report

Typestate Analysis

Symbolic Execution

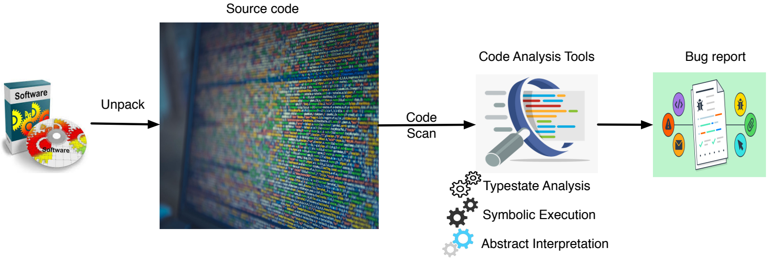
Abstract Interpretation

Challenges

1) Developing a static program analyser requires both deep programming theories and extensive engineering efforts
   --- Klee (https://klee.github.io/) started from 2008, it took ~10 years from publication, prototype and popular usage.

2) Program analysers for analysing large programs (MLOC) often over- or under- approximations, resulting in
   --- false alarms (imprecise)
   --- false negative and missing bugs (unsound)

New Paradigm for Software Security Analysis



Typestate Analysis

Symbolic Execution

Abstract Interpretation

Challenges

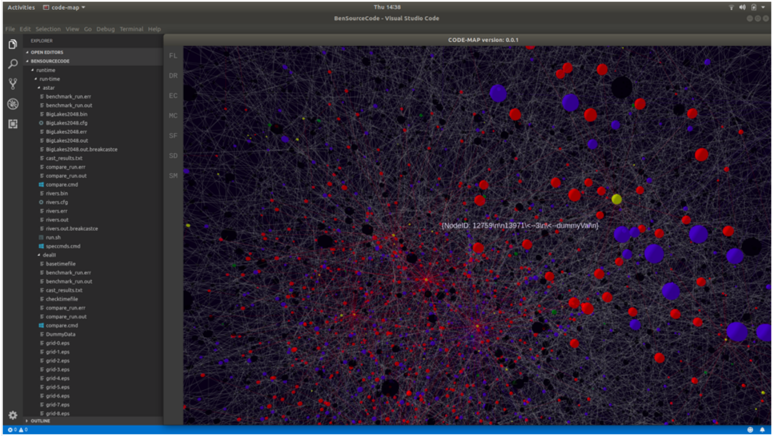
1) Developing a static program analyser requires both deep programming theories and extensive engineering efforts
   --- Klee (https://klee.github.io/) started from 2008, it took ~10 years from publication, prototype and popular usage.
2) Program analysers for analysing large programs (MLOC) often over- or under- approximations, resulting in
   --- false alarms (imprecise)
   --- false negative and missing bugs (unsound)

Opportunities

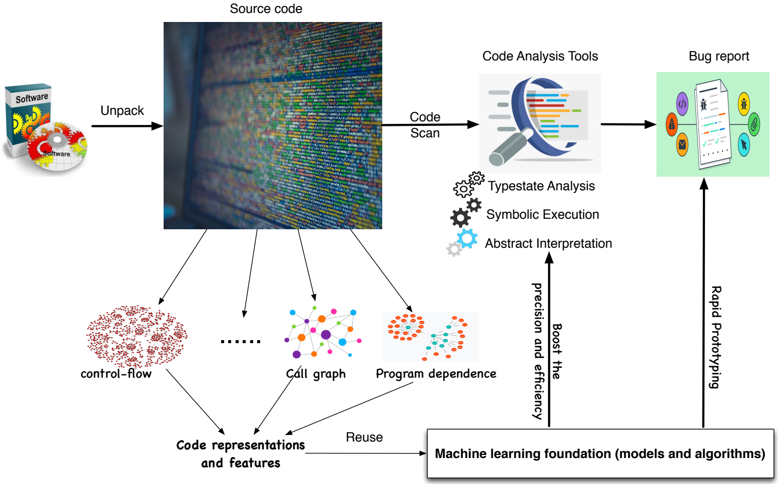
1) Our very own code analysis platform SVF (https://github.com/SVF-tools/SVF) with years-long efforts from 2014.
   --- publicly available with over 230 stars and 2k downloads, producing over 10 CORE-A/A* papers,
   --- plenary talk in EuroLLVM 2016, FSE Platinum Artifact Award 2016 and ICSE Distinguished Paper 2018
   --- used, cited and commented by leading research groups, Cambridge, UIUC, UCSB and Oracle.
2) New software security paradigm : code representation as "big data"
   --- control-flow graphs, data-flow graphs and abstract syntax trees

Code Representation

Program dependence graph of source code of OpenCV project (a computer vision library)



New Paradigm for Software Security Analysis



Source code

Unpack

Code Scan

Code Analysis Tools

Bug report

Typestate Analysis

Symbolic Execution

Abstract Interpretation

control-flow

……

Call graph

Program dependence

Boost the precision and efficiency

Rapid Prototyping

Code representations and features

Reuse

Machine learning foundation (models and algorithms)

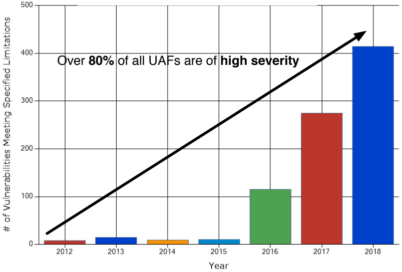
Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - Foundation: SVF - value-flow analysis framework
  - Key features: sparse and on-demand analysis
  - Applications: value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case study 1: Boost the performance of existing detectors (ACSAC 2017)
    - Machine-learning-guided type-state analysis to detect use-after-free vulnerabilities
  - Case study 2: Rapid prototyping via code embedding (OOPSLA 2020 and TOSEM 2021)
    - Code summarization, and vulnerability detection via code embedding

Temporal Temporal Safety Error: Use-After-Free

- Use-after-free, a.k.a, dangling pointer dereference, i.e., referencing a memory object after it has been released
- One of the most severe memory vulnerabilities
    - Crashes and data corruption
    - Information leakage
    - Control-flow hijacking

US National Vulnerability Database (NVD)



Over **80%** of all UAFs are of **high severity**

Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime
memory layout

Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p1;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

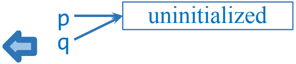
Runtime
memory layout

Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p ;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime
memory layout



Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p ;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime
memory layout

Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p ;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime
memory layout

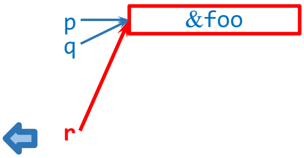
Memory freed and become available

Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p ;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime memory layout

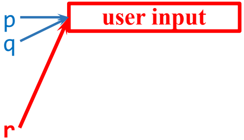


Use-After-Free Vulnerability

A simple attack model

```
typedef void (*func_ptr)();

void foo() {…}

int main() {
  func_ptr* p = malloc(4);
  func_ptr* q = p ;
  *p = &foo;
  free(p);
  long int* r = malloc(4);
  *r = userInput();
  (*q)();  // UAF bug
}
```

Runtime
memory layout



Related Work – Dynamic Approaches

- Detection
  - *Full memory safety*: e.g., CETS [ISMM'10]
  - *Taint tracking*: e.g., Undangle [ISSTA'12]
  - *Redzone*: e.g., AddressSanitizer [Usenix ATC'12]
  - *Optimization*: e.g., DangSan [EuroSys'17]
- Mitigation
  - *Safe allocator*: e.g., DieHarder [CCS'10], Cling [Security'10], FreeGuard [CCS'17]
  - *Safe deallocator*: e.g., VTPin [ACSAC'16]
  - *Nullification*: e.g., DangNull [NDSS'15], FreeSentry [NDSS'15]
  - *Control-flow integrity*: e.g., CFI [CCS'05], PathArmor [CCS'15], ShrinkWrap [ACSAC'15]

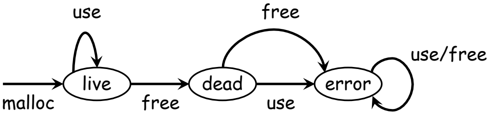
Related Work – Static Approaches

Early detection and zero runtime overhead

- *Buffer overflow* E.g., Archer [FSE'03], Marple [FSE'08], Parfait [FSE'10]
- *Memory leak* E.g., Saturn [FSE'05], FastCheck [PLDI'07], Saber [ISSTA'12]
- *Information flow* E.g., TAJ [PLDI'09], Merlin [PLDI'14], DroidSafe [NDSS'15]
- *Data race* E.g., RacerX [SOSP'03], LockSmith [PLDI'06], DroidRacer [PLDI'14]
- *UAF* **Relatively unexplored**

Typestate Analysis for Memory Safety

- A static approach using automata as the specification by capturing spatio-temporal correlations simultaneously.
  - **control-flow-reachability** (temporal property): free (p) can reach use(q) along control-flows
  - **pointer analysis** (spatial property): p and q are aliases (pointing to the same object)
- Spatio-temporal correlation is too strong for efficiently analysing large-size program.
  - – `php-5.6.8`: **1,391 frees x 244,917 uses = 340 million pairs** with **billions** of calling contexts.



Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen and Yulei Sui. *Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities.* (ICSE 2020)

Hua Yan, Yulei Sui, Shiping Chen and Jingling Xue. *Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities.* (ICSE 2018)

Hua Yan, Yulei Sui, Shiping Chen and Jingling Xue. *Machine-Learning-Guided Typestate Analysis for Use-After-Free Detection.* 33th Annual Computer Security Applications Conference (ACSAC 2017)

Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Insights: Leverage historical bug patterns and programming experience

Alike and predictable with some common characteristics

```
void fun() {
  ...
  for(...) {
    p = malloc(…);
    use(p); //Likely false UAF
    free(p);
  }
}
```

```
void fun() {
  ...
  p = malloc(…);
  for(...) {
    use(p); //Likely true UAF
    free(p);
  }
}
```

Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Insights: Leverage historical bug patterns and programming experience

```
void foo(Apple* p) {
  free(p);
}
...
void bar(Orange* q) {
  use(q);//Likely false UAF
}
```

```
void foo(Apple* p) {
  free(p);
}
...
void bar(Apple* q) {
  use(q);//Likely true UAF
}
```

Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Insights: Leverage historical bug patterns and programming experience

```
void fun() {
  ...
  if (Cnd) {
    free(p);
    p = null;
  }
  ...
  if (p != null) {
    use(p);//Likely false UAF
  }
}
```

```
void fun() {
  ...
  if (Cnd) {
    free(p);
    //p = null;
  }
  ...
  //if (p != null) {
    use(p);//Likely true UAF
  }
}
```

Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Insights: Leverage historical bug patterns and programming experience

```
1:  void foo(Apple* p) {
2:    free(p);
    }
    ...
3:  void bar(Apple* q) {
4:    use(q);//Likely false UAF
    }
```

Imprecise static
over-approximation

$pt(p) = \{o_1 , o_2, ..., \boldsymbol{o_{100}}\}$

$pt(q) = \{\boldsymbol{o_{100}} , o_{101}, ..., o_{200}\}$

```
1:  void foo(Apple* p) {
2:    free(p);
    }
    ...
3:  void bar(Apple* q) {
4:    use(q);//Likely true UAF
    }
```

Precise static
over-approximation

$pt(p) = \{\boldsymbol{o_1}\}$

$pt(q) = \{\boldsymbol{o_1}\}$

Machine-Learning-Guided Typestate Analysis

Feature Engineering

- 35 Features in 4 groups
- Type information
    - E.g., array, struct, container, global, type compatibility
- Control flow
    - E.g., loop, recursion, distance, dominance, use before free
- Common characteristics
    - E.g., nullification, flags, reallocation, address comparison
- Classification using machine learning
    - E.g., size of points-to set, #UAF@free, #UAF@use, #aliases, points-to cycles

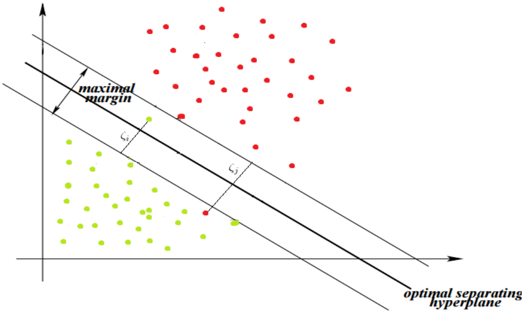
Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Feature Engineering: Leverage historical bug patterns and programming experience since many use-free patterns are alike and predictable with some common characteristics

Group	ID	Feature	Type	Description
Type Information	1	Array	Boolean	o is an array or an element of an array
	2	Struct	Boolean	o is a struct or an element of a struct
	3	Container	Boolean	o is a container (e.g., vector or map) or an element of a container
	4	IsLoad	Boolean	$use(q)$ is a load instruction
	5	IsStore	Boolean	$use(q)$ is a store instruction
	6	IsExtCall	Boolean	$use(q)$ is an external call
	7	GlobalFree	Boolean	$free(p)$, where p is a global pointer
	8	GlobalUse	Boolean	$use(q)$, where q is a global pointer
	9	CompatibleType	Boolean	p and q are type-compatible at $free(p)$ and $use(q)$
Control Flow	10	InSameLoop	Boolean	$free(p)$ and $use(q)$ are in the same loop
	11	InSameRecursion	Boolean	$free(p)$ and $use(q)$ are in the same recursion cycle
	12	#FunctionInBetween	Integer	number of functions in the shortest path from $free(p)$ to $use(q)$ in the program’s call graph
	13	DiffIteration	Boolean	$use(q)$ appears after $free(p)$ via a loop back-edge
	14	Dom	Boolean	$free(p)$ dominates $use(q)$
	15	PostDom	Boolean	$use(q)$ post-dominates $free(p)$
	16	#IndCalls	Integer	number of indirect calls in the shortest path from $free(p)$ to $use(q)$ in the program’s call graph
	17	UseBeforeFree	Boolean	a UAF pair, $free(p)$ and $use(q)$, is also a use-before-free
Common Programming Practices	18	NullifyAfterFree	Boolean	p is set to null immediately after $free(p)$
	19	ReturnConstInt	Boolean	a const integer is returned after $free(p)$
	20	ReturnBoolean	Boolean	a Boolean value is returned after $free(p)$
	21	Casting	Boolean	pointer casting is applied to q at $use(q)$
	22	ReAllocAfterFree	Boolean	p is redefined to point to a newly allocated object immediately after $free(p)$
	23	RefCounting	Boolean	o is an reference-counted object
	24	ValidatedFreePtr	Boolean	null checking for p before $free(p)$
	25	ValidatedUsePtr	Boolean	null checking for q before $use(q)$
Points-to Information	26	SizeOfPointsToSetAtFree	Integer	number of objects pointed to by p at $free(p)$
	27	SizeOfPointsToSetAtUse	Integer	number of objects pointed to by q at $use(q)$
	28	#UAFSharingSameFree	Integer	number of UAF pairs sharing the same $free(p)$
	29	#UAFSharingSameUse	Integer	number of UAF pairs sharing the same $use(q)$
	30	#Aliases	Integer	number of pointers pointing to o
	31	AllocInLoop	Boolean	o is allocated in a loop
	32	AllocInRecursion	Boolean	o is allocated in recursion
	33	LinkedList	Boolean	o is in a points-to cycle (signifying its presence in a linked-list)
	34	SamePointer	Boolean	p and q at $free(p)$ and $use(q)$ are the same pointer variable
	35	DefinedBeforeFree	Boolean	q at $use(q)$ is defined before $free(p)$

Machine-Learning-Guided Typestate Analysis

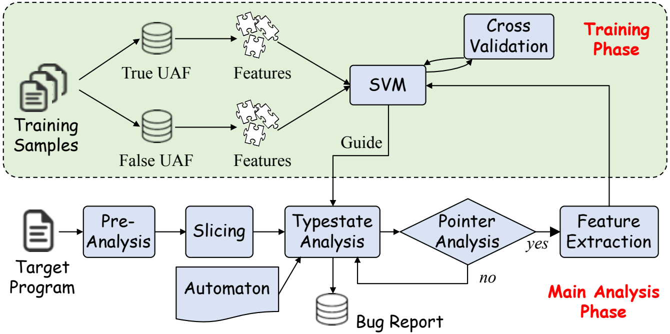
Support Vector Machine – Two-Class SVM



Figures shamelessly stolen from:
http://blog.hackerearth.com/simple-tutorial-svm-parameter-tuning-python-r

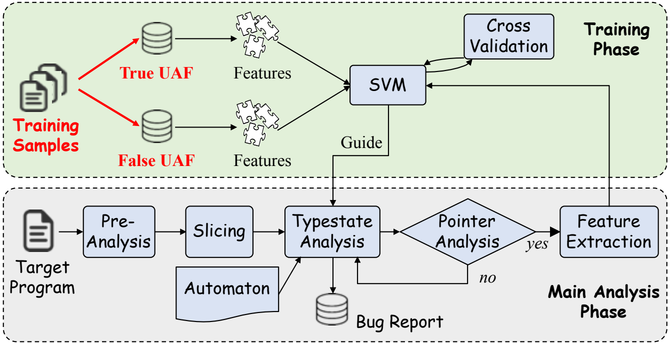
Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Framework



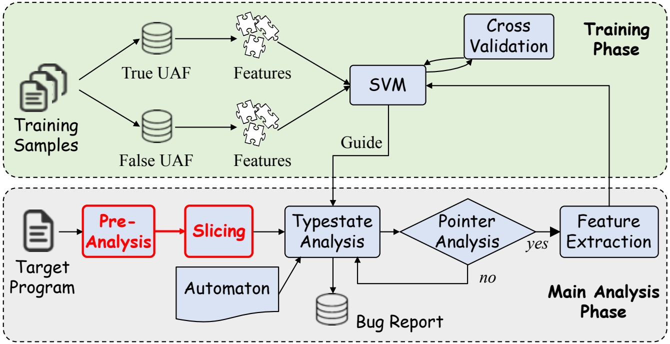
Machine-Learning-Guided Typestate Analysis (ACSAC '17)

Framework



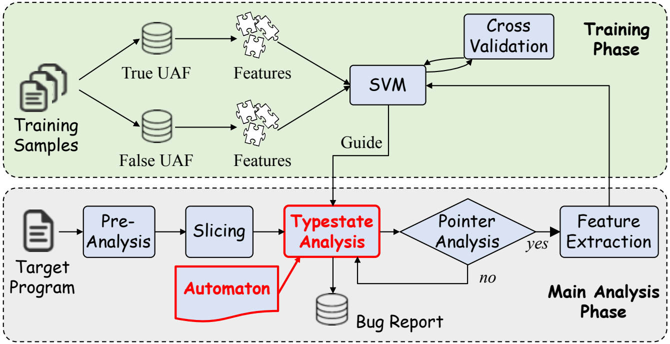
Machine-Learning-Guided Typestate Analysis (ACSAC '17)

Framework



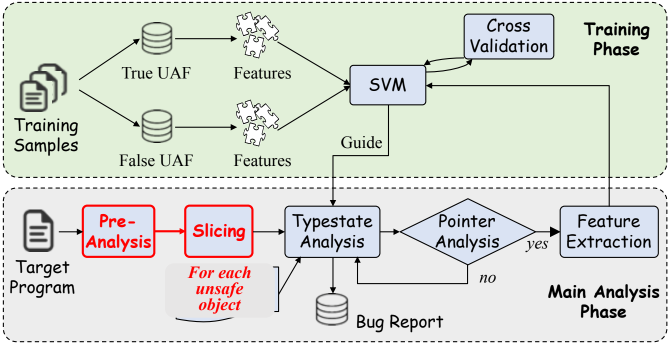
Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Framework



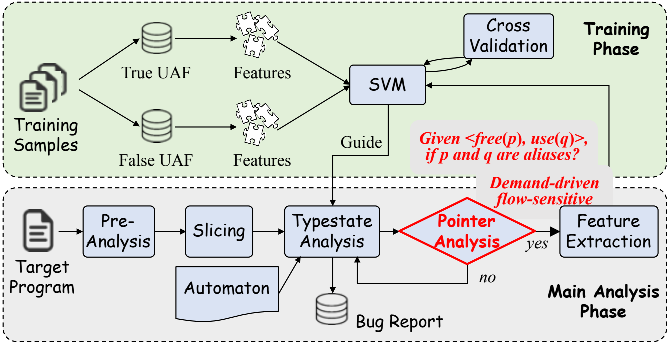
Machine-Learning-Guided Typestate Analysis (ACSAC ’17)

Framework



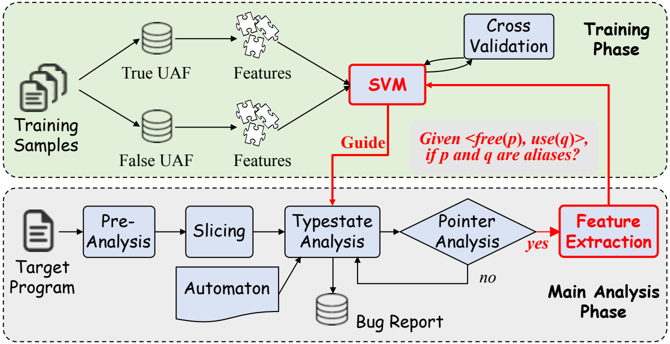
Machine-Learning-Guided Typestate Analysis (ACSAC '17)

Framework



Machine-Learning-Guided Typestate Analysis (ACSAC '17)

Framework

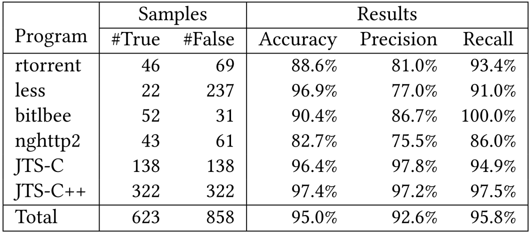


Platform

- Implemented based on our SVF framework [CC '16] and used our demand-driven pointer analysis [FSE '16]
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF` with over 2K downloads.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **230+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, **FSE Platinum Artifact Award 2016** and **ICSE Distinguished Paper 2018**.
  - Serves as a **foundation** for developing other analyses, with participants and contributors from both industry and academia, including UIUC, UCSB, IBM, Google, Qualcomm and Veracode.
- Third-party libraries
  - LLVM Compiler IR
  - Pointer Analysis [FSE '16]
  - SMT-solver z3
  - Machine learning libSVM

Machine-Learning-Guided Typestate Analysis

Training



Program	Samples		Results		
	#True	#False	Accuracy	Precision	Recall
rtorrent	46	69	88.6%	81.0%	93.4%
less	22	237	96.9%	77.0%	91.0%
bitlbee	52	31	90.4%	86.7%	100.0%
nghttp2	43	61	82.7%	75.5%	86.0%
JTS-C	138	138	96.4%	97.8%	94.9%
JTS-C++	322	322	97.4%	97.2%	97.5%
Total	623	858	95.0%	92.6%	95.8%

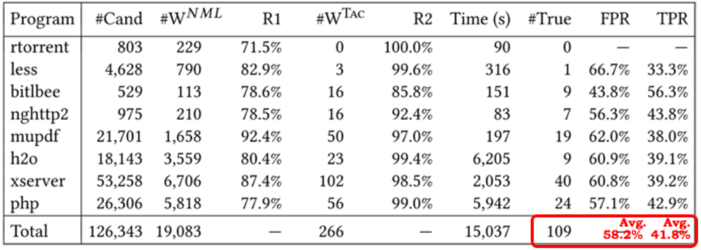
- True bugs training samples
  - Juliet Test Suite
  - Dynamically verify use-before-free instances
  - Manual injection
- False positive training samples
  - Juliet Test Suite
  - Tac-NML (typestate analysis without machine learning)
  - Manual inspection

Testing

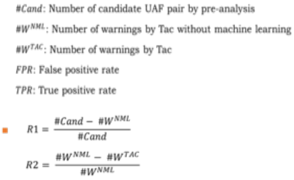
Program	Version	Language	LOC	#Frees	#Uses
rtorrent	0.96	C++	13,036	118	3,039
less	451	C	27,134	86	7,902
bitlbee	4.2	C	68,413	201	5,897
nghttp2	1.6.0	C++	71,387	29	7,566
mupdf	1.2.337	C++	122,481	253	105,911
h2o	1.7.2	C++	517,731	896	150,887
xserver	1.14.3	C	568,964	1,675	90,841
php	5.6.7	C	709,356	1,391	244,917
Total	—	—	2,098,502	4,649	616,960

Machine-Learning-Guided Typestate Analysis

Results



Program	#Cand	$\#W^{NML}$	R1	$\#W^{\text{Tac}}$	R2	Time (s)	#True	FPR	TPR
rtorrent	803	229	71.5%	0	100.0%	90	0	–	–
less	4,628	790	82.9%	3	99.6%	316	1	66.7%	33.3%
bitlbee	529	113	78.6%	16	85.8%	151	9	43.8%	56.3%
nghttp2	975	210	78.5%	16	92.4%	83	7	56.3%	43.8%
mupdf	21,701	1,658	92.4%	50	97.0%	197	19	62.0%	38.0%
h2o	18,143	3,559	80.4%	23	99.4%	6,205	9	60.9%	39.1%
xserver	53,258	6,706	87.4%	102	98.5%	2,053	40	60.8%	39.2%
php	26,306	5,818	77.9%	56	99.0%	5,942	24	57.1%	42.9%
Total	126,343	19,083	–	266	–	15,037	109	Avg. 58.2%	Avg. 41.8%



$\#Cand$: Number of candidate UAF pair by pre-analysis

$\#W^{NML}$: Number of warnings by Tac without machine learning

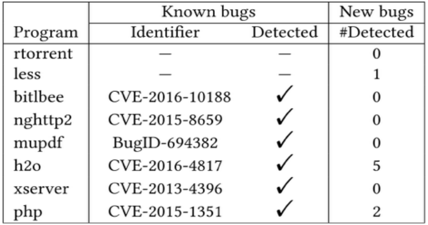
$\#W^{TAC}$: Number of warnings by Tac

FPR: False positive rate

TPR: True positive rate

- $R1 = \dfrac{\#Cand - \#W^{NML}}{\#Cand}$

$R2 = \dfrac{\#W^{NML} - \#W^{TAC}}{\#W^{NML}}$



Program	Known bugs		New bugs
	Identifier	Detected	#Detected
rtorrent	–	–	0
less	–	–	1
bitlbee	CVE-2016-10188	✓	0
nghttp2	CVE-2015-8659	✓	0
mupdf	BugID-694382	✓	0
h2o	CVE-2016-4817	✓	5
xserver	CVE-2013-4396	✓	0
php	CVE-2015-1351	✓	2

Outline

- **Existing software bugs and vulnerabilities**
- **Automated static analysis and dynamic analysis**
  - SVF: Value-flow analysis framework
  - Value-flow analysis to detect memory corruption errors
- **Learning-based software security analysis**
  - Case study 1: boost the performance of existing detectors
    - Machine-learning-guided type-state analysis to detect use-after-free vulnerabilities
  - Case study 2: rapid prototyping via code embedding
    - Code summarization, and vulnerability detection via code embedding

Code Embedding

- Distributed representation
  - Distributed representation of words (Word2Vec) and documents (Doc2Vec). Unlocking the potential of deep learning and NLP.
  - **Local representation** (object as a single representational element); **distributed representation** (object as a feature vector)

Object	Local representation	Distributed representation
`small apple`	1	[-0.2, -0.2, 0.0, 0.1]
`big apple`	2	[-0.1, -0.2, 0.0, 0.1]
`orange`	3	[-0.1, 0.5, 0.0, 0.3]
`car`	4	[0.0, 0.0, 0.5, 0.1]

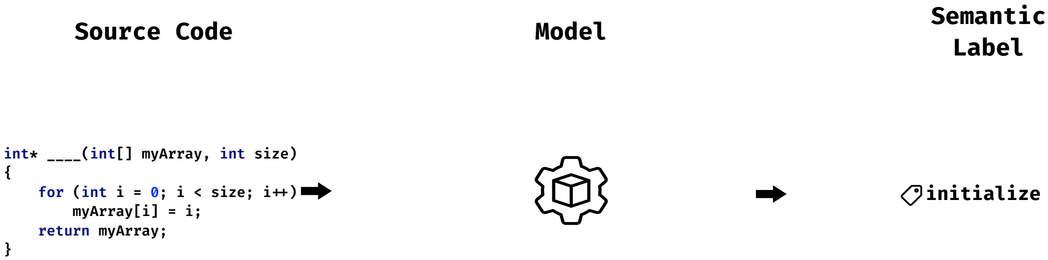
  - An object's meaning is **distributed across its vector components**. Semantically similar objects are mapped to close vectors.
- Code embedding
  - Learning **distributed vector representations for code** (e.g., via neural networks).
  - Capture **correlations** between **code snippets** and **code semantics** in a natural and effective manner.

Code Embedding

Source Code **Model** **Code Property Prediction**

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding



Source Code

Model

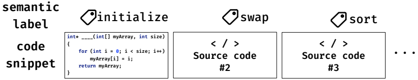
Semantic Label

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

initialize

Code Embedding

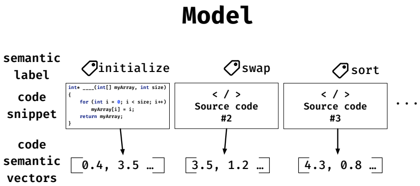
Model



```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding

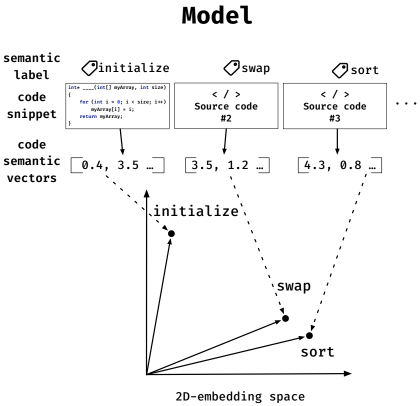
Code semantic vector in geometric space



```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding

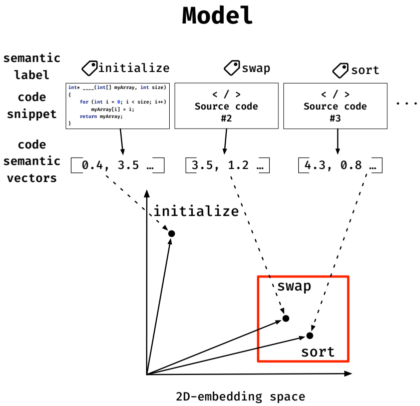
Code semantic vector in geometric space



```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding

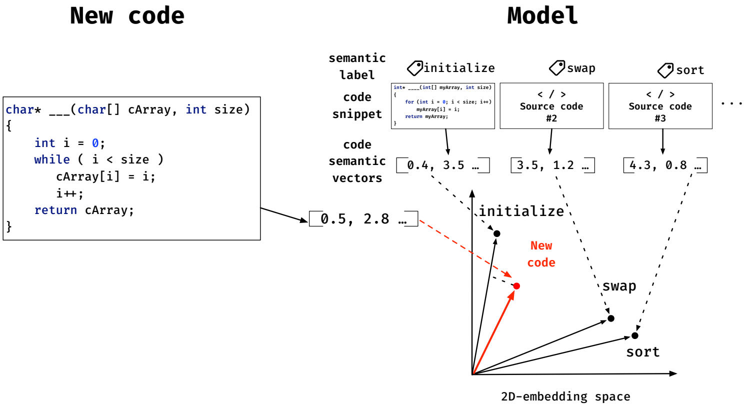
Code semantic vector in geometric space



```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding

Code semantic vector in geometric space



New code

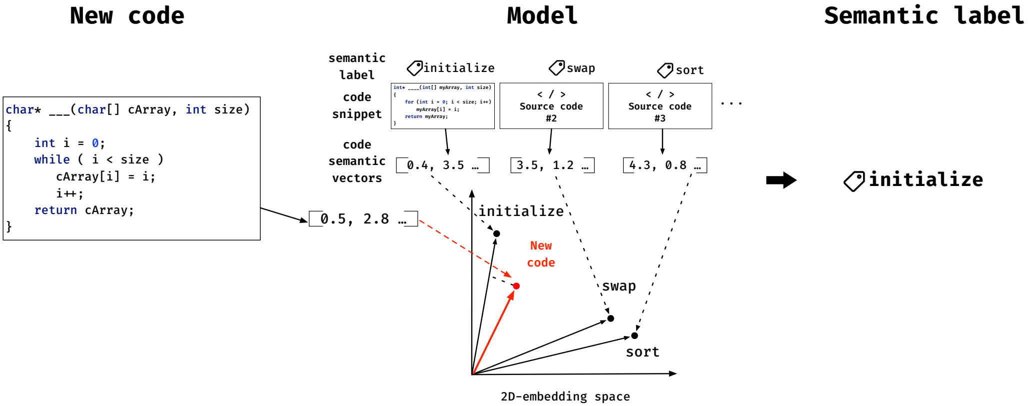
```
char* ___(char[] cArray, int size)
{
    int i = 0;
    while ( i < size )
       cArray[i] = i;
       i++;
    return cArray;
}
```

Model

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Code Embedding

Code semantic vector in geometric space



```
char* ___(char[] cArray, int size)
{
    int i = 0;
    while ( i < size )
       cArray[i] = i;
       i++;
    return cArray;
}
```

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Existing Embedding Approaches

Structure-oblivious embedding

Source code → **A bag of 'sentences'**[1,2]

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

---

[1] Distributed representations of words and phrases and their compositionality. In NeurIPS '13

[2] Distributed representations of sentences and documents. In ICML '14

Existing Embedding Approaches

Structure-oblivious embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

➡

A bag of ‘sentences’[1,2]

```
int*____(int[]myArray,int size)
```

[1] Distributed representations of words and phrases and their compositionality. In NeurIPS ’13

[2] Distributed representations of sentences and documents. In ICML ‘14

Existing Embedding Approaches

Structure-oblivious embedding

Source code

A bag of 'sentences'[1,2]

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

➡

int	*	____	(	int	[]	myArray	,	int	size	)

[1] Distributed representations of words and phrases and their compositionality. In NeurIPS '13

[2] Distributed representations of sentences and documents. In ICML '14

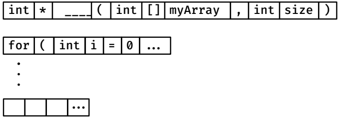
Existing Embedding Approaches

Structure-oblivious embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

A bag of ‘sentences’[1,2]



int | * | ____ | (| int | [] | myArray | , | int | size |)

for | (| int | i | = | 0 | ...

.
.
.

...

[1] Distributed representations of words and phrases and their compositionality. In NeurIPS ’13

[2] Distributed representations of sentences and documents. In ICML ‘14

Existing Embedding Approaches

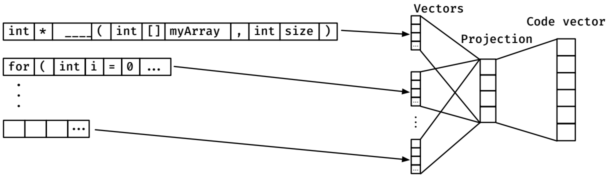
Structure-oblivious embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

A bag of ‘sentences’ [1,2]

Embedding



[1] Distributed representations of words and phrases and their compositionality. In NeurIPS ’13

[2] Distributed representations of sentences and documents. In ICML ‘14

Existing Embedding Approaches

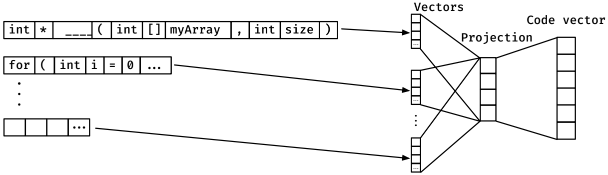
Structure-oblivious embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}

char* ___(char[] cArray, int size)
{
    int i = 0;
    while ( i < size )
       cArray[i] = i;
       i++;
    return cArray;
}
```

A bag of ‘sentences’ [1,2]

Embedding



Textually different but semantically equivalent

[1] Distributed representations of words and phrases and their compositionality. In NeurIPS ’13

[2] Distributed representations of sentences and documents. In ICML ‘14

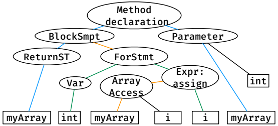
Existing Embedding Approaches

Structure-preserving embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

Abstract Syntax Tree [3]



---

[3] code2vec: Learning distributed representations of code. POPL .2019

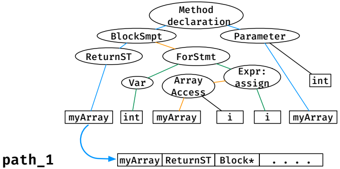
Existing Embedding Approaches

Structure-preserving embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

A bag of ‘paths’ on AST[3]



---

[3] code2vec: Learning distributed representations of code. POPL .2019

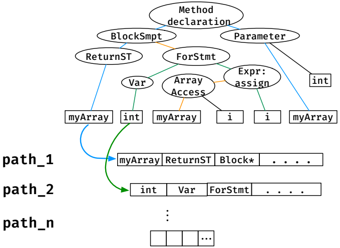
Existing Embedding Approaches

Structure-preserving embedding

Source code

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```

A bag of ‘paths’ on AST [3]



---

[3] code2vec: Learning distributed representations of code. POPL .2019

Existing Embedding Approaches

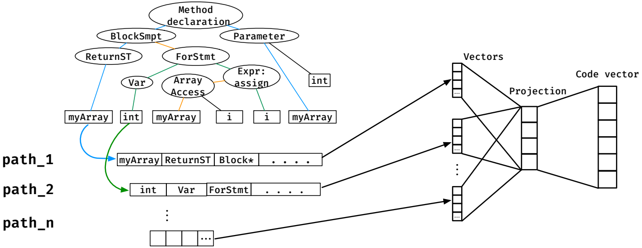
Structure-preserving embedding

Source code

A bag of ‘paths’ on AST[3]

Embedding

```
int* ____(int[] myArray, int size)
{
    for (int i = 0; i < size; i++)
        myArray[i] = i;
    return myArray;
}
```



[3] code2vec: Learning distributed representations of code. POPL .2019

Problems and Limitations

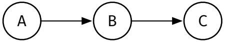
(a) Fail to capture asymmetric transitivity

(b) Alias-unaware

(c) Intraprocedural/context-insensitivity

Problems and Limitations

(a) Fail to capture asymmetric transitivity



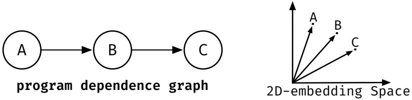
program dependence graph

(b) Alias-unaware

(c) Intraprocedural/context-insensitivity

Problems and Limitations

(a) Fail to capture asymmetric transitivity



$V_A \cdot V_C^T > 0$

A → B → C

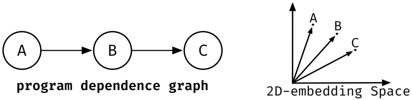
C → B → A

(b) Alias-unaware

(c) Intraprocedural/context-insensitivity

Problems and Limitations

(a) Fail to capture asymmetric transitivity



$V_A \cdot V_C^T > 0$

A → B → C ✓ **Real reachability and correctly preserved**

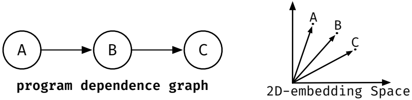
C → B → A ✗ **Spurious reachability but imprecisely preserved**

(b) Alias-unaware

(c) Intraprocedural/context-insensitivity

Problems and Limitations

(a) Fail to capture asymmetric transitivity

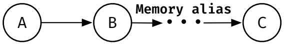


$V_A \cdot V_C^T > 0$

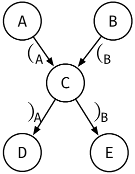
A → B → C ✓ Real reachability and correctly preserved

C → B → A ✗ Spurious reachability but imprecisely preserved

(b) Alias-unaware

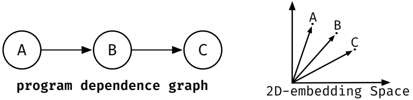


(c) Intraprocedural/context-insensitivity



Problems and Limitations

(a) Fail to capture asymmetric transitivity

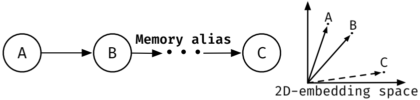


$V_A \cdot V_C^T > 0$

A → B → C ✓ Real reachability and correctly preserved

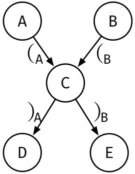
C → B → A ✗ Spurious reachability but imprecisely preserved

(b) Alias-unaware



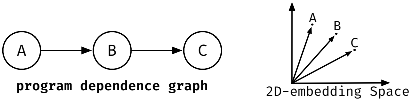
$V_A \cdot V_C^T < 0$ A → B ↛ C ✗ Real reachability but unsoundly preserved

(c) Intraprocedural/context-insensitivity



Problems and Limitations

(a) Fail to capture asymmetric transitivity

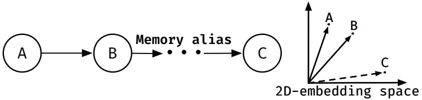


$V_A \cdot V_C^T > 0$

A → B → C ✓ **Real reachability and correctly preserved**

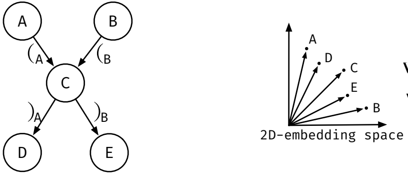
C → B → A ✗ **Spurious reachability but imprecisely preserved**

(b) Alias-unaware



$V_A \cdot V_C^T < 0$ A → B ↛ C ✗ **Real reachability but unsoundly preserved**

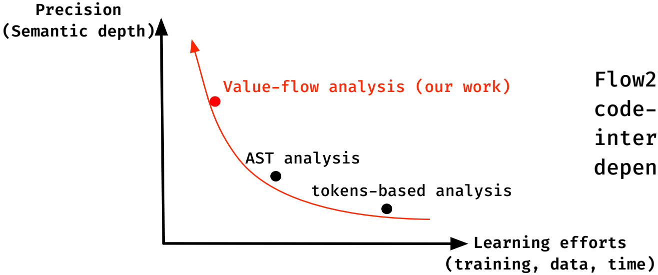
(c) Intraprocedural/context-insensitivity



$V_A \cdot V_D^T > 0$ A → C → D ✓ **Real reachability and correctly preserved**

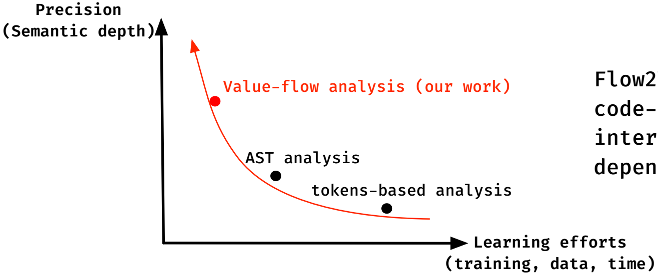
$V_A \cdot V_E^T > 0$ A → C → E ✗ **Spurious reachability but imprecisely preserved**

The Aim of This Work (OOPSLA ’20)

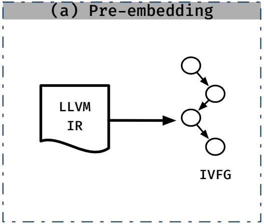


Flow2Vec: a high-order proximity code-embedding approach by preserving interprocedural alias-aware program dependence

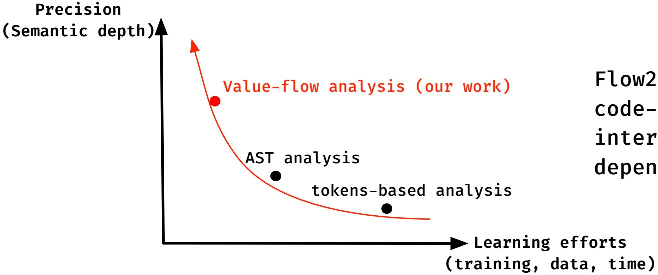
The Aim of This Work (OOPSLA ’20)



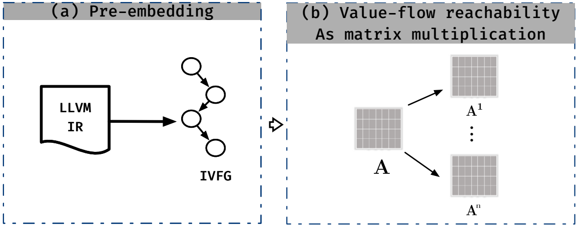
Flow2Vec: a high-order proximity code-embedding approach by preserving interprocedural alias-aware program dependence



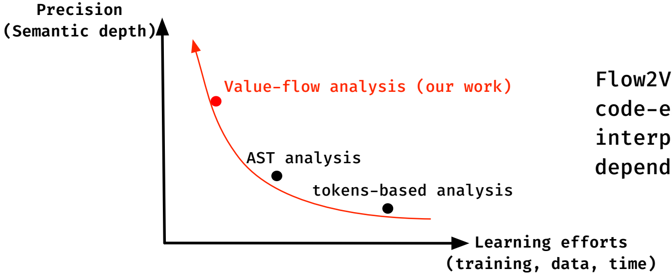
The Aim of This Work (OOPSLA ’20)



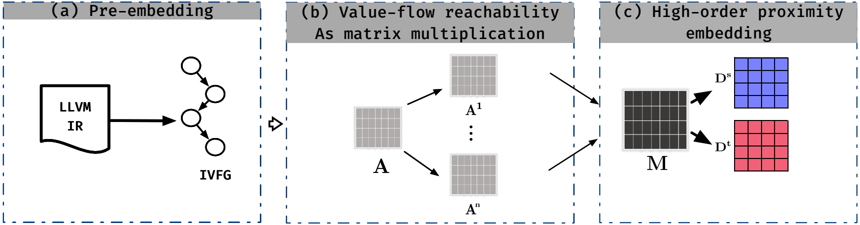
Flow2Vec: a high-order proximity code-embedding approach by preserving interprocedural alias-aware program dependence



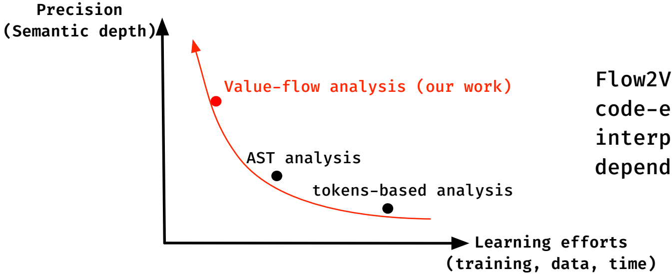
The Aim of This Work (OOPSLA ’20)



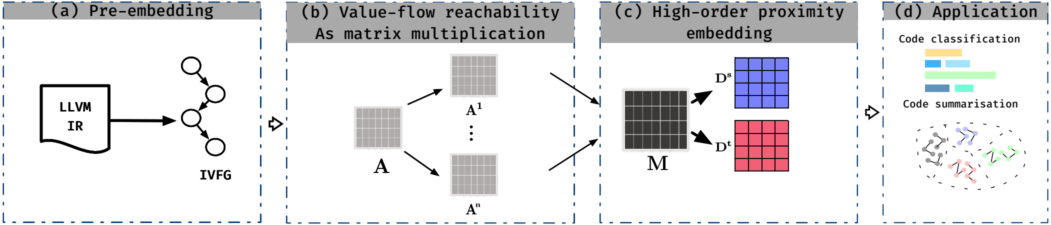
Flow2Vec: a high-order proximity code-embedding approach by preserving interprocedural alias-aware program dependence



The Aim of This Work (OOPSLA ’20)



Flow2Vec: a high-order proximity code-embedding approach by preserving interprocedural alias-aware program dependence



A Motivating Example

Phase (a) Pre-embedding

```
    foo(){
l1:   stack = malloc(...);
l2:   queue = malloc(...);
l3:   p = initialize(stack); // cs1
l4:  q = initialize(queue); // cs2


     ......
    }
l5: initialize(x){
      // initialization for
     // objects that 'x' points to
     ......
l6:  return x;
    }
```

A Motivating Example

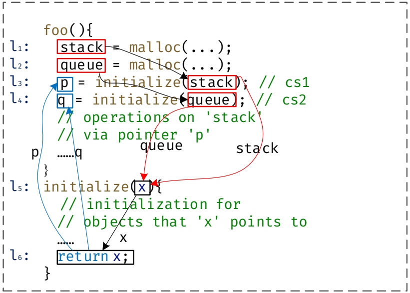
Phase (a) Pre-embedding

```
     foo(){
l1:    stack = malloc(...);
l2:    queue = malloc(...);
l3:    p = initialize(stack); // cs1
l4:   q = initialize(queue); // cs2
       // operations on 'stack'
      // via pointer 'p'
      ……
     }
l5:  initialize(x){
       // initialization for
      // objects that 'x' points to
      ……
l6:   return x;
     }
```

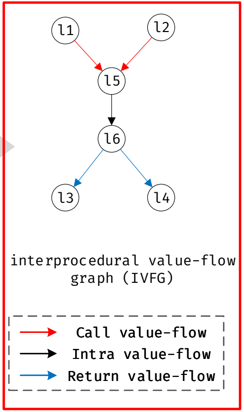
Call site 1	p refers to stack
Call site 2	q refers to queue

A Motivating Example

Phase (a) Pre-embedding



```
    foo(){
l1:   stack = malloc(...);
l2:   queue = malloc(...);
l3:   p = initialize(stack); // cs1
l4:   q = initialize(queue); // cs2
      // operations on 'stack'
      // via pointer 'p'
      ......q
    }
l5: initialize(x){
      // initialization for
      // objects that 'x' points to
      ......
l6:   return x;
    }
```

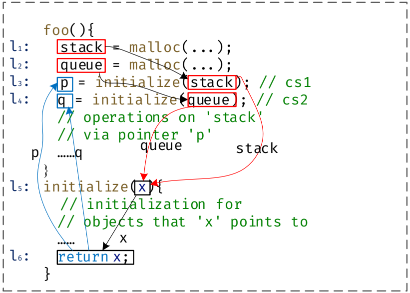


interprocedural value-flow graph (IVFG)

Call site 1	p refers to stack
Call site 2	q refers to queue

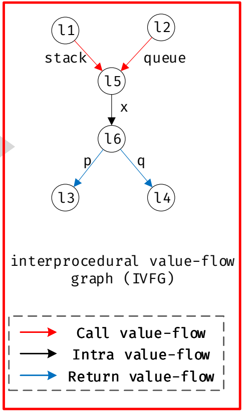
A Motivating Example

Phase (a) Pre-embedding



```
    foo(){
l1:    stack = malloc(...);
l2:    queue = malloc(...);
l3:    p = initialize(stack); // cs1
l4:    q = initialize(queue); // cs2
       // operations on 'stack'
       // via pointer 'p'
       ......
    }
l5: initialize(x){
       // initialization for
       // objects that 'x' points to
       ......
l6:    return x;
    }
```

Call site 1	p refer to stack
Call site 2	q refer to queue

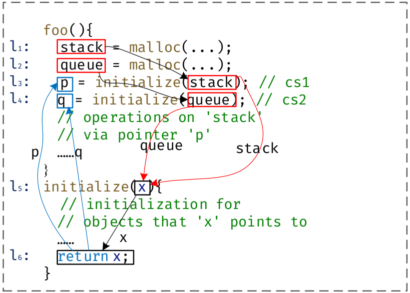


interprocedural value-flow graph (IVFG)

- Call value-flow
- Intra value-flow
- Return value-flow

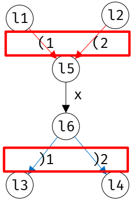
A Motivating Example

Phase (a) Pre-embedding



```
    foo(){
l1:   stack = malloc(...);
l2:   queue = malloc(...);
l3:   p = initialize(stack); // cs1
l4:   q = initialize(queue); // cs2
      // operations on 'stack'
      // via pointer 'p'
      ……q
    }
l5: initialize(x){
      // initialization for
      // objects that 'x' points to
      ……
l6:   return x;
    }
```

Call site 1	p refers to stack
Call site 2	q refers to queue

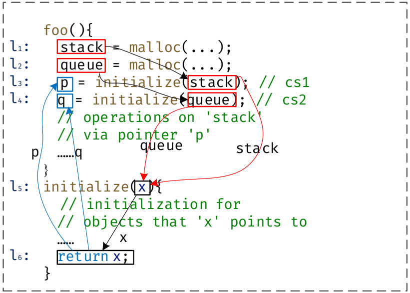


interprocedural value-flow graph (IVFG)

→ Call value-flow
→ Intra value-flow
→ Return value-flow

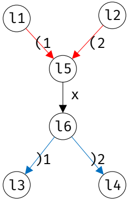
A Motivating Example

Phase (a) Pre-embedding



```
   foo(){
l1:   stack = malloc(...);
l2:   queue = malloc(...);
l3:   p = initialize(stack); // cs1
l4:   q = initialize(queue); // cs2
      // operations on 'stack'
      // via pointer 'p'
      ......
   }
l5: initialize(x){
      // initialization for
      // objects that 'x' points to
      ......
l6:   return x;
   }
```

Call site 1	p refers to stack
Call site 2	q refers to queue



interprocedural value-flow graph (IVFG)

- → Call value-flow
- → Intra value-flow
- → Return value-flow

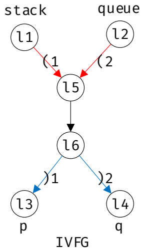
$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



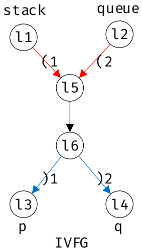
$$\begin{array}{c|cccccc}
 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\
\hline
l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\
l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\
l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\
l_6 & 0 & 0 &)_1 &)_2 & 0 & 0
\end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

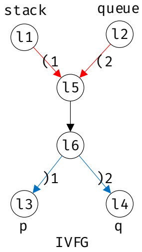
Symbolic
Adjacency matrix

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$$\mathbf{A}$$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_1{*}1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2{*}1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1{*})_1 & 1{*})_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$$

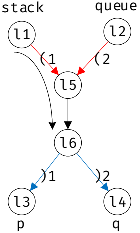
Matrix multiplication

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

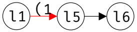
Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & \boxed{(_1 * 1} \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2 * 1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1 *)_1 & 1 *)_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

$$\mathbf{A}^2_{1,6} = (_1 * 1$$

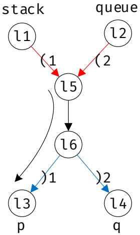


$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_1{*}1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2{*}1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & \boxed{1{*})_1} & 1{*})_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

$\mathbf{A}^2_{1,6} = (_1{*}1 \quad l_1 \xrightarrow{(_1} l_5 \rightarrow l_6$

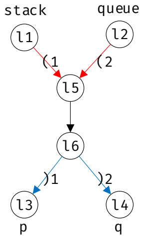
$\boxed{\mathbf{A}^2_{5,3} = 1{*})_1 \quad l_5 \rightarrow l_6 \xrightarrow{)_1} l_3}$

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$$\mathbf{A}$$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_1 \ast 1 \ast)_1 & (_1 \ast 1 \ast)_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2 \ast 1 \ast)_1 & (_2 \ast 1 \ast)_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

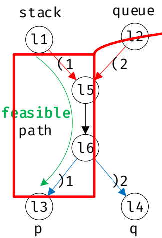
Matrix multiplication

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & \boxed{(_1 * 1 *)_1} & (_1 * 1 *)_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2 * 1 *)_1 & (_2 * 1 *)_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

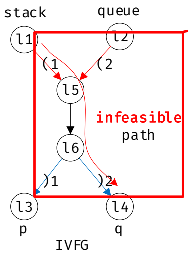
$\mathbf{A}^3_{1,3} = (_1 * 1 *)_1$ $\;l1 \xrightarrow{(1} l5 \rightarrow l6 \xrightarrow{)1} l3$ feasible path of length 3

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_1{*}1{*})_1 & (_1{*}1{*})_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2{*}1{*})_1 & (_2{*}1{*})_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

$$\mathbf{A}^3_{1,3} = (_1{*}1{*})_1 \quad l_1 \xrightarrow{(_1} l_5 \rightarrow l_6 \xrightarrow{)_1} l_3$$

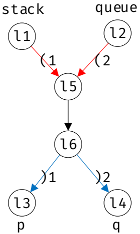
$$\mathbf{A}^3_{1,4} = (_1{*}1{*})_2 \quad l_1 \xrightarrow{(_1} l_5 \rightarrow l_6 \xrightarrow{)_2} l_4$$ infeasible path of length 3

$$\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$$

The power of matrix
h-th order reachability

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

Symbolic Adjacency matrix

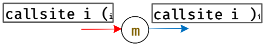
$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & \boxed{(_1 * 1 *)_1} & \boxed{\cancel{(_1 * 1 *)_2}} & 0 & 0 \\ l_2 & 0 & 0 & \boxed{\cancel{(_2 * 1 *)_1}} & \boxed{(_2 * 1 *)_2} & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

$$L : C = CC \mid (_1\ C\ )_1 \mid \ldots \mid (_n\ C\ )_n \mid \epsilon$$

context-free language

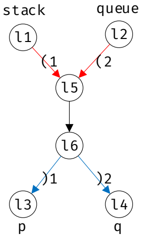


entering a method m from callsite i $(_i$

and exits from m back to callsite i $)_i$

A Motivating Example

Phase (b) Value-flow reachability as matrix multiplication



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$$\mathbf{A}$$

Symbolic Adjacency matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & \boxed{1} & \boxed{0} & 0 & 0 \\ l_2 & 0 & 0 & \boxed{0} & \boxed{1} & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

Matrix multiplication

A Motivating Example

Phase (c) High-order proximity embedding

$$\underbrace{\underset{\mathbf{A}}{\begin{array}{c|cccccc|} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}} \qquad \underset{\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}}{\begin{array}{c|cccccc|} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_1*1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2*1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1*)_1 & 1*)_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}} \qquad \underset{\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}}{\begin{array}{c|cccccc|} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_1*1*)_1 & (_1*1*)_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2*1*)_1 & (_2*1*)_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}}}$$

$$\boxed{\boxed{\texttt{Katz Index}} \quad \mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h = 0.8 \cdot \mathbf{A} + 0.8^2 \cdot \mathbf{A}^2 + 0.8^3 \cdot \mathbf{A}^3}$$

A Motivating Example

Phase (c) High-order proximity embedding

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_{1} & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_{2} & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_{1} &)_{2} & 0 & 0 \end{array}$$

$$\mathbf{A}$$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_{1}*1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_{2}*1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1*)_{1} & 1*)_{2} & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_{1}*1*)_{1} & (_{1}*1*)_{2} & 0 & 0 \\ l_2 & 0 & 0 & (_{2}*1*)_{1} & (_{2}*1*)_{2} & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

Katz Index

$$\mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h = \boxed{0.8} \cdot \mathbf{A} + \boxed{0.8^2} \cdot \mathbf{A}^2 + \boxed{0.8^3} \cdot \mathbf{A}^3$$

Context-sensitive value-flow reachability matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & \boxed{0.512*(_{1}*1*)_{1}} & \cancel{0.512*(_{1}*1*)_{2}} & 0.8*(_{1} & 0.64*(_{1}*1 \\ l_2 & 0 & 0 & \cancel{0.512*(_{2}*1*)_{1}} & \boxed{0.512*(_{2}*1*)_{2}} & 0.8*(_{2} & 0.64*(_{2}*1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64*1*)_{1} & 0.64*1*)_{2} & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8*)_{1} & 0.8*)_{2} & 0 & 0 \end{array}$$

A Motivating Example

Phase (c) High-order proximity embedding

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_1*1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2*1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1*)_1 & 1*)_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_1*1*)_1 & (_1*1*)_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2*1*)_1 & (_2*1*)_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$

Katz Index

$$\mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h = 0.8 \cdot \mathbf{A} + 0.8^2 \cdot \mathbf{A}^2 + 0.8^3 \cdot \mathbf{A}^3$$

Context-sensitive value-flow reachability matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0.512 & 0 & 0.8 & 0.64 \\ l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0 \end{array}$$

A Motivating Example

Phase (c) High-order proximity embedding

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & (_1 & 0 \\ l_2 & 0 & 0 & 0 & 0 & (_2 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_6 & 0 & 0 &)_1 &)_2 & 0 & 0 \end{array}$$

$\mathbf{A}$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0 & 0 & 0 & (_1 * 1 \\ l_2 & 0 & 0 & 0 & 0 & 0 & (_2 * 1 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 1 *)_1 & 1 *)_2 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & (_1 * 1 *)_1 & (_1 * 1 *)_2 & 0 & 0 \\ l_2 & 0 & 0 & (_2 * 1 *)_1 & (_2 * 1 *)_2 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$

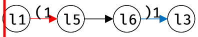
Katz Index

$$\mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h = 0.8 \cdot \mathbf{A} + 0.8^2 \cdot \mathbf{A}^2 + 0.8^3 \cdot \mathbf{A}^3$$

Context-sensitive value-flow reachability matrix

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & \boxed{0.512} & 0 & 0.8 & 0.64 \\ l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0 \end{array}$$

Example: l1 to l3



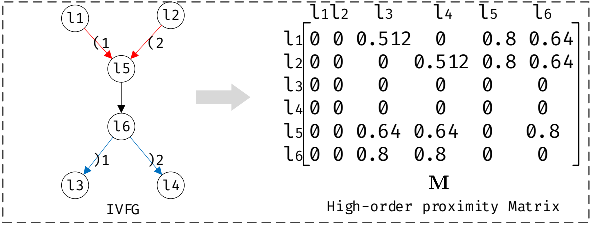
Reachability from l1 to l3:

$$\mathbf{M}_{1,3} = 0.8 \cdot \mathbf{A}_{1,3} + 0.8^2 \cdot \mathbf{A}^2_{1,3} + 0.8^3 \cdot \boxed{\mathbf{A}^3_{1,3}}$$

$$= 0.8 * 0 + 0.8^2 * 0 + 0.8^3 * \boxed{1} = 0.512$$

A Motivating Example

Phase (c) High-order proximity embedding



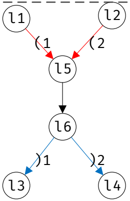
$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0.512 & 0 & 0.8 & 0.64 \\ l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0 \end{array}$$

$\mathbf{M}$

High-order proximity Matrix

A Motivating Example

Phase (c) High-order proximity embedding



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0.512 & 0 & 0.8 & 0.64 \\ l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0 \end{array}$$

$$\mathbf{M}$$

High-order proximity Matrix

Singular Value Decompose

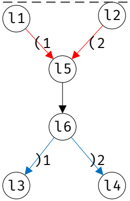
$$\mathbf{M} \approx \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{D}^s \cdot \mathbf{D}^{t\top}$$

$$\mathbf{D}^s = \left[\sqrt{\sigma_1} \cdot \mathbf{u}_1, \cdots, \sqrt{\sigma_K} \cdot \mathbf{u}_K\right] = \left[\mathbf{d}_1^s, \cdots, \mathbf{d}_N^s\right]$$

$$\mathbf{D}^t = \left[\sqrt{\sigma_1} \cdot \mathbf{v}_1, \cdots, \sqrt{\sigma_K} \cdot \mathbf{v}_K\right] = \left[\mathbf{d}_1^t, \cdots, \mathbf{d}_N^t\right]$$

A Motivating Example

Phase (c) High-order proximity embedding



IVFG

$$\begin{array}{c|cccccc} & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline l_1 & 0 & 0 & 0.512 & 0 & 0.8 & 0.64 \\ l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\ l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0 \end{array}$$

$\mathbf{M}$

High-order proximity Matrix

Singular Value Decompose

$$\mathbf{M} \approx \mathbf{U\Sigma V}^\top = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{D}^s \cdot \mathbf{D}^{t\top}$$

$$\mathbf{D}^s = \left[\sqrt{\sigma_1}\cdot\mathbf{u}_1, \cdots, \sqrt{\sigma_K}\cdot\mathbf{u}_K\right] = [\mathbf{d}_1^s, \cdots, \mathbf{d}_N^s]$$

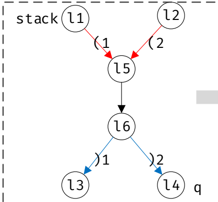
$$\mathbf{D}^t = \left[\sqrt{\sigma_1}\cdot\mathbf{v}_1, \cdots, \sqrt{\sigma_K}\cdot\mathbf{v}_K\right] = [\mathbf{d}_1^t, \cdots, \mathbf{d}_N^t]$$

$$\left[\begin{array}{ccc:ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right] \quad \left[\begin{array}{ccc:ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

$\mathbf{D}^s$ $\qquad$ $\mathbf{D}^t$

A Motivating Example

Phase (c) High-order proximity embedding



IVFG

$$
\begin{array}{c|cccccc}
 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\
\hline
l_1 & 0 & 0 & 0.512 & 0 & 0.8 & 0.64 \\
l_2 & 0 & 0 & 0 & 0.512 & 0.8 & 0.64 \\
l_3 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_5 & 0 & 0 & 0.64 & 0.64 & 0 & 0.8 \\
l_6 & 0 & 0 & 0.8 & 0.8 & 0 & 0
\end{array}
$$

$$\mathbf{M}$$

High-order proximity Matrix

Singular Value Decompose

$$\mathbf{M} \approx \mathbf{U\Sigma V}^\top = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{D}^s \cdot \mathbf{D}^{t\top}$$

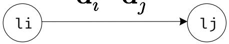
$$\mathbf{D}^s = \left[\sqrt{\sigma_1} \cdot \mathbf{u}_1, \cdots, \sqrt{\sigma_K} \cdot \mathbf{u}_K\right] = \left[\mathbf{d}_1^s, \cdots, \mathbf{d}_N^s\right]$$

$$\mathbf{D}^t = \left[\sqrt{\sigma_1} \cdot \mathbf{v}_1, \cdots, \sqrt{\sigma_K} \cdot \mathbf{v}_K\right] = \left[\mathbf{d}_1^t, \cdots, \mathbf{d}_N^t\right]$$

$$
\underbrace{\left[\begin{array}{ccc:ccc}
-0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\
0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.33 & -0.79 & 0 & 0 & 0 \\
0 & 0.71 & -0.58 & 0 & 0 & 0
\end{array}\right]}_{\mathbf{D}^s}
\quad
\underbrace{\left[\begin{array}{ccc:ccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
-0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\
0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\
0 & -0.71 & -0.58 & 0 & 0 & 0 \\
0 & -0.33 & -0.79 & 0 & 0 & 0
\end{array}\right]}_{\mathbf{D}^t}
$$

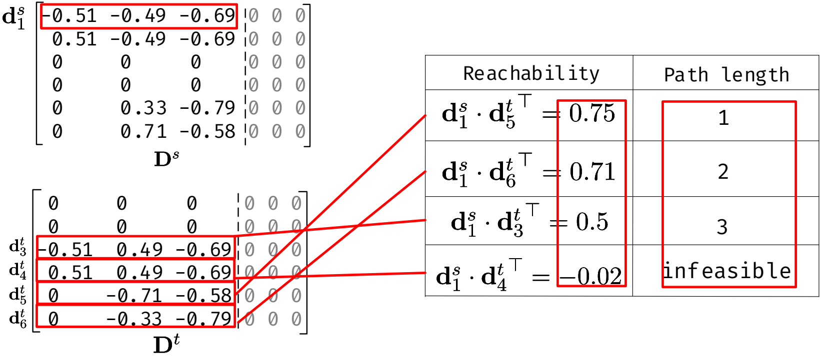
Reachability between
i and j

$\mathbf{d}_i^s \cdot \mathbf{d}_j^t$



A Motivating Example

Phase (c) High-order proximity embedding



$$\begin{array}{c}\mathbf{d}^s_1 \\ \\ \\ \\ \\ \\ \end{array}\left[\begin{array}{ccc:ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right]$$

$\mathbf{D}^s$

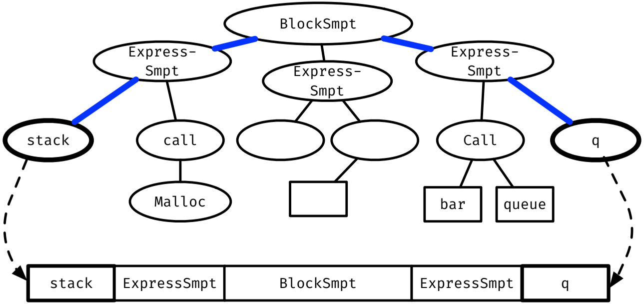
$$\begin{array}{c} \\ \\ \mathbf{d}^t_3 \\ \mathbf{d}^t_4 \\ \mathbf{d}^t_5 \\ \mathbf{d}^t_6 \end{array}\left[\begin{array}{ccc:ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

$\mathbf{D}^t$

Reachability	Path length
$\mathbf{d}^s_1 \cdot {\mathbf{d}^t_5}^\top = 0.75$	1
$\mathbf{d}^s_1 \cdot {\mathbf{d}^t_6}^\top = 0.71$	2
$\mathbf{d}^s_1 \cdot {\mathbf{d}^t_3}^\top = 0.5$	3
$\mathbf{d}^s_1 \cdot {\mathbf{d}^t_4}^\top = -0.02$	infeasible

A Motivating Example

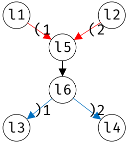
Phase (c) High-order proximity embedding



infeasible dependence relation between `stack` to `q`

A Motivating Example

Phase (d) Value-Flow Vector and Applications

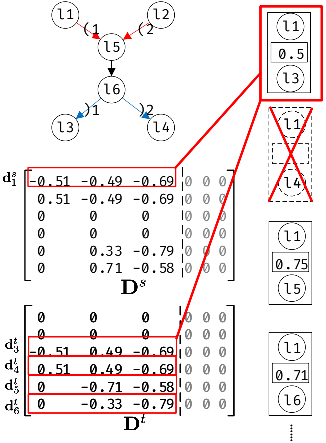


$$\underset{\mathbf{D}^s}{\left[\begin{array}{ccc:ccc}
-0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\
0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.33 & -0.79 & 0 & 0 & 0 \\
0 & 0.71 & -0.58 & 0 & 0 & 0
\end{array}\right]}$$

$$\underset{\mathbf{D}^t}{\left[\begin{array}{ccc:ccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
-0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\
0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\
0 & -0.71 & -0.58 & 0 & 0 & 0 \\
0 & -0.33 & -0.79 & 0 & 0 & 0
\end{array}\right]}$$

A Motivating Example

Phase (d) Value-flow Vectors and Applications



$$\begin{array}{c} \mathbf{d}_1^s \\ \\ \\ \\ \\ \\ \end{array} \left[\begin{array}{ccc|ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^s$$

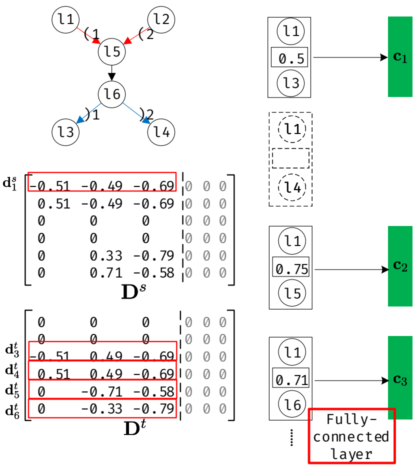
$$\begin{array}{c} \\ \\ \mathbf{d}_3^t \\ \mathbf{d}_4^t \\ \mathbf{d}_5^t \\ \mathbf{d}_6^t \end{array} \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^t$$

A Motivating Example

Phase (d) Value-flow Vectors and Applications

$$\mathbf{c}_n = \tanh(\mathbf{W} \cdot [l_i; \mathbf{d}^s_i \cdot \mathbf{d}^{t\top}_j; l_j])$$



$$\begin{array}{c} \mathbf{d}^s_1 \\ \\ \\ \\ \\ \\ \end{array} \left[\begin{array}{ccc|ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right]$$

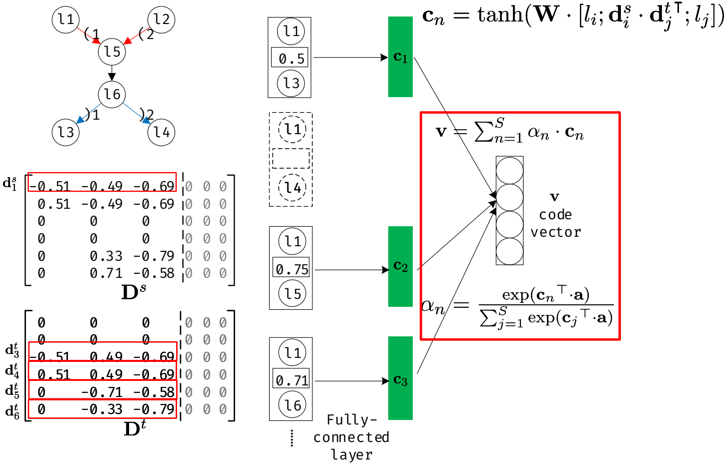
$$\mathbf{D}^s$$

$$\begin{array}{c} \\ \\ \mathbf{d}^t_3 \\ \mathbf{d}^t_4 \\ \mathbf{d}^t_5 \\ \mathbf{d}^t_6 \end{array} \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^t$$

A Motivating Example

Phase (d) Value-flow Vectors and Applications



$$\mathbf{c}_n = \tanh(\mathbf{W} \cdot [l_i; \mathbf{d}_i^s \cdot \mathbf{d}_j^{t\top}; l_j])$$

$$\begin{array}{c} \mathbf{d}_1^s \\ \\ \\ \\ \\ \\ \end{array} \left[\begin{array}{ccc|ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^s$$

$$\begin{array}{c} \\ \\ \mathbf{d}_3^t \\ \mathbf{d}_4^t \\ \mathbf{d}_5^t \\ \mathbf{d}_6^t \end{array} \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

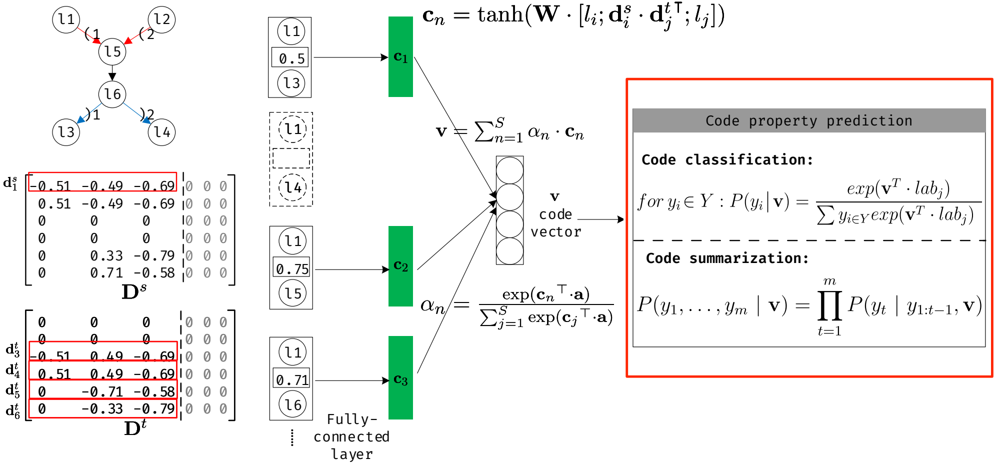
$$\mathbf{D}^t$$

$$\mathbf{v} = \sum_{n=1}^{S} \alpha_n \cdot \mathbf{c}_n$$

$$\alpha_n = \frac{\exp(\mathbf{c}_n{}^\top \cdot \mathbf{a})}{\sum_{j=1}^{S} \exp(\mathbf{c}_j{}^\top \cdot \mathbf{a})}$$

A Motivating Example

Phase (d) Value-flow Vectors and Applications



$$\begin{array}{c} \mathbf{d}_1^s \\ \\ \\ \\ \\ \\ \end{array} \left[\begin{array}{ccc:ccc} -0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & -0.49 & -0.69 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33 & -0.79 & 0 & 0 & 0 \\ 0 & 0.71 & -0.58 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^s$$

$$\begin{array}{c} \\ \\ \mathbf{d}_3^t \\ \mathbf{d}_4^t \\ \mathbf{d}_5^t \\ \mathbf{d}_6^t \end{array} \left[\begin{array}{ccc:ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0.51 & 0.49 & -0.69 & 0 & 0 & 0 \\ 0 & -0.71 & -0.58 & 0 & 0 & 0 \\ 0 & -0.33 & -0.79 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{D}^t$$

$$\mathbf{c}_n = \tanh(\mathbf{W} \cdot [l_i; \mathbf{d}_i^s \cdot \mathbf{d}_j^{t\top}; l_j])$$

$$\mathbf{v} = \sum_{n=1}^{S} \alpha_n \cdot \mathbf{c}_n$$

$$\alpha_n = \frac{\exp(\mathbf{c}_n{}^\top \cdot \mathbf{a})}{\sum_{j=1}^{S} \exp(\mathbf{c}_j{}^\top \cdot \mathbf{a})}$$

Code property prediction

Code classification:

$$for\ y_i \in Y : P(y_i | \mathbf{v}) = \frac{exp(\mathbf{v}^T \cdot lab_j)}{\sum y_{i\in Y} exp(\mathbf{v}^T \cdot lab_j)}$$

Code summarization:

$$P(y_1, \ldots, y_m \mid \mathbf{v}) = \prod_{t=1}^{m} P(y_t \mid y_{1:t-1}, \mathbf{v})$$

Experimental Evaluation

Benchmarks

32 popular open-source C/C++ projects

Total Line of Instructions:4,922,162

Total Methods:17,529

Total Pointers: 2,913,748

Total Objects: 190,157

Total Number of Calls:536,033

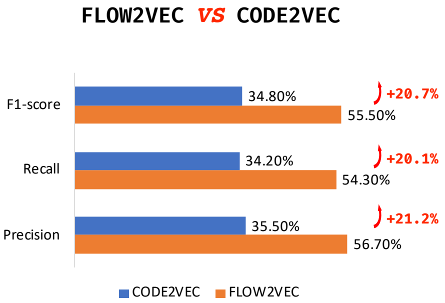
Total IVFGNodes: 4,637,301

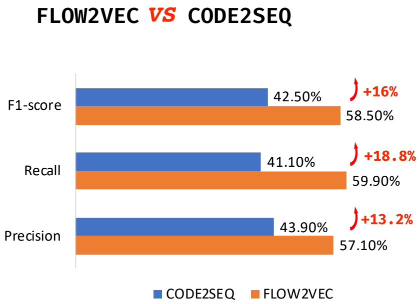
Total IVFGEdges: 6,531,578

*Conducted machine: Intel Xeon Gold 6132 @ 2.60GHz CPUs and 128GB of RAM (All finish analyzing in 272.5mins)

Experimental Evaluation

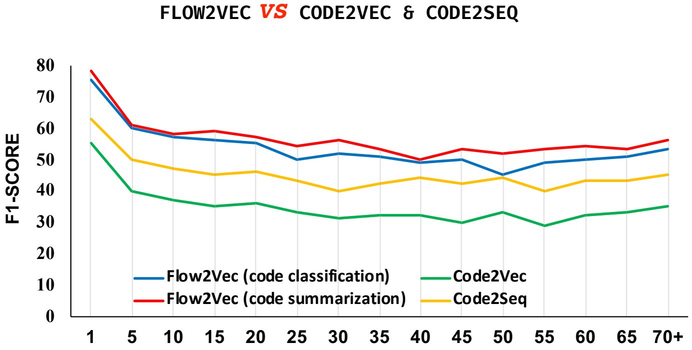
Comparison with baselines





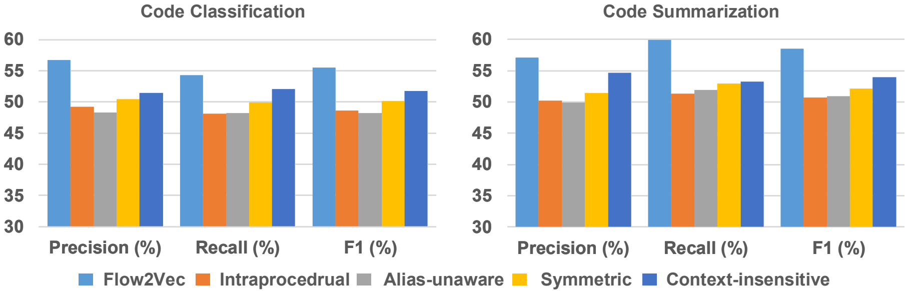
Experimental Evaluation

F1-score under different lengths of code

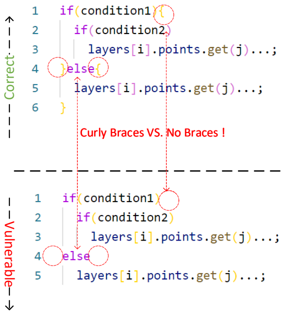


Experimental Evaluation

Ablation analysis



A Wide Variety of Vulnerabilities

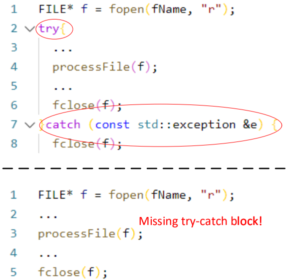


```
if(condition1){
  if(condition2)
    layers[i].points.get(j)...;
}else{
  layers[i].points.get(j)...;
}
```

Curly Braces VS. No Braces !

```
if(condition1)
  if(condition2)
    layers[i].points.get(j)...;
else
  layers[i].points.get(j)...;
```

(a) CWE-691 (the racoon daemon in IPsec-Tools 0.8.2)

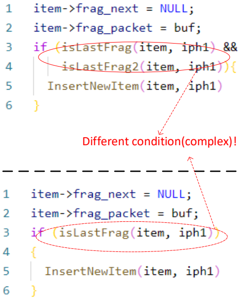


```
FILE* f = fopen(fName, "r");
try{
  ...
  processFile(f);
  ...
  fclose(f);
}catch (const std::exception &e) {
  fclose(f);
```

Missing try-catch block!

```
FILE* f = fopen(fName, "r");
...
processFile(f);
...
fclose(f);
```

(b) CWE-404 (https://cwe.mitre.org/)



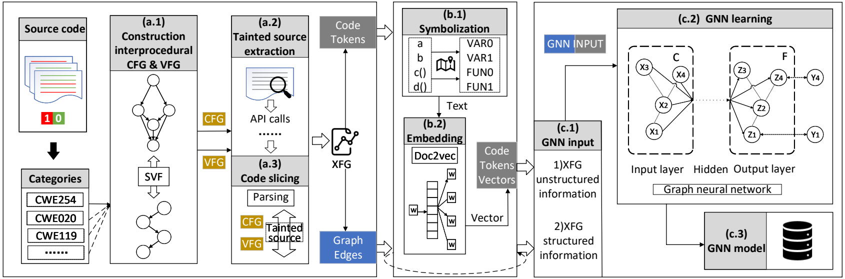
```
item->frag_next = NULL;
item->frag_packet = buf;
if (isLastFrag(item, iph1) &&
   isLastFrag2(item, iph1)){
  InsertNewItem(item, iph1)
}
```

Different condition(complex)!

```
item->frag_next = NULL;
item->frag_packet = buf;
if (isLastFrag(item, iph1))
{
  InsertNewItem(item, iph1)
}
```

(c) CVE-2016-10396 (the racoon daemon in IPsec-Tools 0.8.2)

Vulnerability Detection via Code Embedding (TOSEM ’21)



(a) Control and data slicing

(b) Code tokens symbolization and embedding

(c) Deep graph neural networks learning

Vulnerabilities from Software Assurance Reference Dataset (SARD)

(1) **CWE119: Improper Restriction of Operations within the Bounds of a Memory Buffer.** The program reads from or writes to a memory location that is outside of the intended boundary of the memory buffer.

(2) **CWE20: Improper Input Validation.** The program does not validate or incorrectly validates input that can affect the control-flow or data-flow of a program.

(3) **CWE125: Out-of-bounds Read.** The program reads data past the end, or before the beginning, of the intended buffer.

(4) **CWE190: Integer Overflow or Wraparound.** The program performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value.

(5) **CWE22: Improper Limitation of a Pathname to a Restricted Directory.** The program uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

(6) **CWE399: Resource Management Errors.** It is related to improper management of system resources.

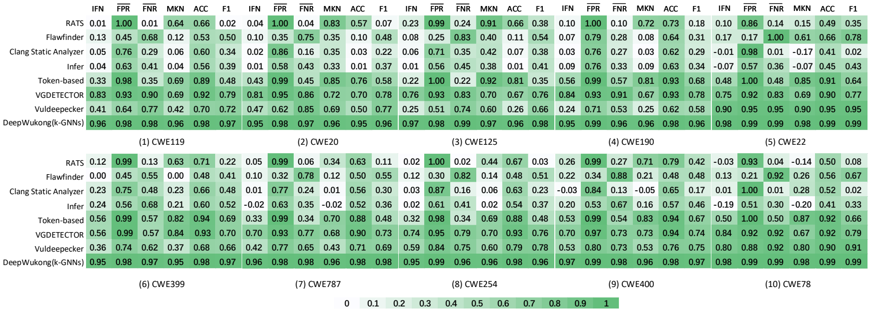
(7) **CWE787: Out-of-bounds Write.** The program writes data past the end, or before the beginning, of the intended buffer.

(8) **CWE254: Security Features.** It is related to security related operations, e.g., authentication, access control, confidentiality, cryptography, and privilege management, etc.

(9) **CWE400: Uncontrolled Resource Consumption.** The program does not properly control the allocation and maintenance of a limited resource thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.

(10) **CWE78: Improper Neutralization of Special Elements.** The vulnerable program constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.

Results



	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.01	1.00	0.01	0.64	0.66	0.02
Flawfinder	0.13	0.45	0.68	0.12	0.53	0.50
Clang Static Analyzer	0.05	0.76	0.29	0.06	0.60	0.34
Infer	0.04	0.63	0.41	0.04	0.56	0.39
Token-based	0.33	0.98	0.35	0.69	0.89	0.48
VGDETECTOR	0.83	0.93	0.90	0.69	0.92	0.79
Vuldeepecker	0.41	0.64	0.77	0.42	0.70	0.72
DeepWukong(k-GNNs)	0.96	0.98	0.98	0.96	0.98	0.97

(1) CWE119

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.04	1.00	0.04	0.83	0.57	0.07
Flawfinder	0.10	0.35	0.75	0.35	0.10	0.48
Clang Static Analyzer	0.02	0.86	0.16	0.35	0.03	0.22
Infer	0.01	0.58	0.43	0.33	0.01	0.37
Token-based	0.43	0.99	0.45	0.85	0.76	0.58
VGDETECTOR	0.81	0.95	0.86	0.72	0.70	0.78
Vuldeepecker	0.47	0.62	0.85	0.69	0.50	0.77
DeepWukong(k-GNNs)	0.95	0.98	0.97	0.96	0.95	0.96

(2) CWE20

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.23	0.99	0.24	0.91	0.66	0.38
Flawfinder	0.08	0.25	0.83	0.40	0.11	0.54
Clang Static Analyzer	0.06	0.71	0.35	0.42	0.07	0.38
Infer	0.01	0.56	0.45	0.38	0.01	0.41
Token-based	0.22	1.00	0.22	0.92	0.81	0.35
VGDETECTOR	0.76	0.93	0.83	0.70	0.67	0.76
Vuldeepecker	0.25	0.51	0.74	0.60	0.26	0.66
DeepWukong(k-GNNs)	0.97	0.98	0.99	0.97	0.96	0.98

(3) CWE125

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.10	1.00	0.10	0.72	0.73	0.18
Flawfinder	0.07	0.79	0.28	0.08	0.64	0.31
Clang Static Analyzer	0.03	0.76	0.27	0.03	0.62	0.29
Infer	0.09	0.76	0.33	0.09	0.63	0.34
Token-based	0.56	0.99	0.57	0.81	0.93	0.68
VGDETECTOR	0.84	0.93	0.91	0.67	0.93	0.78
Vuldeepecker	0.24	0.71	0.53	0.25	0.62	0.58
DeepWukong(k-GNNs)	0.95	0.99	0.96	0.96	0.98	0.96

(4) CWE190

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.10	0.86	0.14	0.15	0.49	0.35
Flawfinder	0.17	0.17	1.00	0.61	0.66	0.78
Clang Static Analyzer	-0.01	0.98	0.01	-0.17	0.41	0.02
Infer	-0.07	0.57	0.36	-0.07	0.45	0.43
Token-based	0.48	1.00	0.48	0.85	0.91	0.64
VGDETECTOR	0.75	0.92	0.83	0.69	0.90	0.77
Vuldeepecker	0.90	0.95	0.95	0.90	0.95	0.95
DeepWukong(k-GNNs)	0.98	0.99	0.99	0.98	0.99	0.99

(5) CWE22

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.12	0.99	0.13	0.63	0.71	0.22
Flawfinder	0.00	0.45	0.55	0.00	0.48	0.41
Clang Static Analyzer	0.23	0.75	0.48	0.23	0.66	0.48
Infer	0.24	0.56	0.68	0.21	0.60	0.52
Token-based	0.56	0.99	0.57	0.82	0.94	0.69
VGDETECTOR	0.56	0.99	0.57	0.84	0.93	0.70
Vuldeepecker	0.36	0.74	0.62	0.37	0.68	0.66
DeepWukong(k-GNNs)	0.95	0.98	0.97	0.95	0.98	0.97

(6) CWE399

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.05	0.99	0.06	0.34	0.63	0.11
Flawfinder	0.10	0.32	0.78	0.12	0.50	0.55
Clang Static Analyzer	0.01	0.77	0.24	0.01	0.56	0.30
Infer	-0.02	0.63	0.35	-0.02	0.52	0.36
Token-based	0.33	0.99	0.34	0.70	0.88	0.48
VGDETECTOR	0.70	0.93	0.77	0.68	0.90	0.73
Vuldeepecker	0.42	0.77	0.65	0.43	0.71	0.69
DeepWukong(k-GNNs)	0.96	0.98	0.98	0.96	0.98	0.98

(7) CWE787

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.02	1.00	0.02	0.44	0.67	0.03
Flawfinder	0.12	0.30	0.82	0.14	0.48	0.51
Clang Static Analyzer	0.03	0.87	0.16	0.06	0.63	0.23
Infer	0.02	0.61	0.41	0.02	0.54	0.37
Token-based	0.32	0.98	0.34	0.69	0.88	0.48
VGDETECTOR	0.74	0.95	0.79	0.70	0.93	0.76
Vuldeepecker	0.59	0.84	0.75	0.60	0.79	0.78
DeepWukong(k-GNNs)	0.95	0.99	0.96	0.96	0.98	0.96

(8) CWE254

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	0.26	0.99	0.27	0.71	0.79	0.42
Flawfinder	0.22	0.34	0.88	0.21	0.48	0.48
Clang Static Analyzer	-0.03	0.84	0.13	-0.05	0.65	0.17
Infer	0.20	0.53	0.67	0.16	0.57	0.46
Token-based	0.53	0.99	0.54	0.83	0.94	0.67
VGDETECTOR	0.70	0.97	0.73	0.73	0.94	0.74
Vuldeepecker	0.53	0.80	0.73	0.53	0.76	0.75
DeepWukong(k-GNNs)	0.97	0.99	0.98	0.96	0.98	0.97

(9) CWE400

	IFN	$\overline{FPR}$	$\overline{FNR}$	MKN	ACC	F1
RATS	-0.03	0.93	0.04	-0.14	0.50	0.08
Flawfinder	0.13	0.21	0.92	0.26	0.56	0.67
Clang Static Analyzer	0.01	1.00	0.01	0.28	0.52	0.02
Infer	-0.19	0.51	0.30	-0.20	0.41	0.33
Token-based	0.50	1.00	0.50	0.87	0.92	0.66
VGDETECTOR	0.84	0.92	0.92	0.67	0.92	0.79
Vuldeepecker	0.80	0.88	0.92	0.80	0.90	0.91
DeepWukong(k-GNNs)	0.98	0.99	0.99	0.98	0.99	0.99

(10) CWE78

0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

The darker the cell (the higher the value, the better the performance). Note that, for the FPR and FNR, we present their additive inverse here, which represents 1-FPR and 1-FNR separately. MKN denotes Informedness and Markedness. ACC denotes accuracy and F1 denotes the F1 measure score.

Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding, 24th International Conference on Engineering of Complex Computer Systems (ICECCS 2019)

Statically Detecting Software Vulnerabilities using Deep Graph Neural Network (TOSEM 2021)

Future Research Opportunities

- **A robust, comprehensive and learnable code representation**: Introducing path-sensitive analysis into code feature extraction.
- **Ultra-fast learning-based bug detection**: significantly boosting the performance of conventional program analysis (e.g., data-flow, abstraction interpretation and fuzz testing)
- **Automated and Intelligent vulnerability detection for more interesting clients**: Fault injection and localization for cyber physical systems (CPS)

Thanks!

Q & A