BinSide : Static Analysis Framework for Defects Detection in Binary Code

Hayk Aslanyan[1] (*hayk@ispras.ru*)
Mariam Arutunian[1], Grigor Keropyan[2],
Shamil Kurmangaleev[3], Vahagn Vardanyan[1]

[1] Russian-Armenian University, [2] Yerevan State University,

[3] Ivannikov Institute for System Programming of the Russian Academy of Sciences

25.09.2020

Binary code static analysis

- One of the approaches to the problem of detecting **defects** (potential errors) is **static analysis**: **examining** code without executing the program

- The work considers the **binary code** of programs - a set of commands executed directly by the processor

Binary code static analysis

- Software developers often make mistakes
- Source code analysis sometimes is not enough:
  - Using third-part libraries
  - Unprovability of correctness of all compiler optimizations
- Existing methods for analyzing executable code have limitations in scalability
- Available tools for finding defects in binary code have a low percentage of correct results (usually less than 10%)

Aim

Research and develop methods for binary code static analysis to find **defects**

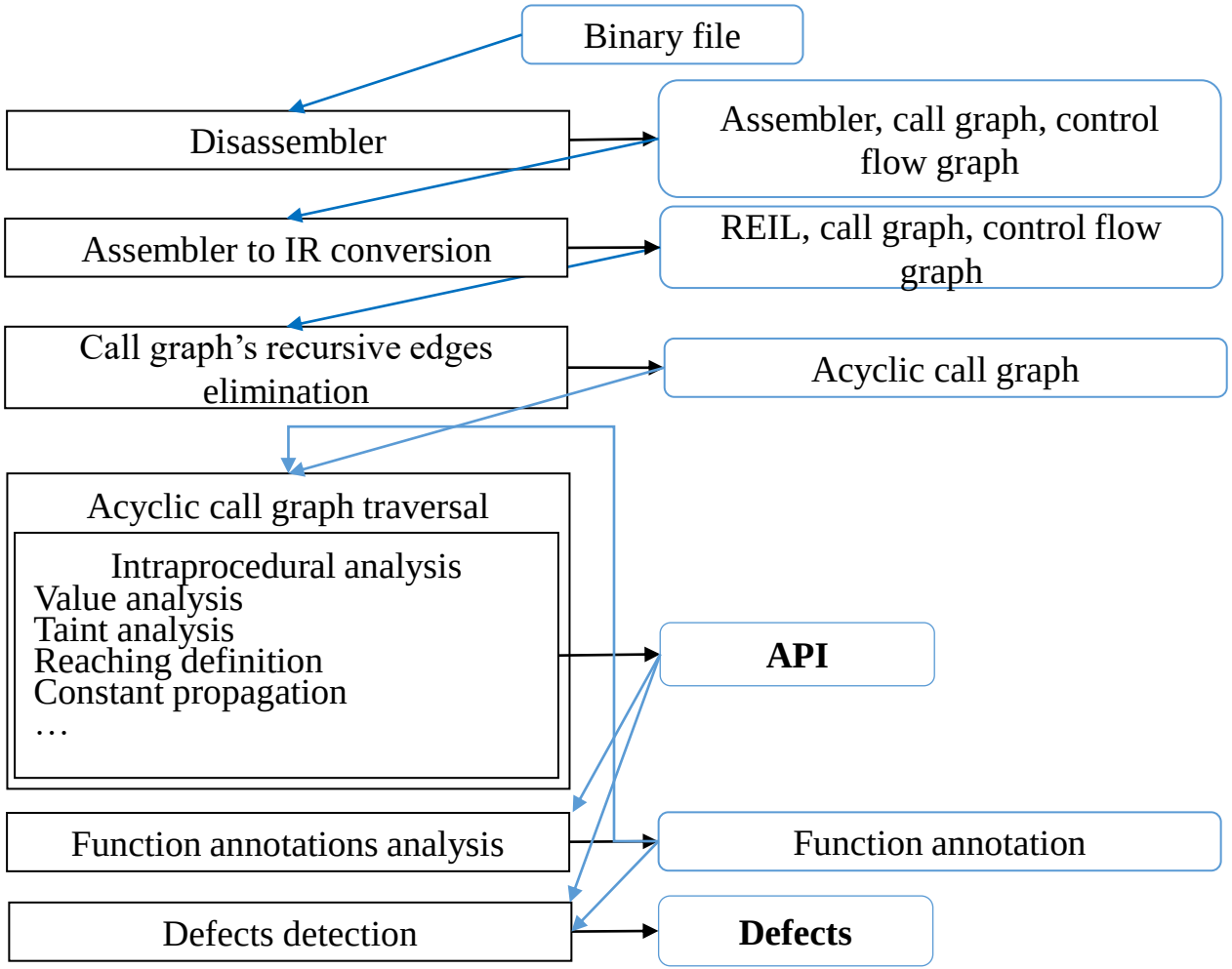
Problem definition

- We consider the following requirements to develop the framework:
  - Opportunity to analyze binaries of different architectures (x86, x86-64, ARM).
  - Interprocedural, context-sensitive and flow-sensitive analysis.
  - The framework should be extensible for writing new analysis and checkers.

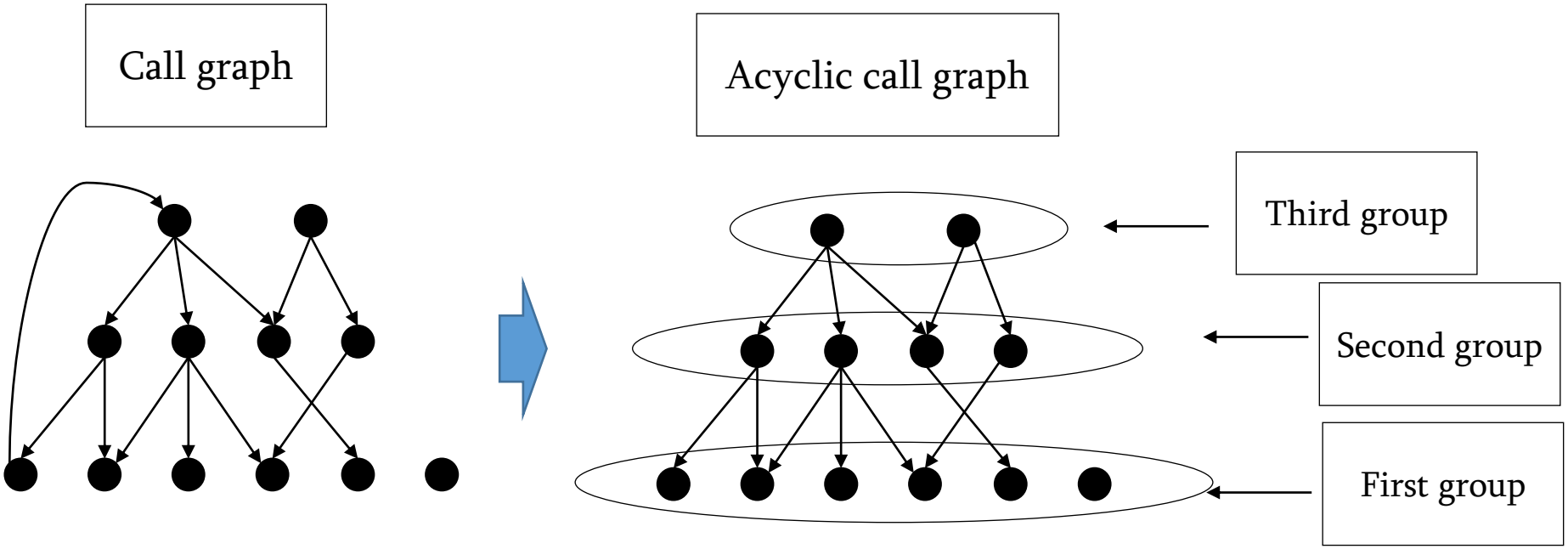
Problem definition

- Research and develop
  - Architecture of static analysis framework to find defects
    - Value analysis
    - Call Graph Edges Recovery
    - Taint analysis
    - Reaching definition analysis
    - Use-def and def-use chains construction
    - Constant propagation and constant folding
    - Freed memory analysis
    - Functions annotations
  - Defects detection
    - Buffer overflow
    - Format string
    - Command injection
    - Use after free
    - Double free

Framework architecture



Interprocedural analysis



Intraprocedural analysis

- Value analysis
  - Call Graph Edges Recovery
- Taint analysis
- Reaching definition analysis
  - Use-def and def-use chains construction
- Constant propagation and constant folding
- Freed memory analysis
- Functions annotations

Intraprocedural analysis - value analysis

- Tracks values in registers and memory cells (variables) in all function's points
  - Model memory in stack, heap, static region
- Variable values
  - *bottom, top, integer, temporary register, target architecture register, memory cell*
- Semilattice
  - *bottom* is the lowest element, *top* is the highest
- Memory model
  - *(reg + constants_array) + constant

Intraprocedural analysis - value analysis

- Memory model - **(reg + constants_array) + constant*
- Modeling memory in stack
  - Define *stack* as current function's frame start address and model all local variables relatively

If ***ebx*** has value ***stack***, then
- after ***mov rax,rbx+4*** instruction, ***rax*** value will be ***stack+4***
- after ***mov rcx,[rbx+8]*** instruction, ***rcx*** will be ****(stack+{8})***

Intraprocedural analysis - value analysis

- Memory model - **(reg + constants_array) + constant*
- Modeling memory in stack
  - Define *stack* as current function's frame start address and model all local variables relatively
- Modeling memory in heap
  - we use heap symbol and call instruction address, which corresponding function allocates memory in heap

After processing ***malloc*** call instruction with address ***0xFFFFFFFF***, ***rax*** value will be ****(heap+{0xFFFFFFFF})*** (in case of cdecl calling convention)

Intraprocedural analysis - value analysis

- Memory model - **(reg + constants_array) + constant*
- Modeling memory in stack
  - Define *stack* as current function's frame start address and model all local variables relatively
- Modeling memory in heap
  - we use heap symbol and call instruction address, which corresponding function allocates memory in heap
- Modeling static memory
  - constant part is used and its value is equals to the variable's memory address

Intraprocedural analysis

- Value analysis
  - Call Graph Edges Recovery
- Taint analysis
- Reaching definition
  - Use-def and def-use ch
- Constant propagation a
- Freed memory analysis
- Functions annotations

- Uses two attributes for variables – *taint* and *untaint*
- Uses *argument_untrusted_annotation*, *return_untrusted_annotation* and *copy_arguments_annotation*

Intraprocedural analysis

- Value analysis
  - Call Graph Edges Re
- Taint analysis
- Reaching definition
  - Use-def and def-use
- Constant propagatio
- Freed memory analysis
- Functions annotations

- Tracks all heap allocations and deallocation
- After analysis, all variables get *dangerous* or *not_dangerous* attributes and all memory cells get *freed* or *not_freed* attributes
- If a memory cell gets *freed* value its all pointers get *dangerous* value

Functions annotations

- *argument_free_annotation*
- *allocate_memory_annotation*
- *argument_dereference_annotation*
- *argument_format_string_annotation*
- *argument_buffer_overflow_annotation*
- *argument_command_injection_annotation*
- *argument_untrusted_annotation*
- *return_untrusted_annotation*
- *copy_arguments_annotation*
- *returns_argument_content_size_annotation*

Functions annotations

- ***argument_free_annotation***
- *allocate_memory_annotation*
- *argument_dereference_annotation*
- *argument_format_string_annotation*
- *argument_buffer_overflow_annotation*
- *argument_command_injection_annotation*
- *argument_untrusted_annotation*
- *return_untrusted_annotation*
- *copy_arguments_annotation*
- *returns_argument_content_size_annotation*

void free(void *ptr)
function from stdlib.h

Functions annotations

- ***argument_free_annotation***
- *allocate_memory_annotation*
- *argument_dereference_annotation*
- *argument_format_string_annotation*
- *argument_buffer_overflow_annotation*
- *argument_command_injection_annotation*
- *argument_untrusted_annotation*
- *return_untrusted_annotation*
- *copy_arguments_annotation*
- *returns_argument_content_size_annotation*

```
void my_free(void *ptr) {
    int* ptr2 = (int*) ptr;
    free(ptr2);
}
```

Functions annotations

- *argument_free_annotation*
- *allocate_memory_annotation*
- *argument_dereference_annotation*
- *argument_format_string_annotation*
- *argument_buffer_overflow_annotation*
- *argument_command_injection_annotation*
- *argument_untrusted_annotation*
- *return_untrusted_annotation*
- *copy_arguments_annotation*
- *returns_argument_content_size_annotation*

Detects detection

- Classic buffer overflow
- Format string
- Command injection
- Use after free
- Double free

Classic buffer overflow

- Argument of *main* function (user controlled) passes to *strcpy* function

```
                    _main:
push    rbp
mov     rbp, rsp
sub     rsp, 80
lea     rcx, [rbp - 64]
mov     rsi, qword ptr [rsi +8]
mov     rdi, rcx
call    _strcpy
…
```

- *rsi* contains *main* second argument
- It has *tainted* attribute

Classic buffer overflow

- Argument of *main* function (user controlled) passes to *strcpy* function

```
                    _main:
push    rbp
mov     rbp, rsp
sub     rsp, 80
lea     rcx, [rbp
mov     rsi, qword ptr [rsi +8]
mov     rdi, rcx
call    _strcpy
…
```

- *rsi* gets *tainted* attribute
- *rsi* passes to *strcpy* function as secon argument without checking buffers sizes

Classic buffer overflow

- Argument of *main* function (user controlled) passes to *strcpy* function

```
                    _main:
push    rbp
mov     rbp, rsp
sub     rsp, 80
lea     rcx, [rbp - 64]
mov     rsi, qword ptr [rsi +8]
mov     rdi, rcx
call    _strcpy
…
```

Buffer overflow

Format string

- User controlled data is passed to *printf* function

```
myprintf:
…..
call      _printf
….
                              taint_func:
……
mov       rdi, qword ptr [rbp - 8]
call      _gets
mov       rdi, qword ptr [rbp - 8]
mov       qword ptr [rbp - 16], rdi
mov       rdi, qword ptr [rbp - 8]
…..
call      myprintf
…
```

Format string

- User controlled data is passed to *printf* function

```
myprintf:
…..
call      _printf
….
                              taint_func:
……
mov       rdi, qword ptr [rbp - 8]
call      _gets
mov       rdi, qword ptr [rbp - 8]
mov       qword ptr [rbp - 16], rdi
mov       rdi, qword ptr [rbp - 8]
…..
call      myprintf
…
```

Gets
argument_format_string_annotation

Format string

- User controlled data is passed to *printf* function

```
myprintf:
…..
call      _printf
….
                          taint_fu
……
mov       rdi, qword ptr [rbp - 8]
call      _gets
mov       rdi, qword ptr [rbp - 8]
mov       qword ptr [rbp - 16], rdi
mov       rdi, qword ptr [rbp - 8]
…..
call      myprintf
…
```

[rbp - 8] gets *tainted* as it passed to *gets* function

Format string

- User controlled data is passed to *printf* function

```
myprintf:
…..
call      _printf
….
                              taint_func:
……
mov       rdi, qword ptr [rbp - 8]
call      _gets
mov       rdi, qword ptr [rbp - 8]
mov       qword ptr [rbp - 16], rdi
mov       rdi, qword ptr [rbp - 8]
…..
call      myprintf
…
```

[rbp - 8] passed to function with *argument_format_string_annotation*

Results on Juliet test suit

- Juliet test suit provides tests with defects and tests where defects are sanitized
- In average, the tool detects **buffer overflow, format string, command injection** defects with **99%** precision and **85%** recall
- In average, the tool detects double free and use after free defects with **40%** precision and **95%** recall

Results

Project, version (CVE)	Arch.	Detected defects count	Analysis time (sec.)	Precision
serenity.exe 3.2.3 (CVE-2009-4097)	x86	2	20	100%
FoxPlayer.exe 1.7.0	x86	1	142	100%
accel-ppd 1.10.0	x86	6	100	67%
gifcolor 5.1.2 (CVE-2016-3177)	x86	4	48	100%
gnome-nettool 3.8.1	x86-64	4	48	75%
libssh.so 4.2.2 (CVE-2012-4559)	x86	20	92	70%
slpd_no_frame 1.2.1 (CVE-2015-5177)	x86	4	50	75%

* Tests are processed in intel core i5-2500 with 16 GB RAM

BinSide vs GUEB*

Project	***GUEB*: DF and UAF count**	***GUEB*: precision**	***BinSide*: DF and UAF count**	***BinSide*: precision**
gnome-nettool 3.8.1	4	**25%**	4	**75%**
gifcolor 5.1.2	15	**6%**	4	**100%**
jasper 1.900.1	255	**1.2%**	4	**50%**
accel-pppd 1.10.0	35	**11.4%**	6	**67%**

*GUEB: tool to detect UAF and DF defects. Developed at **Verimag**

BinSide vs LoongChecker

Binary	***LoongChecker*: FS, BOF, CI count**	***LoongChecker*: precition**	***BinSide*: FS, BOF, CI count**	***BinSide*: precition**
Serenity.exe	8	**12.5%**	2	**100%**
FoxPlayer.exe	27	**4%**	1	**100%**

* LoongChecker : tool to detect BOF, FS and CI defects. Developed at University of Science and Technology of China

Thank you