

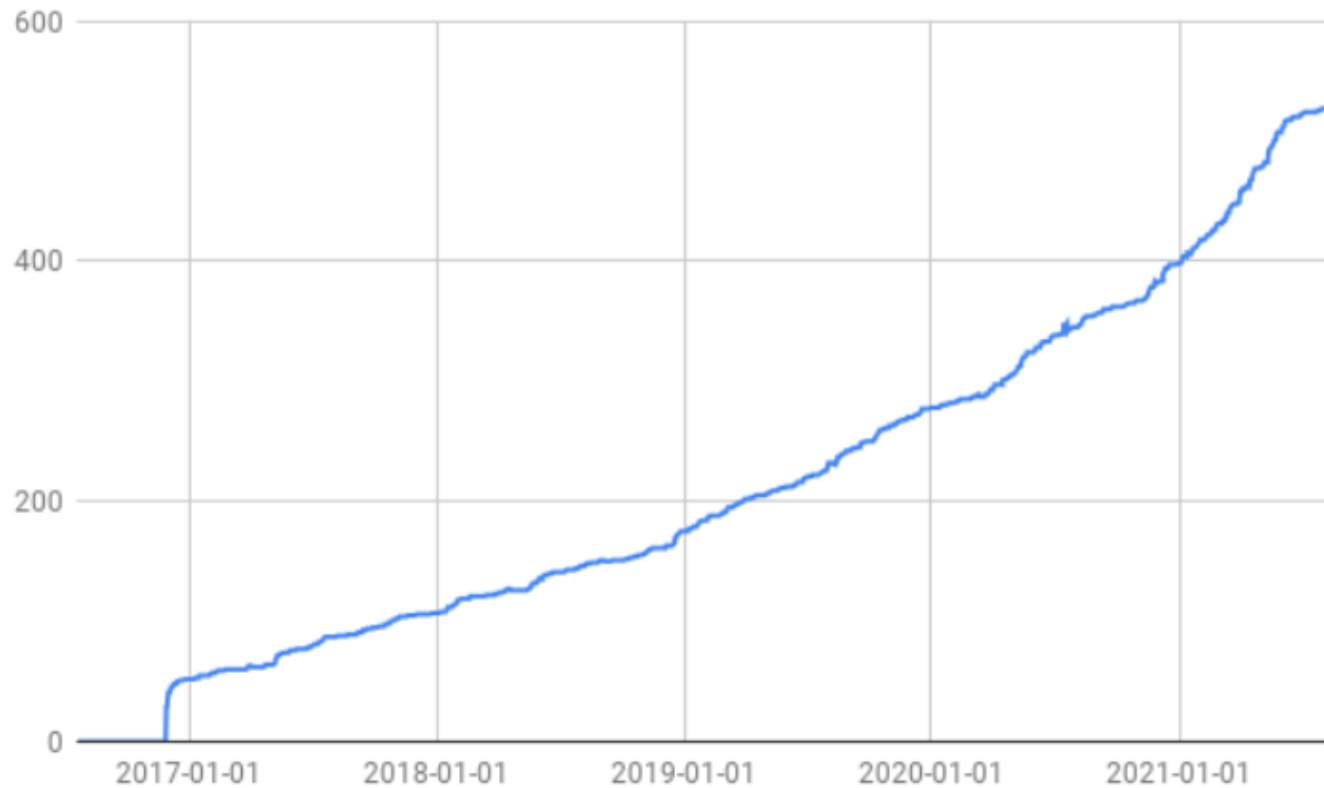
# Taint Checker: Tracking the taint propagation on Binary Programs

Sangjun Park



# Finding Software bugs is Important

- software bugs increase every years



# Bugs type

- memory corruption  
ex) Bof / UAF / Double free bugs
- non-memory corruption  
ex) Logical bugs, Command injection

How to detect the bugs  
non-memory corruption?

# Existing Methods

- Symbolic Execution

QSym, arbirter

- Taint analysis

SaTC, Taint pipe, code sonar, Condysta

- Concolic Execution

Symsan

# Existing Methods

- Symbolic Execution  
QSym, arbirter

- Taint analysis  
SaTC, Taint pipe, code sonar, Condysta

- Concolic Execution  
Symsan

# Taint analysis

- Track information flow from input



# Taint analysis

- Source : origin of data
- Sink : dangerous function / exploitable targets

```
void ping(char *target)
{
    char command[256];
    sprintf(command, "ping %s", target);
    system(command);
}
```



# Taint analysis

- Source : origin of data
- Sink : dangerous function / exploitable targets

```
void ping(char *target) ← source
{
    char command[256];
    sprintf(command, "ping %s", target);
    system(command); ← sink
}
```

# Taint analysis

- Taint Propagation
  - Mark source as taint
  - Pass the taint if value is copied to another variable/buffer
  - check if argument to sink holds the taint



# Taint analysis

- Taint Propagation
  - Mark source as taint
  - Pass the taint if value is copied to another variable/buffer
  - check if argument to sink holds the taint

```
C test.c > ping(char *)  
1 void ping(char *target)  
2 {  
3     char command[256];  
4     sprintf(command, "ping %s", target);  
5     system(command);  
6 }
```



# Taint analysis

- Taint Propagation
  - Mark source as taint
  - Pass the taint if value is copied to another variable/buffer
  - check if argument to sink holds the taint

```
C test.c > ping(char *)  
1 void ping(char *target)  
2 {  
3     char command[256];  
4     sprintf(command, "ping %s", target);  
5     system(command);  
6 }
```



# Taint analysis

- Taint Propagation
  - Mark source as taint
  - Pass the taint if value is copied to another variable/buffer
  - check if argument to sink holds the taint

```
C test.c > ping(char *)  
1 void ping(char *target)  
2 {  
3     char command[];  
4     sprintf(command, "ping %s", target);  
5     system(command);  
6 }
```



# Taint analysis

- Taint Propagation
  - Mark source as taint
  - Pass the taint if value is copied to another variable/buffer
  - check if argument to sink holds the taint

```
C test.c > ping(char *)
1 void ping(char *target)
2 {
3     char command[];
4     sprintf(command, "ping %s", target);
5     system(command);
6 }
```



# Taint analysis

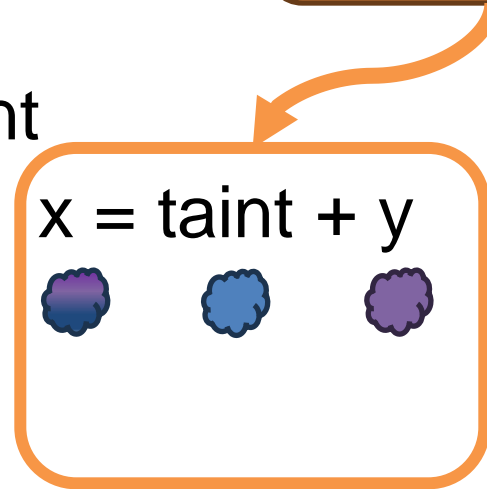


- program argument
- keyboard
- network
- files

# Taint analysis



- program argument
- keyboard
- network
- files



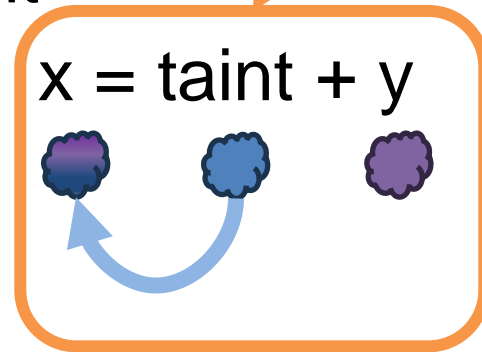
explicit  
data flows



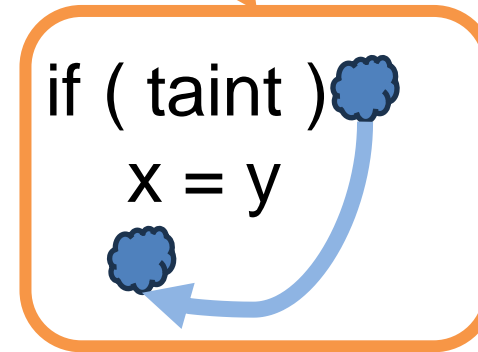
# Taint analysis



- program argument
- keyboard
- network
- files

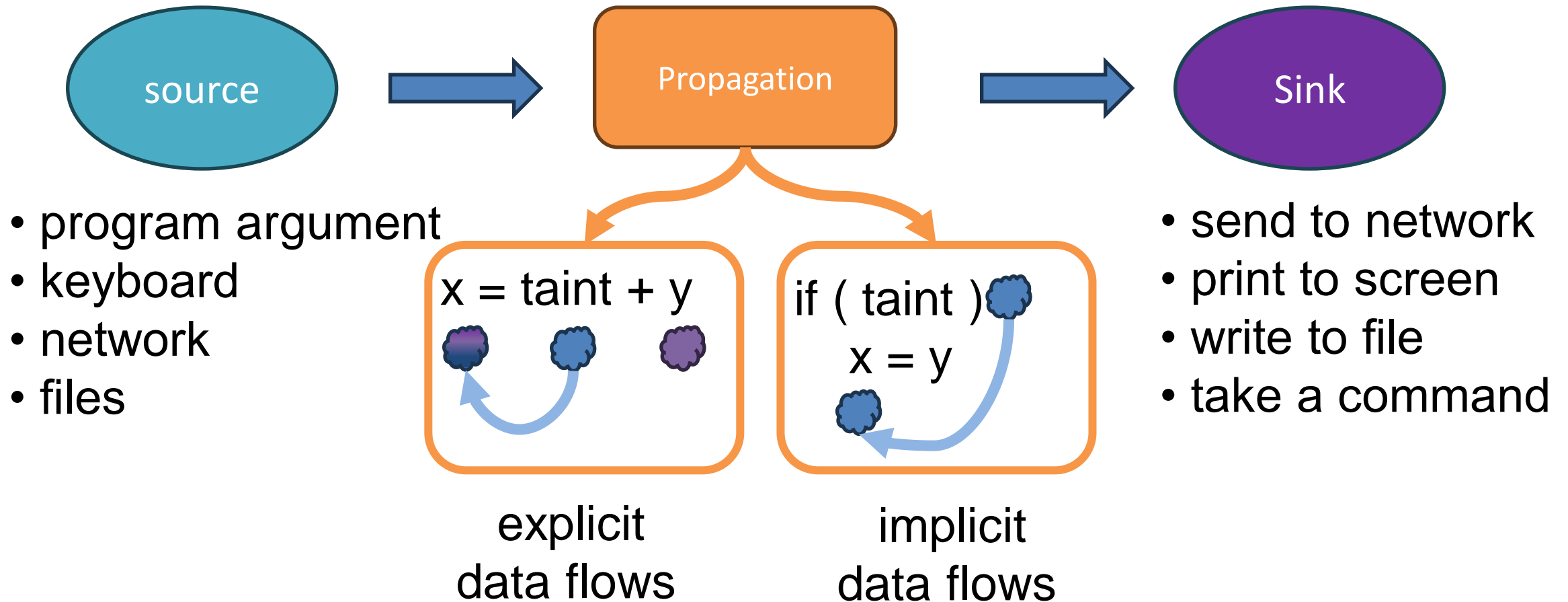


explicit  
data flows



implicit  
data flows

# Taint analysis



# How to implement taint propagation?

# Propagation Implementation

- Need to make a rule for each function or all data transmitted.  
ex) explicit data flow / implicit data flow

# Propagation Implementation

- Need to make a rule for each function or all data transmitted.  
ex) explicit data flow / implicit data flow
- Use angr for binary analysis.
  - Can simulate the program.
  - Support ARM / MIPS.
  - Symbolic execution support.
  - Actively maintained.
  - Run program to track taints.



# Propagation Implementation

- angr simulate program
- When we reach a copying function like sprintf
  - ➔ Pass the taint from the source string to the destination string
- When we reach the sink function
  - ➔ Check if the sink argument is tainted

```
void ping(char *target)
{
    char command[256];
    sprintf(command, "ping %s", target);
    system(command);
}
```

```
target: BitVec(1024*8)
command : BitVec(256*8)
```

```
target <- taint
command <- taint
```

when we reach the sink function, check if command is tainted.

# Challenge 1

- Need to make a rule for each function or all data transmitted.

## Best case

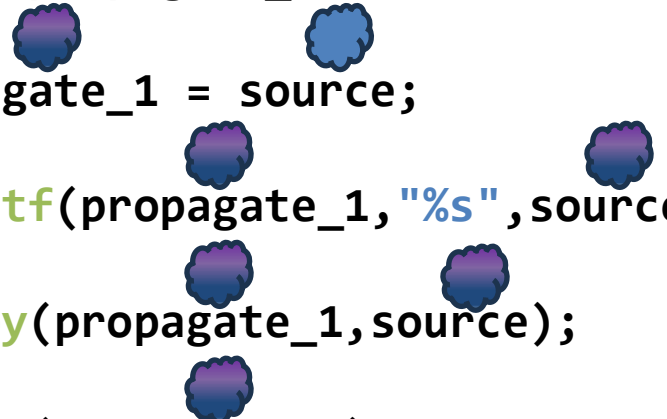
```
char *source;  
char *propagate_1;  
  
propagate_1 = source;  
  
sprintf(propagate_1, "%s", source);  
  
strcpy(propagate_1, source);  
  
system(propagate_1)
```

# Challenge 1

- Need to make a rule for each function or all data transmitted.

## Best case

```
char *source;  
char *propagate_1;  
propagate_1 = source;  
sprintf(propagate_1, "%s", source);  
strcpy(propagate_1, source);  
system(propagate_1)
```

A diagram illustrating data flow. It consists of several blue, cloud-like shapes connected by lines. The shapes are arranged in a way that shows the flow of data from the variable 'source' to 'propagate\_1' through various functions. The flow starts with 'source' at the top, then goes to 'propagate\_1', then through 'sprintf', 'strcpy', and finally 'system'. There are also direct connections from 'source' to 'sprintf' and 'strcpy', and from 'propagate\_1' to 'system'.



# Challenge 1

- Need to make a rule for each function or all data transmitted.

## Best case scenario

```
char *source;  
char *propagate_1;  
  
propagate_1 = source;  
  
sprintf(propagate_1, "%s", source);  
  
strcpy(propagate_1, source);  
  
system(propagate_1)
```

## Worst case scenario

```
char *source;  
char *propagate_1;  
  
user_defined_function(propagate_1, source);  
  
system(propagate_1)
```

# Challenge 1

- Need to make a rule for each function or all data transmitted.

## Best case scenario

```
char *source;  
char *propagate_1;  
  
propagate_1 = source;  
  
sprintf(propagate_1, "%s", source);
```

## Worst case scenario

```
char *source;  
char *propagate_1;  
  
user_defined_function(propagate_1, source);
```

If you set the propagation rule incorrectly for user\_defined\_function, you will not be able to find the bug.


# Challenge 2

- Need to make a rule for each function or all data transmitted.

## Best case scenario

```
char *source;  
char *propagate_1;  
  
propagate_1 = source;  
  
sprintf(propagate_1,"%s",source);  
  
strcpy(propagate_1,source);  
  
system(propagate_1)
```

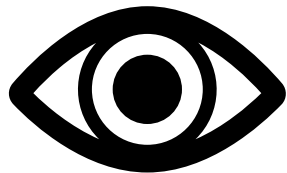
## Worst case scenario2

```
char *x = "/bin/sh";  
char *y = "/bin/ls";  
int source;  
  
  
if( source )  
    x = y  
system(x);
```

How to find the limitation of  
propagation rule?

# Observation

- we can evaluate the propagation rules through visualizing.
- Can find limitation of current propagation rules.

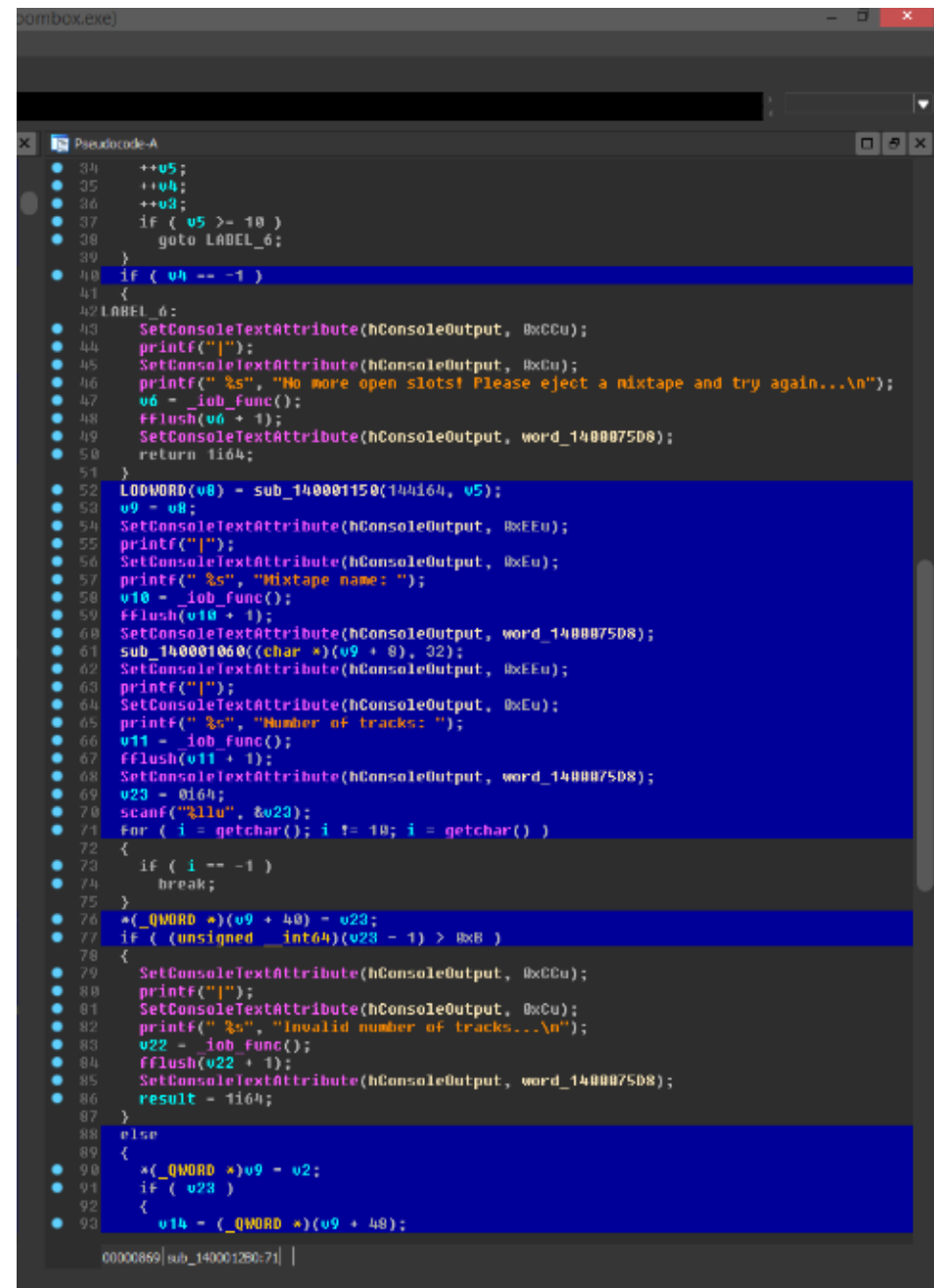


# Observation

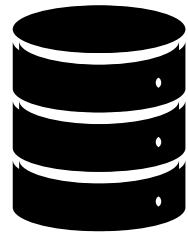
- key idea
  - Matches the process in which taint propagation occurs and the decompiler source code.



- Coverage Viewer
- IDA Light House



# Observation





# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Observation

- Remark where is source and sink
- Remark propagation

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *source; // [rsp+0h] [rbp-120h]
    char propagation[264]; // [rsp+10h] [rbp-110h] BYREF
    unsigned __int64 v6; // [rsp+118h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    source = (const char *)input(argc, argv, envp);
    if ( source )
        printf("AAAAAAAAA");
    else
        printf("BBBBBBBBB");
    sprintf(propagation, "%s", source);
    if ( !malloc(0x100uLL) )
        exit(1);
    system(propagation);
    return 0;
}
```

# Expectation

- Find the limitation of taint propagation implement
- Find ways to improve to create better taint analysis tools.

# Thank You

Questions?

Email: [sangjuns@kaist.ac.kr](mailto:sangjuns@kaist.ac.kr)







# Propagation Implementation

- Accuracy related to distance between source and sink

## Best case scenario



```
char *source;  
source = input();  
system(source)
```

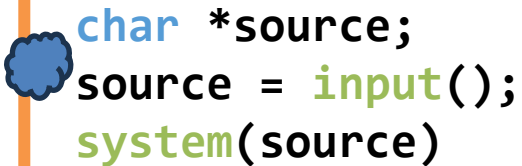


The code snippet is enclosed in an orange rounded rectangle. A blue cloud-like icon is positioned to the left of the first line of code, and another is positioned below the third line of code.

# Propagation Implementation

- Accuracy related to distance between source and sink

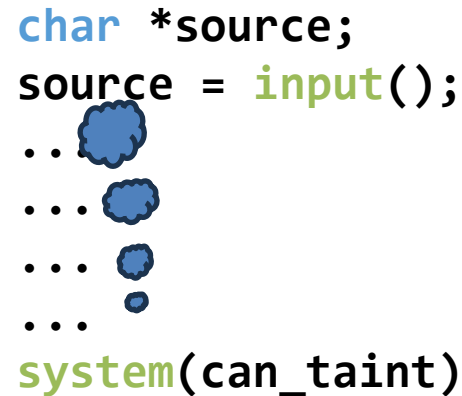
## Best case scenario



A diagram illustrating the best case scenario for propagation. It shows a single path from a source to a sink. The source is represented by a blue cloud icon on the left, and the sink is represented by a blue cloud icon on the right. A straight orange line connects them, passing through the code block.

```
char *source;  
source = input();  
system(source)
```

## Worst case scenario



A diagram illustrating the worst case scenario for propagation. It shows a source on the left and a sink on the right, connected by a long, winding orange line. The line is composed of many small segments, each representing a different code path. The code block is shown with the first line, followed by several lines of ellipses, and then the final line. A red double-headed arrow points from the text 'Too many code to simulate' to the ellipses.

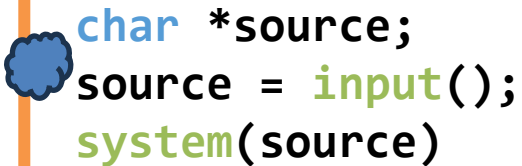
```
char *source;  
source = input();  
..  
..  
..  
..  
..  
system(can_taint)
```

Too many code  
to simulate

# Propagation Implementation

- Accuracy related to distance between source and sink

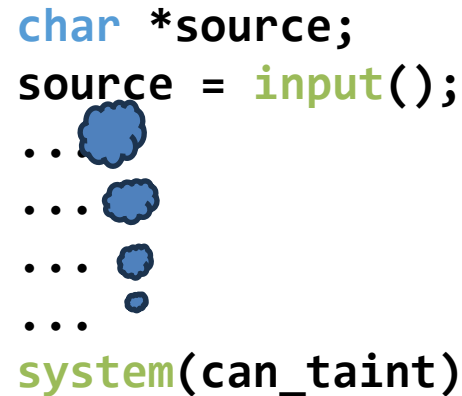
## Best case scenario



```
char *source;  
source = input();  
system(source)
```

The diagram shows a single blue cloud icon at the start of the first line and another at the end of the third line, representing a direct flow from source to sink.

## Worst case scenario



```
char *source;  
source = input();  
...  
...  
...  
...  
system(can_taint)
```

The diagram shows a series of blue cloud icons of decreasing size, representing a long, indirect path from the source to the sink.

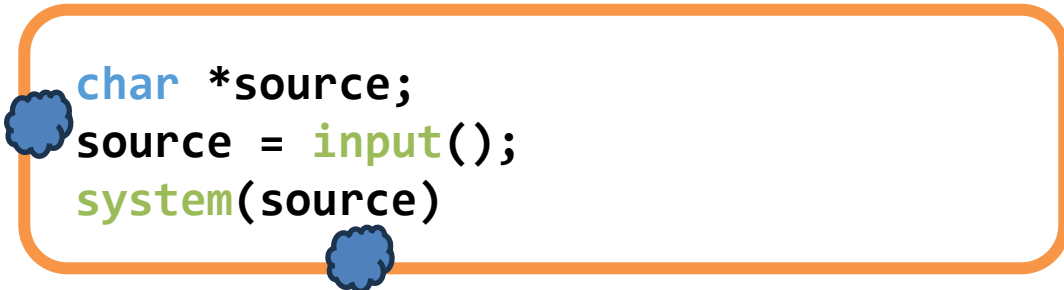
Accuracy is low because the data flow is not tainted perfectly.

# Propagation Implementation

- Accuracy related to distance between source and sink

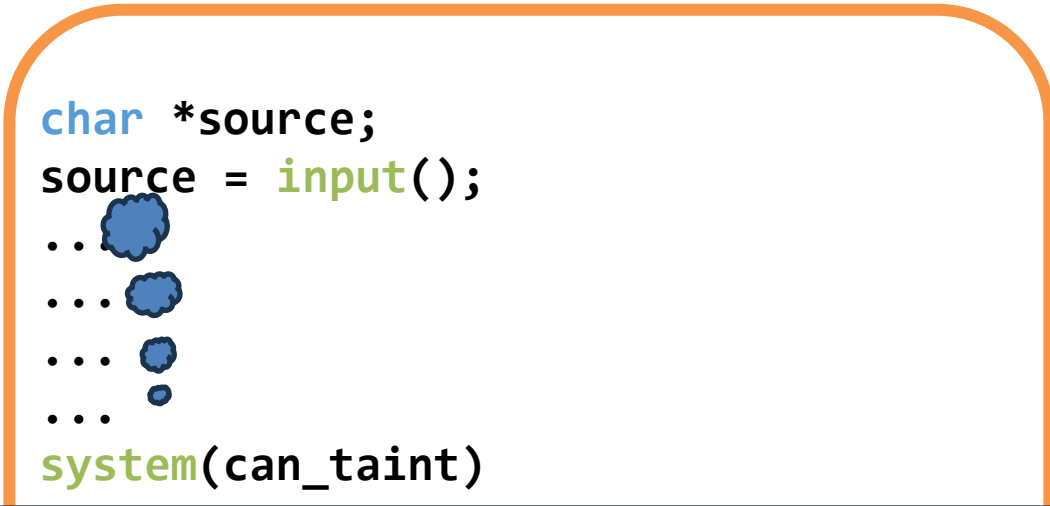
## Best case scenario

```
char *source;  
source = input();  
system(source)
```

A diagram illustrating the best case scenario for propagation. It shows a code block with three lines: 'char \*source;', 'source = input();', and 'system(source)'. A blue cloud icon is placed to the left of the first line, and another blue cloud icon is placed below the 'system(source)' line. The entire code block is enclosed in an orange rounded rectangle.

## Worst case scenario

```
char *source;  
source = input();  
...  
...  
...  
...  
system(can_taint)
```

A diagram illustrating the worst case scenario for propagation. It shows a code block with five lines: 'char \*source;', 'source = input();', three lines of '...', and 'system(can\_taint)'. Four blue cloud icons of decreasing size are placed to the left of the first four lines. The entire code block is enclosed in an orange rounded rectangle.

We cannot detect vulnerability