CodeFlowVis

Visualizer of Code Coverage and Execution Flow

Jaehoon Jang

IS661

2024.04.04

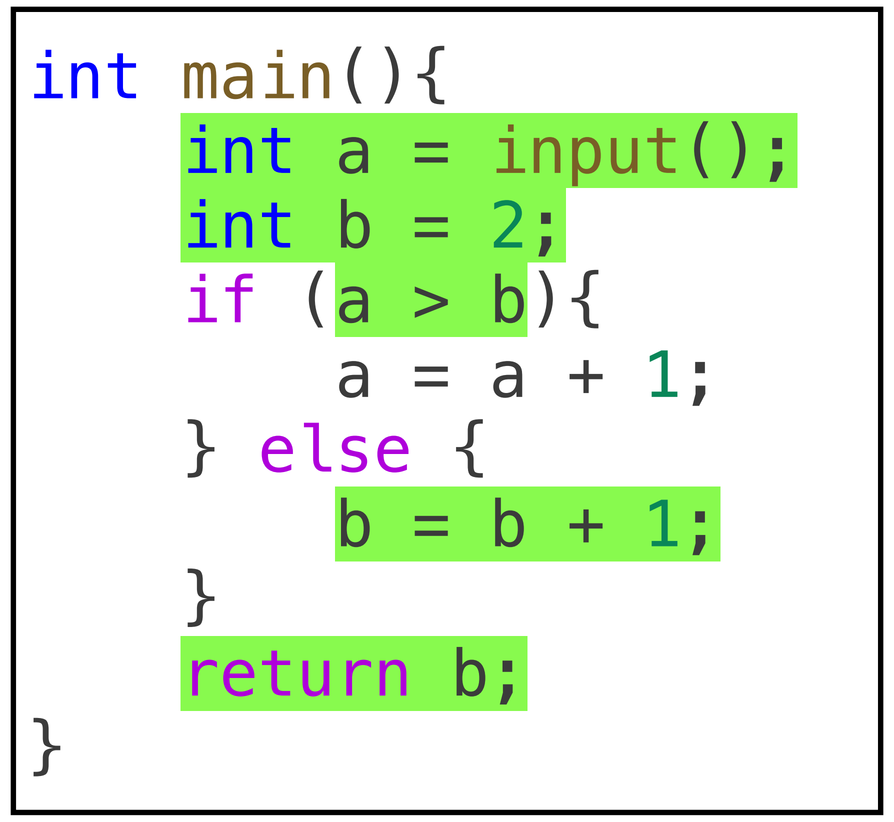
Background

- **Code Coverage**
  - Executed source code during the testing
  - Useful in identifying errors or vulnerabilities

```
int main(){
    int a = input();
    int b = 2;
    if (a > b){
        a = a + 1;
    } else {
        b = b + 1;
    }
    return b;
}
```

Background

- **Code Coverage**
  - Executed source code during the testing
  - Useful in identifying errors or vulnerabilities

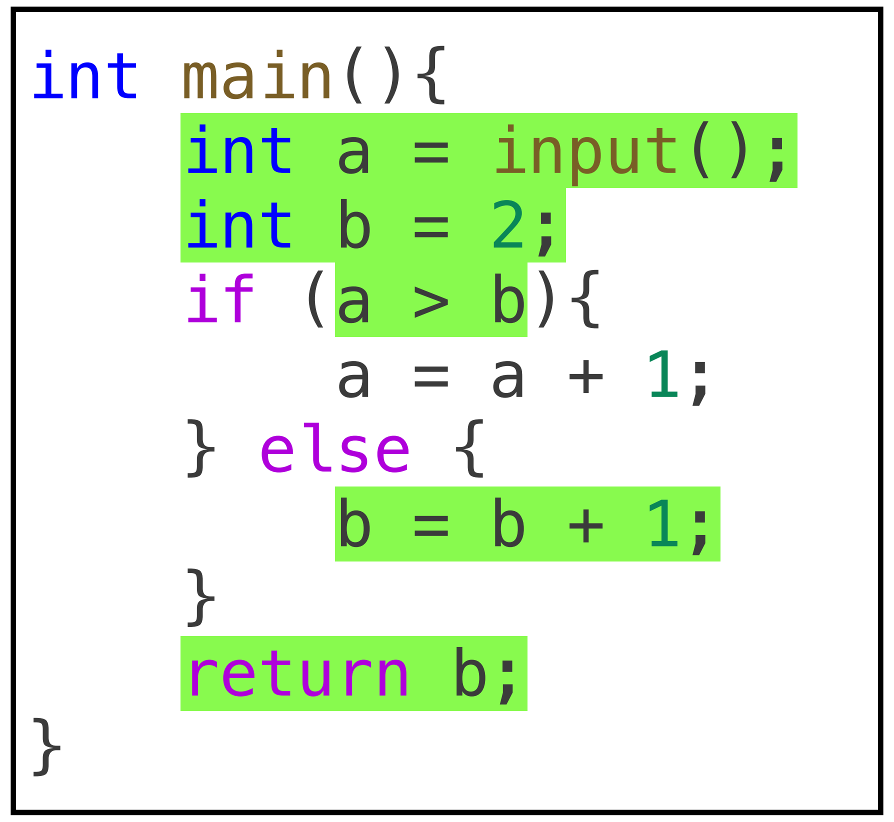


```
int main(){
    int a = input();
    int b = 2;
    if (a > b){
        a = a + 1;
    } else {
        b = b + 1;
    }
    return b;
}
```

a = 1

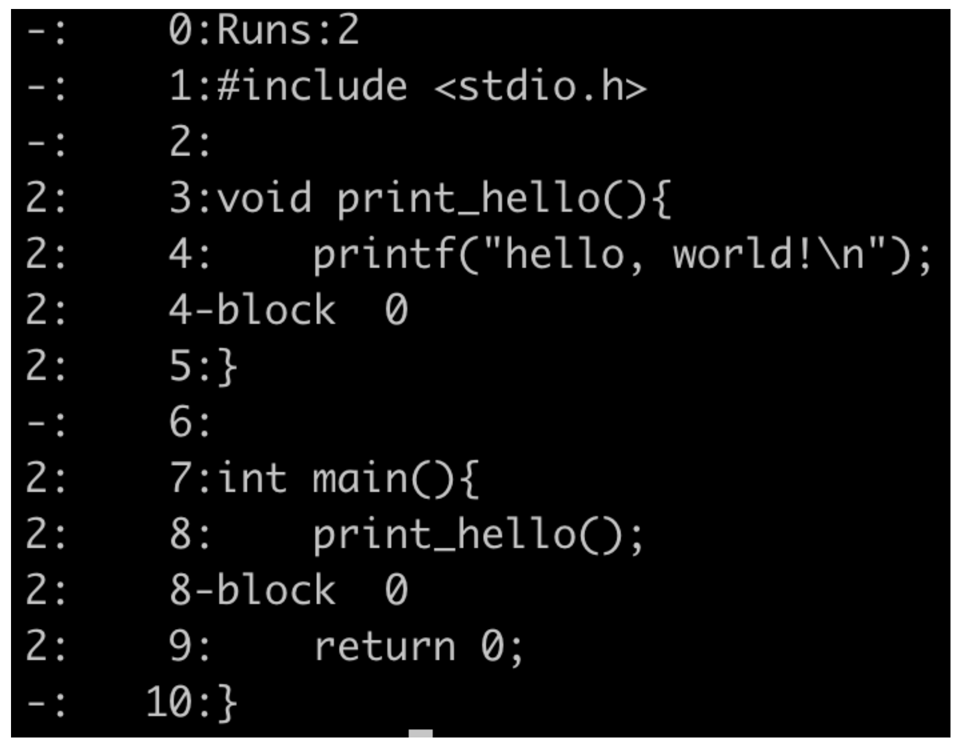
Background

- **Code Coverage**
  - Executed source code during the testing
  - Useful in identifying errors or vulnerabilities
- **Code Coverage Tools**
  - JaCoCo[1], GCOV[2]
  - Measure & Visualize code coverage



```
int main(){
    int a = input();
    int b = 2;
    if (a > b){
        a = a + 1;
    } else {
        b = b + 1;
    }
    return b;
}
```

a = 1



```
-:    0:Runs:2
-:    1:#include <stdio.h>
-:    2:
2:    3:void print_hello(){
2:    4:    printf("hello, world!\n");
2:    4-block  0
2:    5:}
-:    6:
2:    7:int main(){
2:    8:    print_hello();
2:    8-block  0
2:    9:    return 0;
-:   10:}
```

[1] JaCoCo, https://www.jacoco.org/jacoco/
[2] GCOV, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Problem

- Difficult to identify the **execution flow**
- It is easy when it is a single function & file.

```
int main(){
    int a = input();
    int b = 2;
    if (a > b){
        f();
    } else {
        g();
    }
    return b;
}
```

a = 1

Problem

- Difficult to identify the **execution flow**
- It is easy when it is a single function & file.

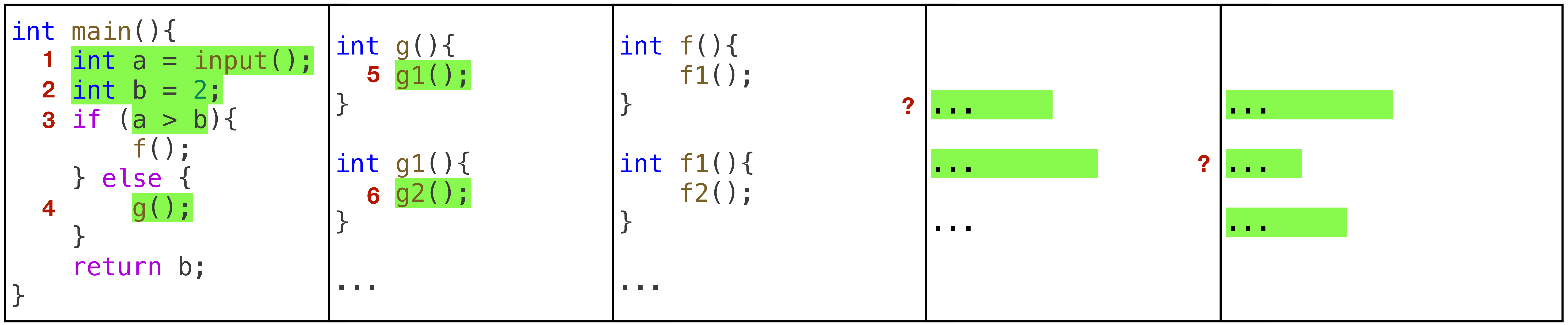
```c
int main(){
  1 int a = input();
  2 int b = 2;
  3 if (a > b){
        f();
    } else {
  4     g();
    }
    return b;
}
```

main.c

a = 1

Problem

- Difficult to identify the **execution flow**
- It is easy when it is a single function & file.
- But It's hard when there are complex and big codes.



```
int main(){
  1 int a = input();
  2 int b = 2;
  3 if (a > b){
        f();
    } else {
  4     g();
    }
    return b;
}
```

```
int g(){
  5 g1();
}

int g1(){
  6 g2();
}

...
```

```
int f(){
    f1();
}

int f1(){
    f2();
}

...
```

```
? ...
  ...
  ...
```

```
  ...
? ...
  ...
```

main.c g.c f.c ...

a = 1

Problem

- Difficult to identify the execution flow
- It is easy when it is a single function & file.
- But it's hard when there are complex and big codes.

Why is it important to know the execution flow?

```
int main(){
    int a = input();
    int b = 2;
    if (a > b){
        f();
    } else {
        g();
    }
    return b;
}
```

```
int f(){
    f1();
}

int f1(){
    f2();
}

...
```

```
int g(){
    g1();
}

int g1(){
    g2();
}

...
```

```
...

...

...
```

```
...

...

...
```

a = 1

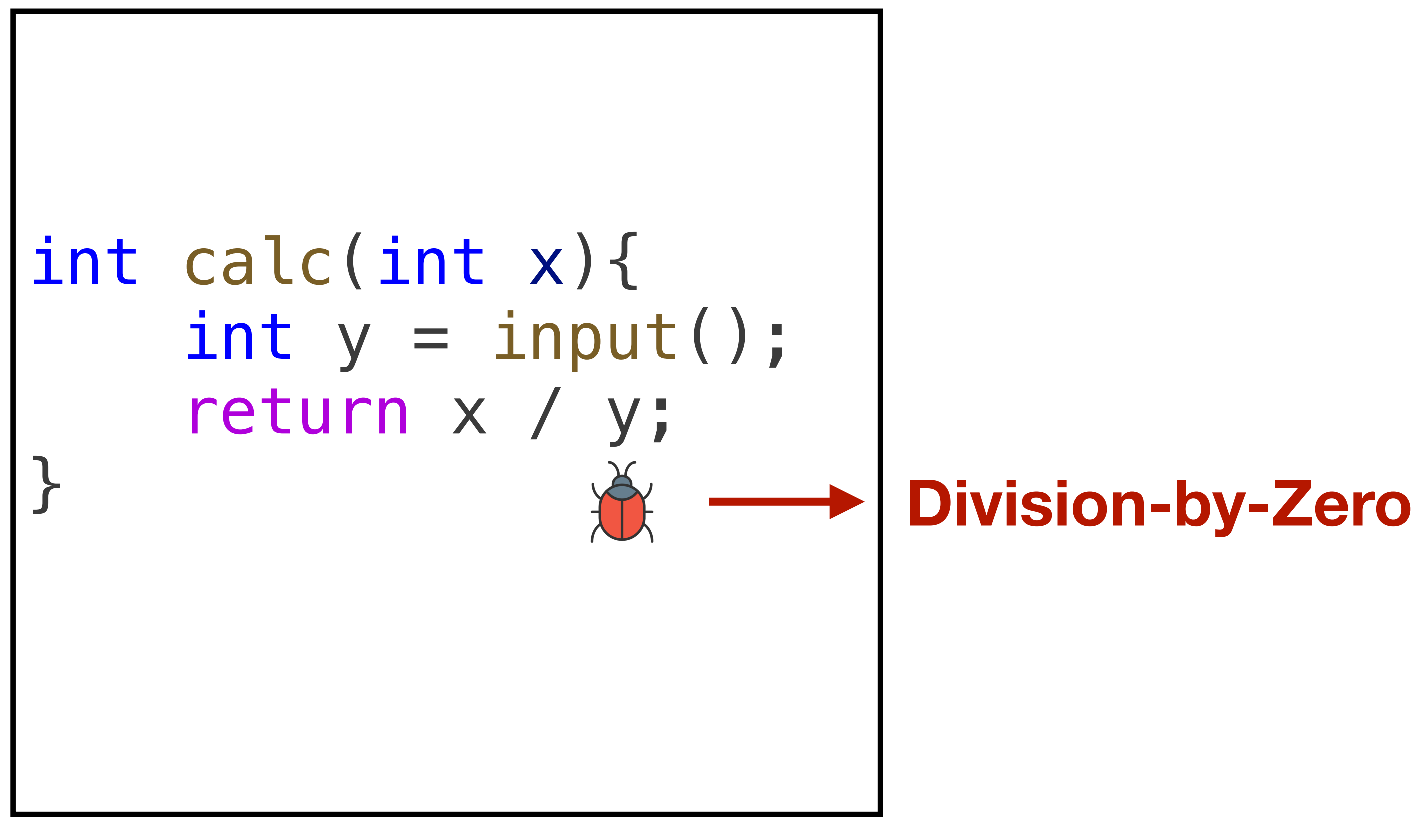
Importance of Execution Flow

- It's important to understand the **context** and **execution flow** of **errors**
- Understand the **root cause** of the error and perform **accurate patches**
- If not, errors are likely to occur **again**

```
int calc(int x){
    int y = input();
    return x / y;
}
```

Importance of Execution Flow

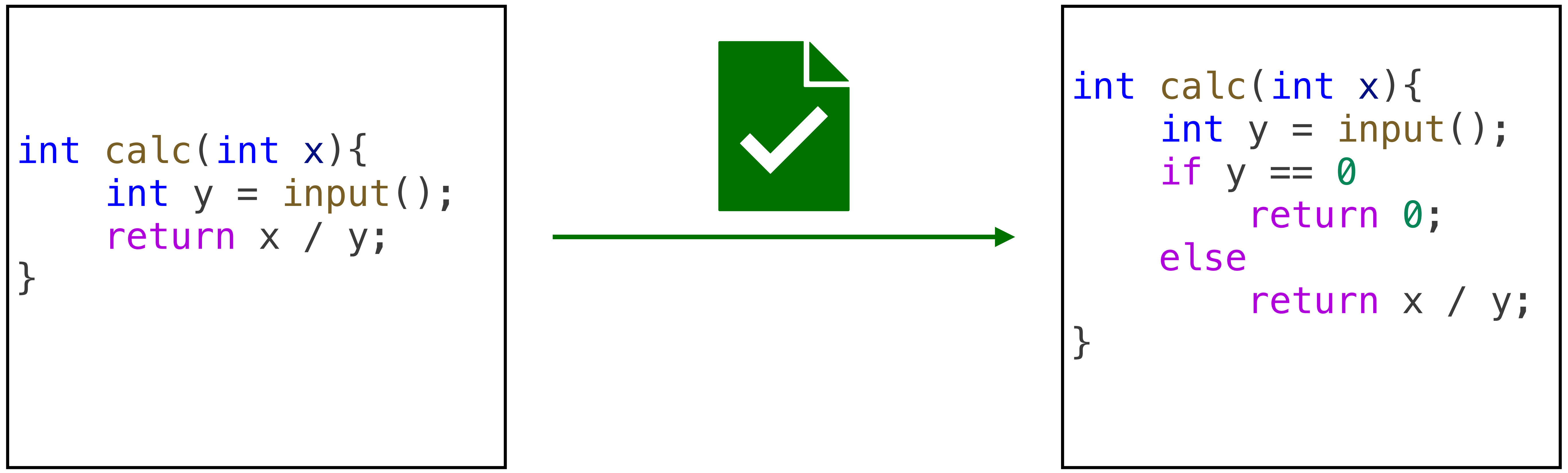
- It's important to understand the **context** and **execution flow** of **errors**
- Understand the **root cause** of the error and perform **accurate patches**
- If not, errors are likely to occur **again**



```
int calc(int x){
    int y = input();
    return x / y;
}
```

Importance of Execution Flow

- It's important to understand the **context** and **execution flow** of **errors**
- Understand the **root cause** of the error and perform **accurate patches**
- If not, errors are likely to occur **again**

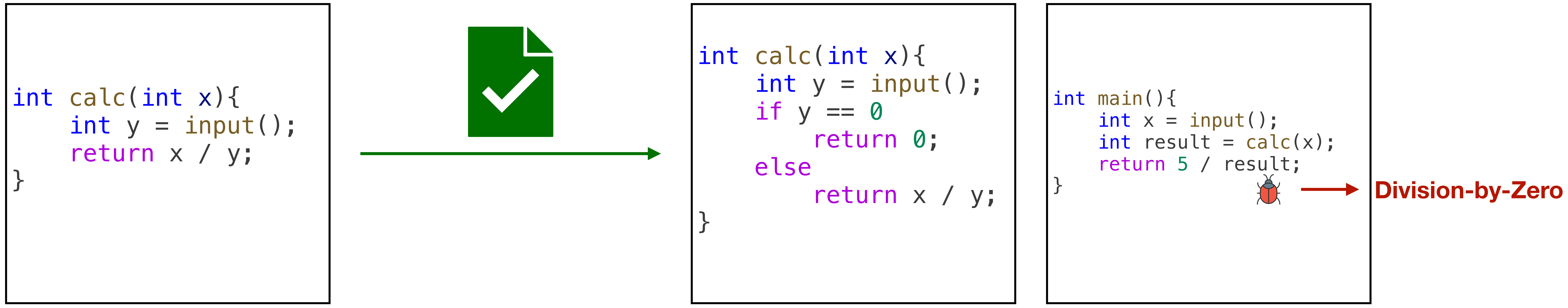


```
int calc(int x){
    int y = input();
    return x / y;
}
```

```
int calc(int x){
    int y = input();
    if y == 0
        return 0;
    else
        return x / y;
}
```

Importance of Execution Flow

- It's important to understand the **<u>context</u>** and **<u>execution flow</u>** of **<u>errors</u>**
- Understand the **<u>root cause</u>** of the error and perform **<u>accurate patches</u>**
- If not, errors are likely to occur **<u>again</u>**



```
int calc(int x){
    int y = input();
    return x / y;
}
```

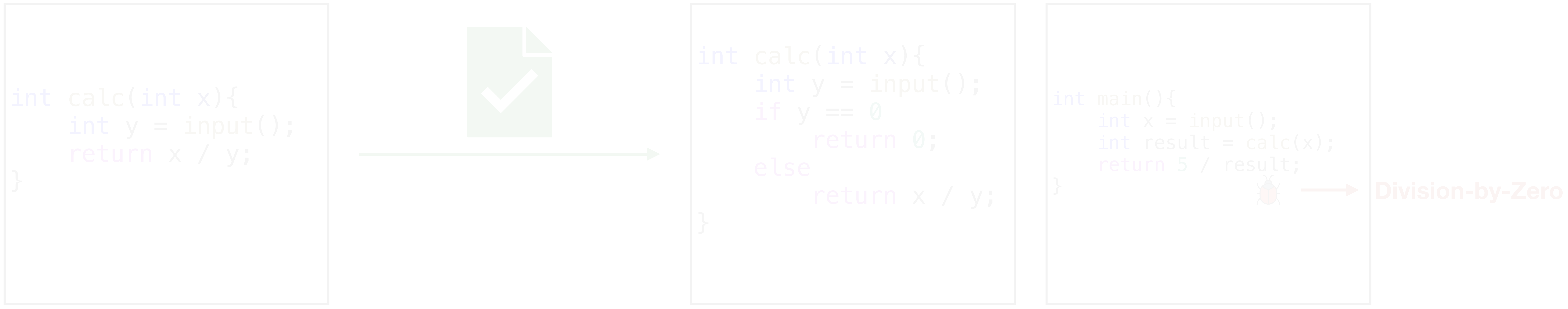
```
int calc(int x){
    int y = input();
    if y == 0
        return 0;
    else
        return x / y;
}
```

```
int main(){
    int x = input();
    int result = calc(x);
    return 5 / result;
}
```

Importance of Execution Flow

- It's important to understand **the context and execution flow of errors**
- Understand the **root cause of the error** and **perform accurate patches**
- If not, errors are likely to occur again

We need to know the execution flow!



```
int calc(int x){
    int y = input();
    return x / y;
}
```

```
int calc(int x){
    int y = input();
    if y == 0
        return 0;
    else
        return x / y;
}
```

```
int main(){
    int x = input();
    int result = calc(x);
    return 5 / result;
}
```

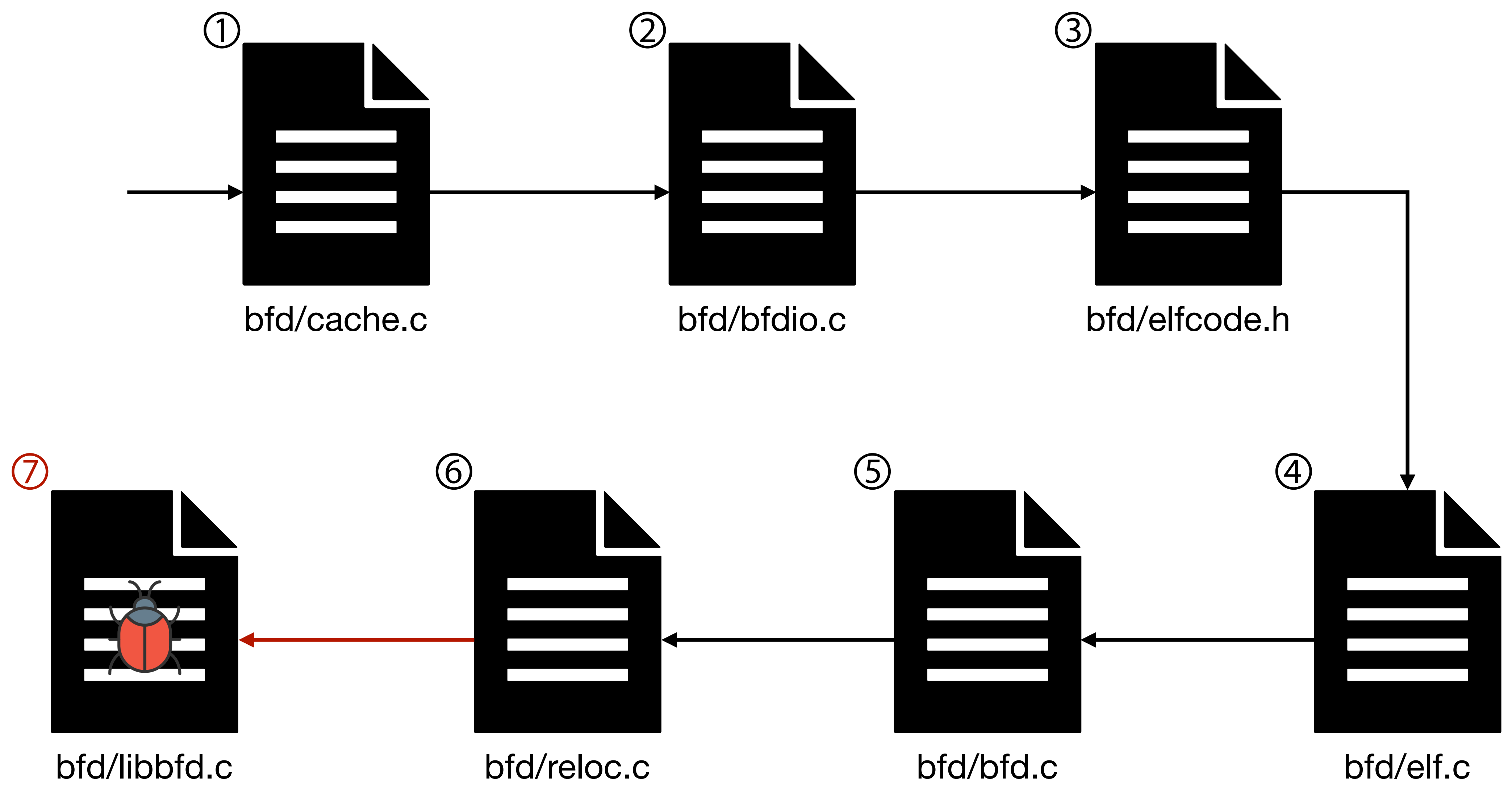
CodeVisFlow Overview #1

- **CodeFlowVis**
    - Provides code coverage and execution flow results at the same time
    - Can know what code a particular input went through when an error occurred
    - Gather the execution flow information by llvm pass

Errors and Complexity

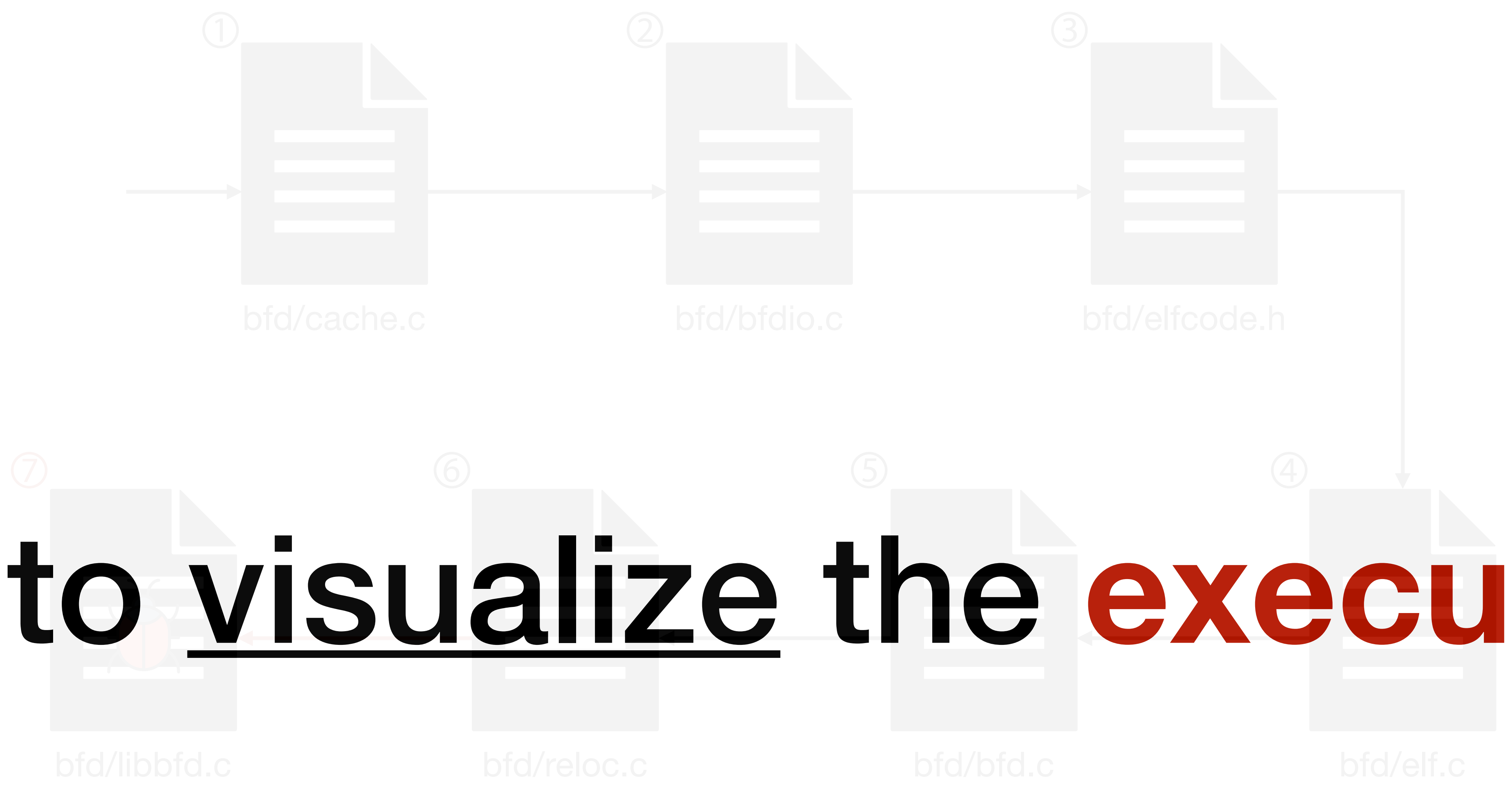
- The size of the software is growing.
- As a result, the flow of errors is becoming more complicated
  → Hard to understand without the execution flow
- Example
  - CVE-2017-8396

Errors and Complexity



- **17** Calls / Returns for **6** files, including cache.c, bfdio.c, etc
- Pointer dereference occurs in libbfd.c
- Error after such complex function calls and execution flows

Errors and Complexity



We need to visualize the execution flow!

- 17 Calls / Returns for 6 files, including cache.c, bfdio.c, etc
- Pointer dereference occurs in libbfd.c
- Error after such complex function calls and execution flows

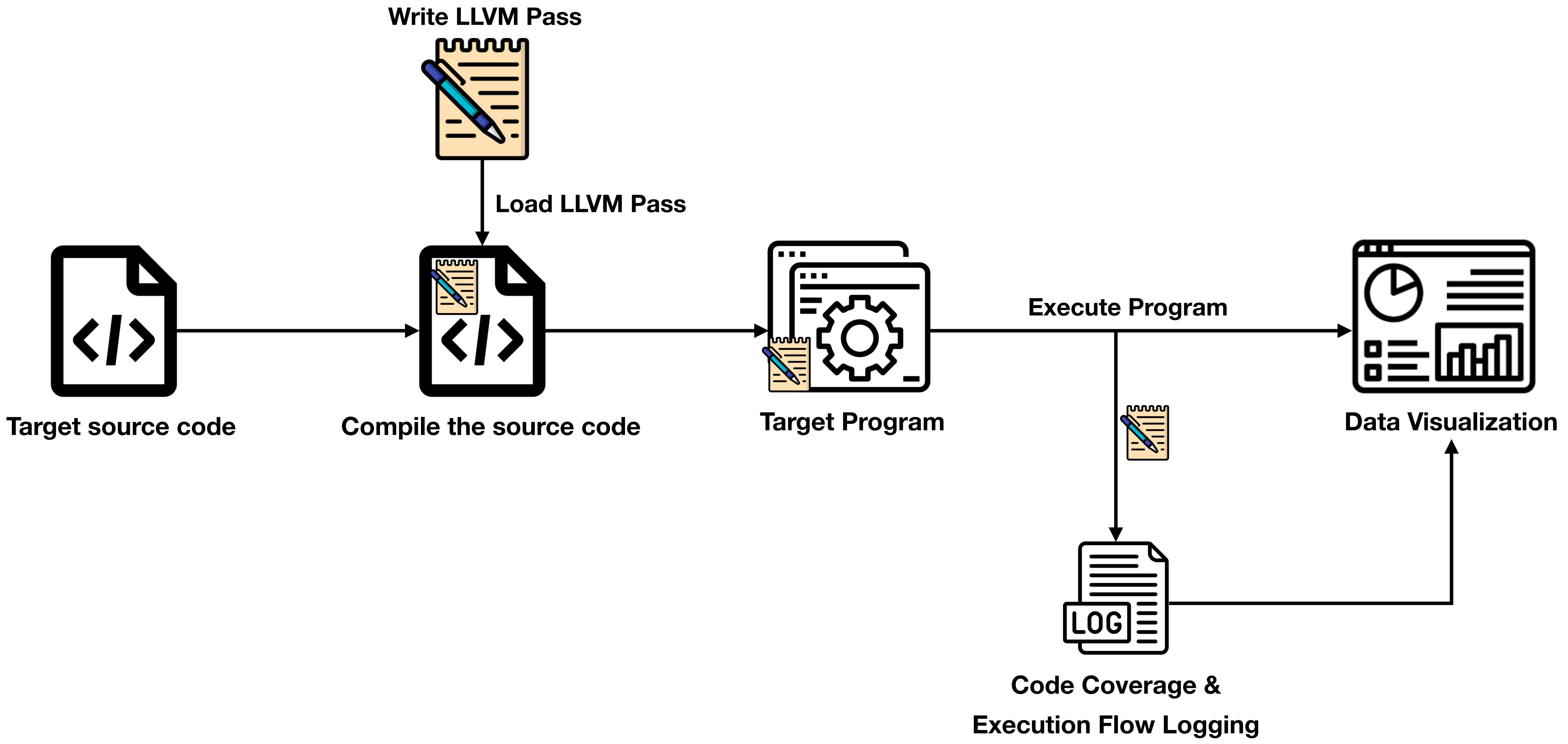
CodeVisFlow Overview #2

- **CodeFlowVis**
  - Can clearly identify which function call flow caused the error
  - Visualize execution flow and code coverage
    - By the information gathered in #1

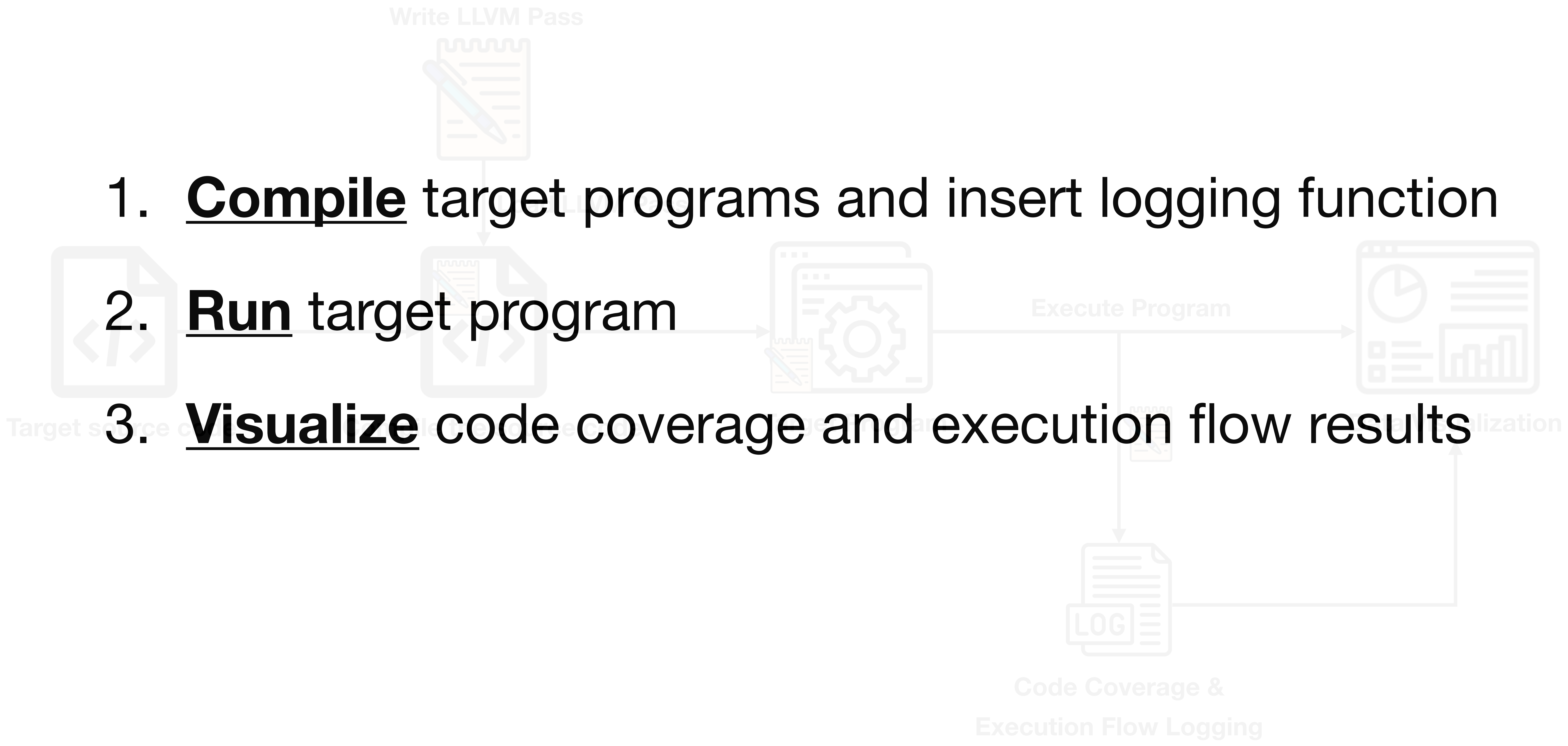
Goal

- There are two goals to achieve this research
    1. Develop a system that records execution flow information
    2. Visualize code coverage and execution flow

Overview

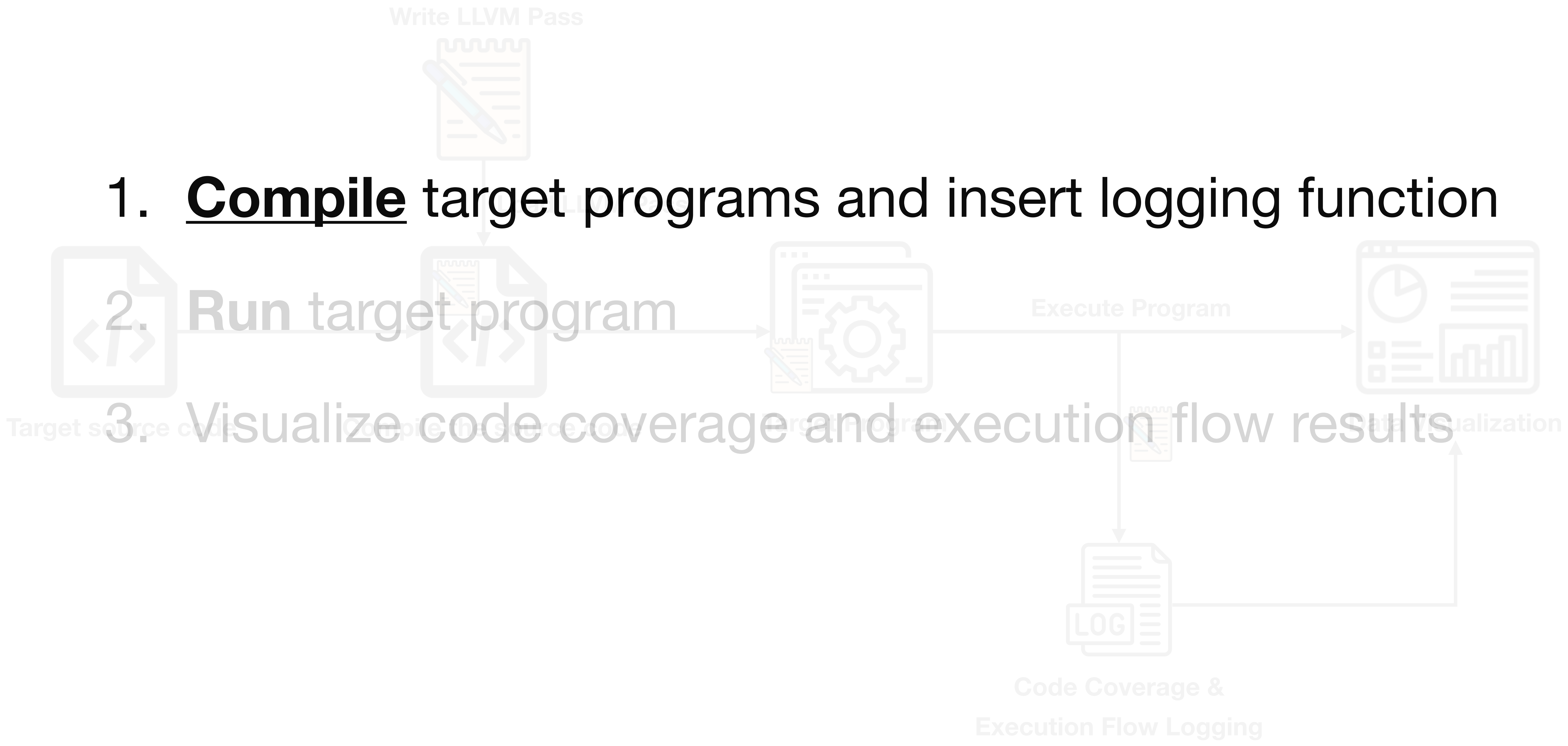

Write LLVM Pass
Load LLVM Pass
Target source code
Compile the source code
Target Program
Execute Program
Data Visualization
LOG
Code Coverage &
Execution Flow Logging

Overview



1. **Compile** target programs and insert logging function
2. **Run** target program
3. **Visualize** code coverage and execution flow results

Overview

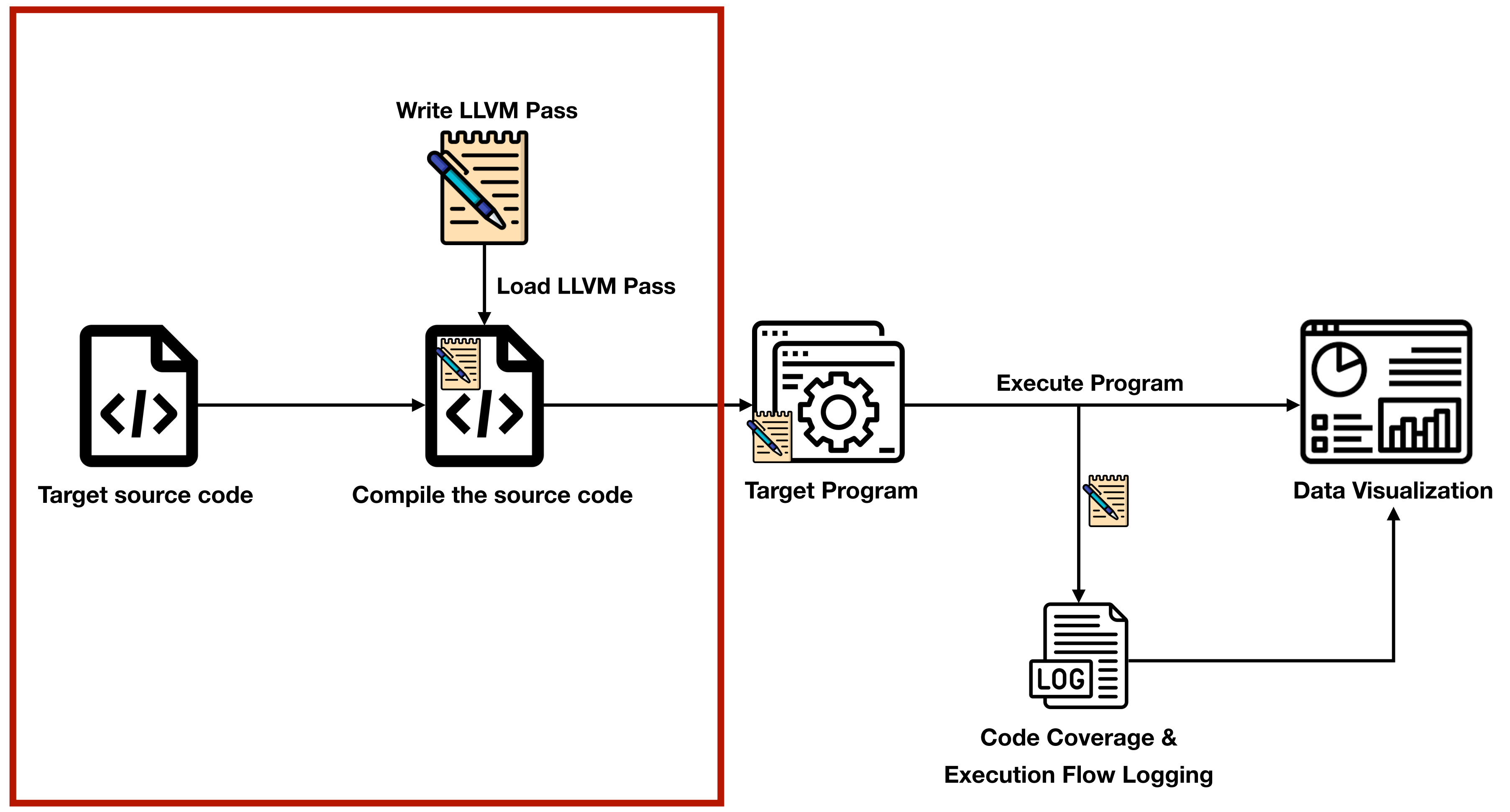


1. **Compile** target programs and insert logging function
2. **Run** target program
3. Visualize code coverage and execution flow results

Method

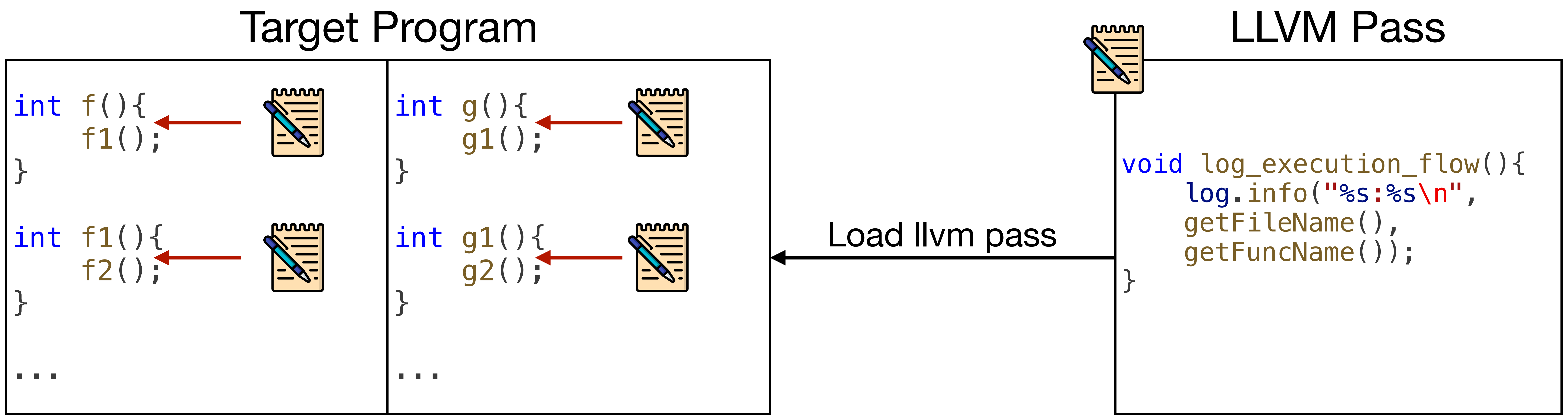
1. Compile target programs and insert logging function

- Compilation of the target program to be analyzed
- Based on the LLVM compiler to load a special purpose LLVM Pass
- LLVM pass gathers and records execution flow



LLVM Pass

- LLVM Pass is a component of the LLVM compiler infrastructure
  - analyzes or transforms the Intermediate Representation (IR) of a program
- Insert logging function for every function in target program with llvm pass



```
int f(){
    f1();
}

int f1(){
    f2();
}

...
```

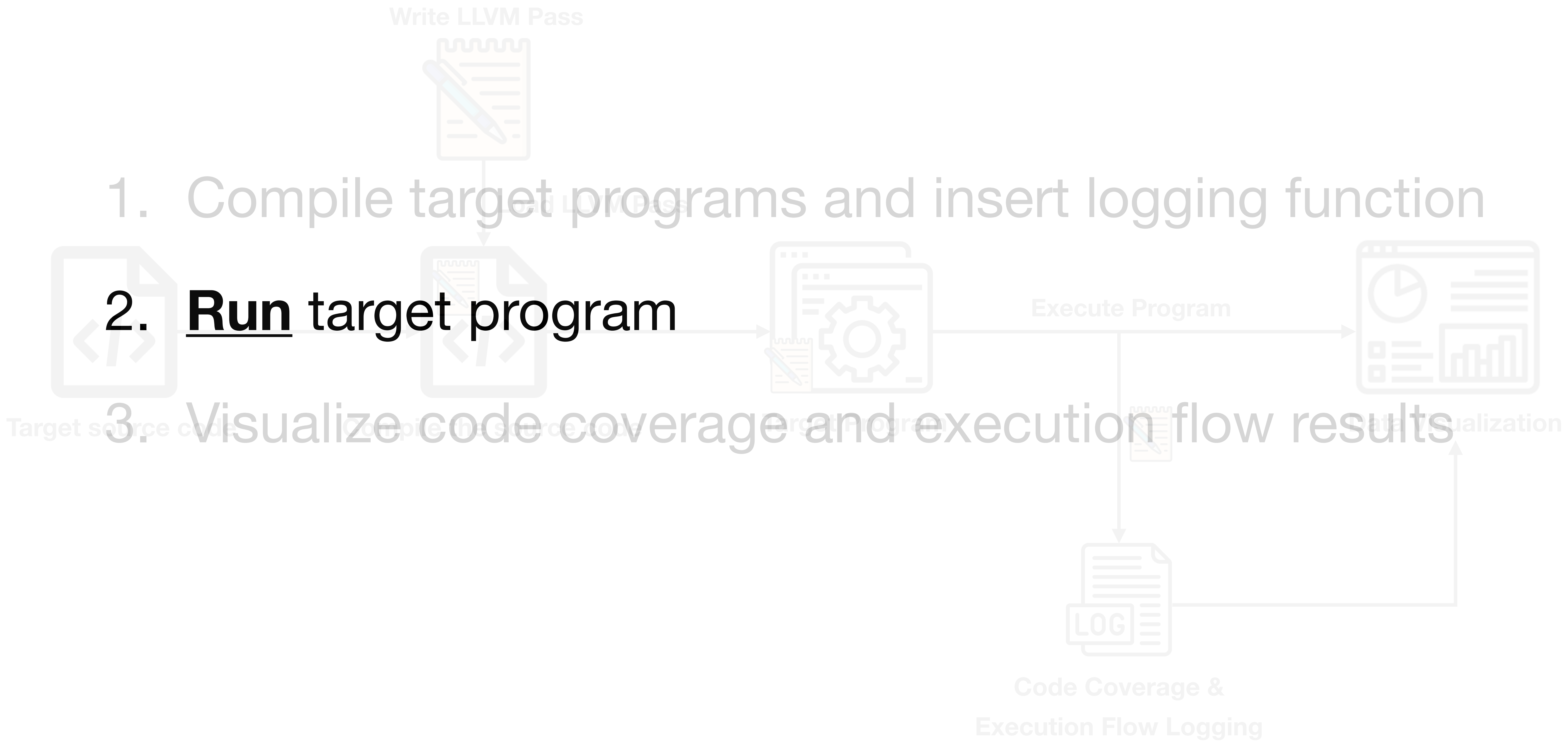
```
int g(){
    g1();
}

int g1(){
    g2();
}

...
```

```
void log_execution_flow(){
    log.info("%s:%s\n",
    getFileName(),
    getFuncName());
}
```

Overview

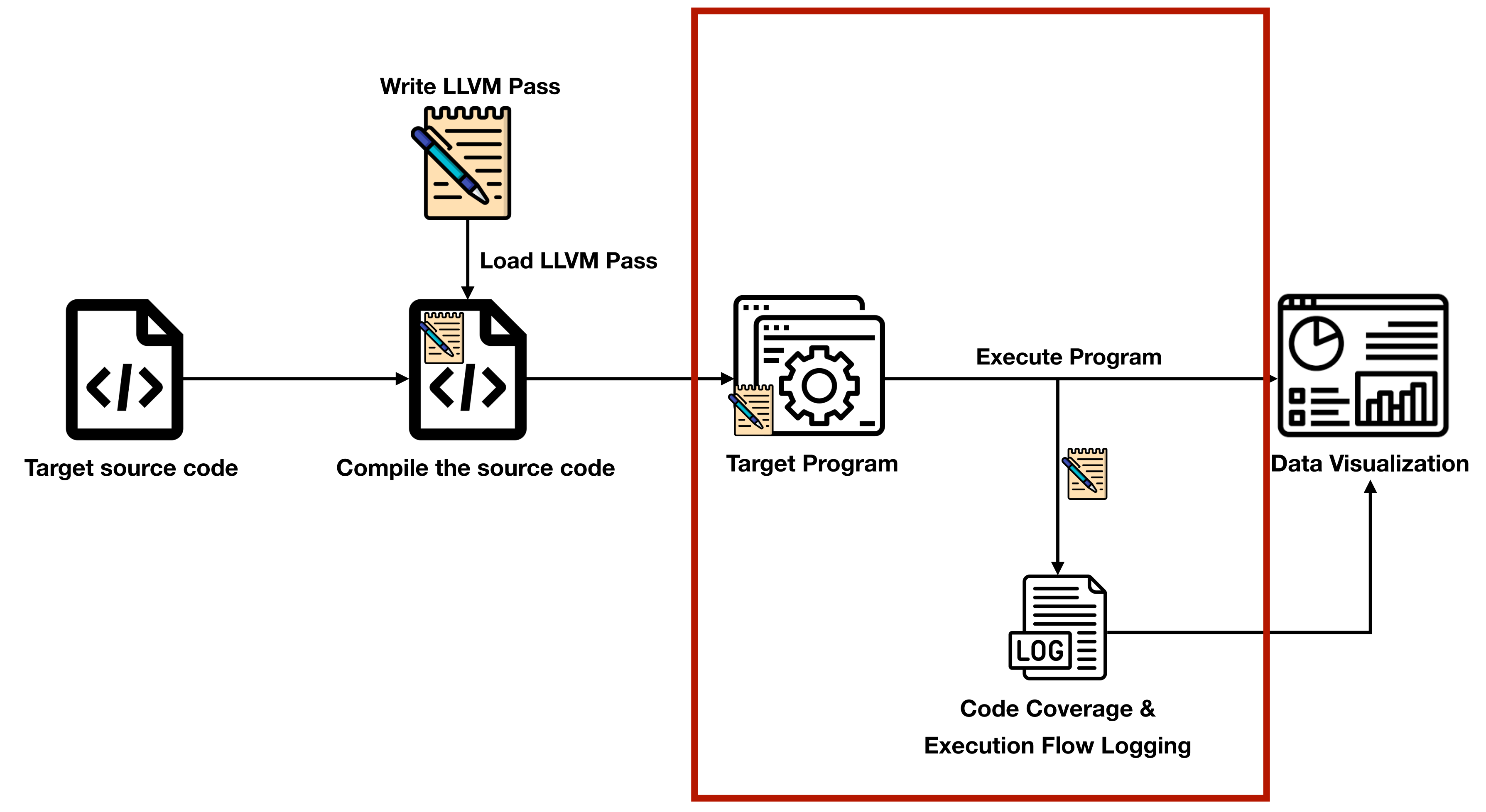


1. Compile target programs and insert logging function
2. **<u>Run</u>** target program
3. Visualize code coverage and execution flow results

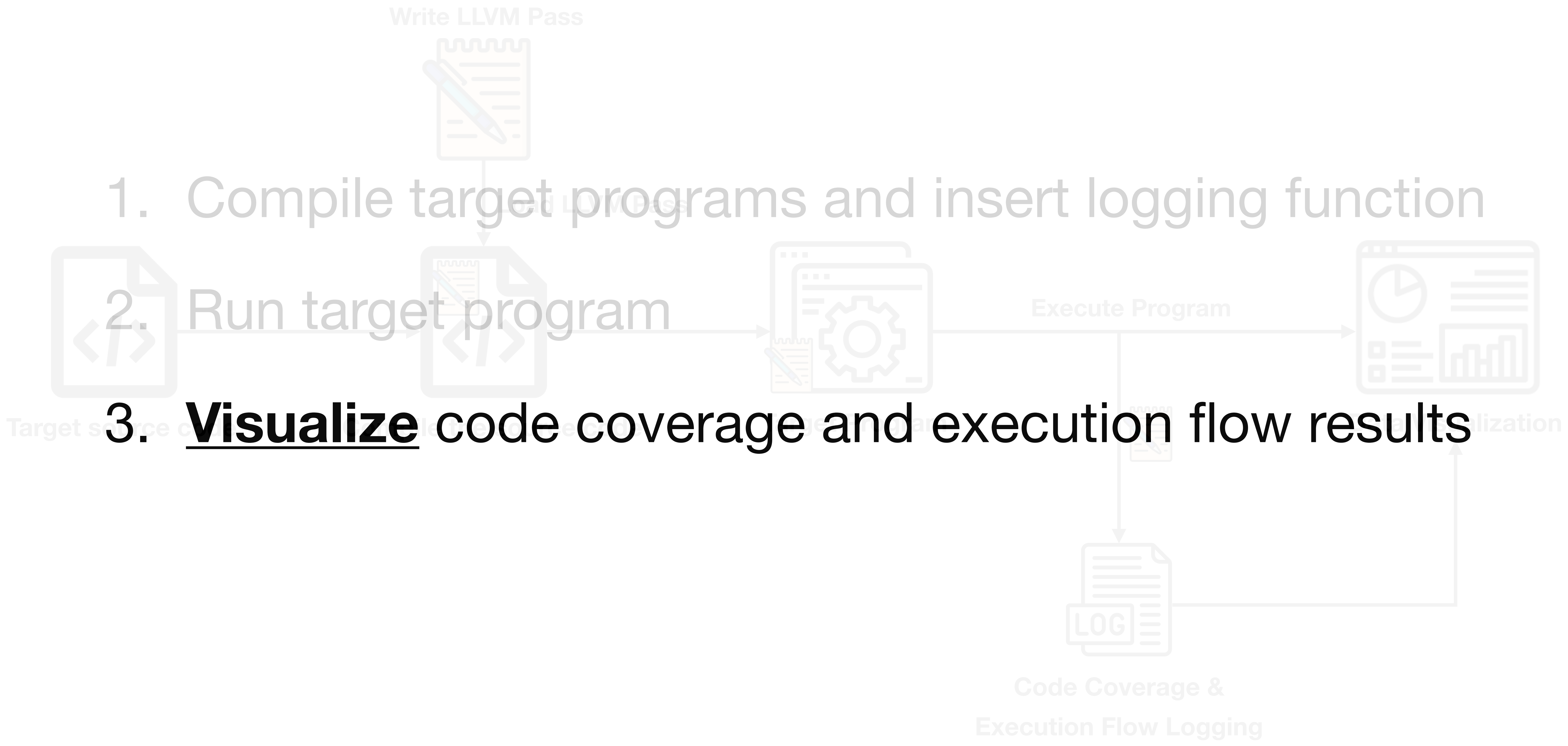
Method

2. Run target program

- Target program is executed with a test case
- The program collects all function calls executed
  - Through code inserted by LLVM Pass.
- LLVM Pass records execution flow information
- GCOV records code coverage information



Overview

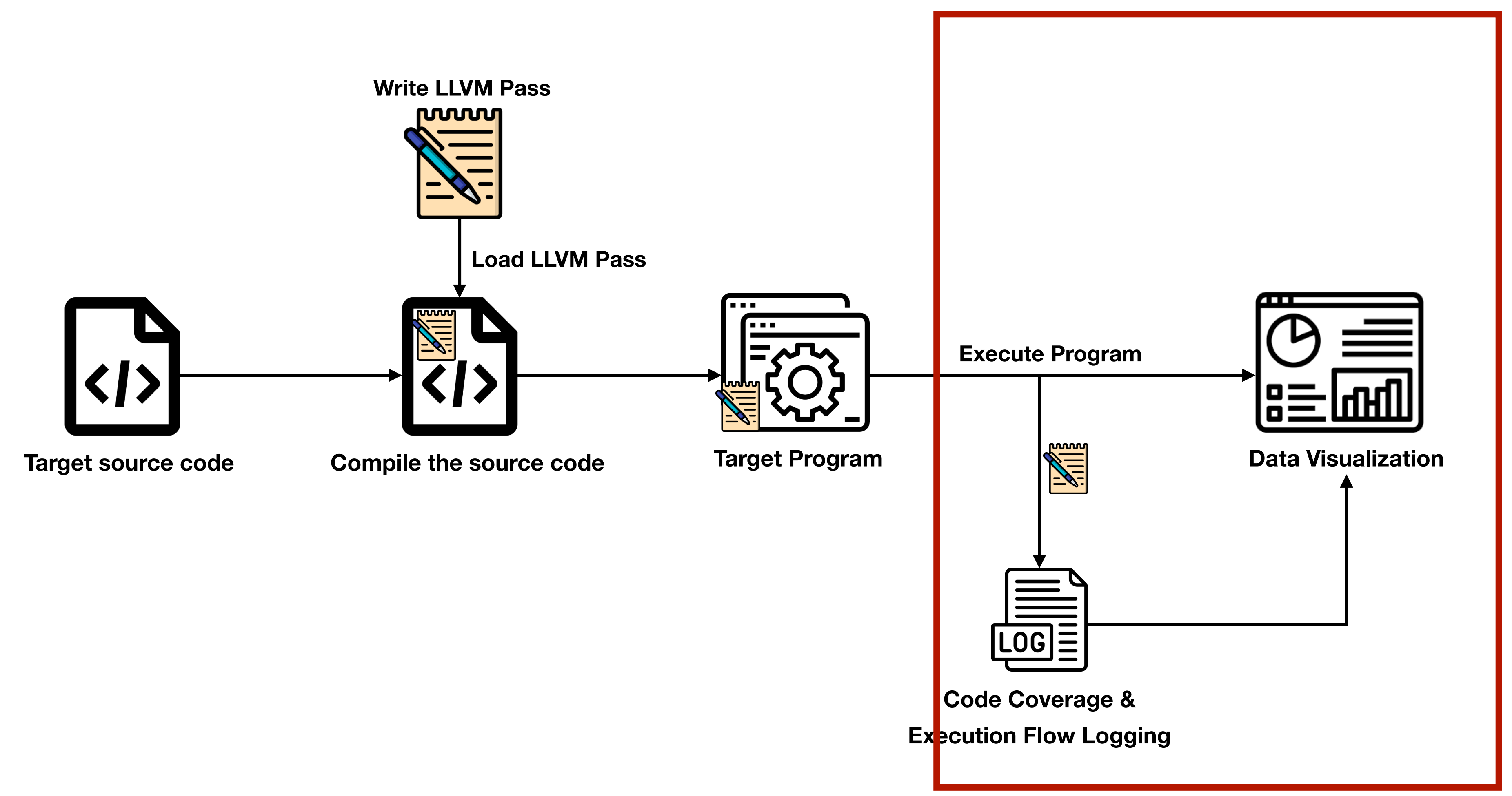


1. Compile target programs and insert logging function
2. Run target program
3. <u>**Visualize**</u> code coverage and execution flow results

Method

3. Visualize code coverage and execution flow results

- Coverage data is read through LCOV[3]
  - analyzed, and converted into an HTML report

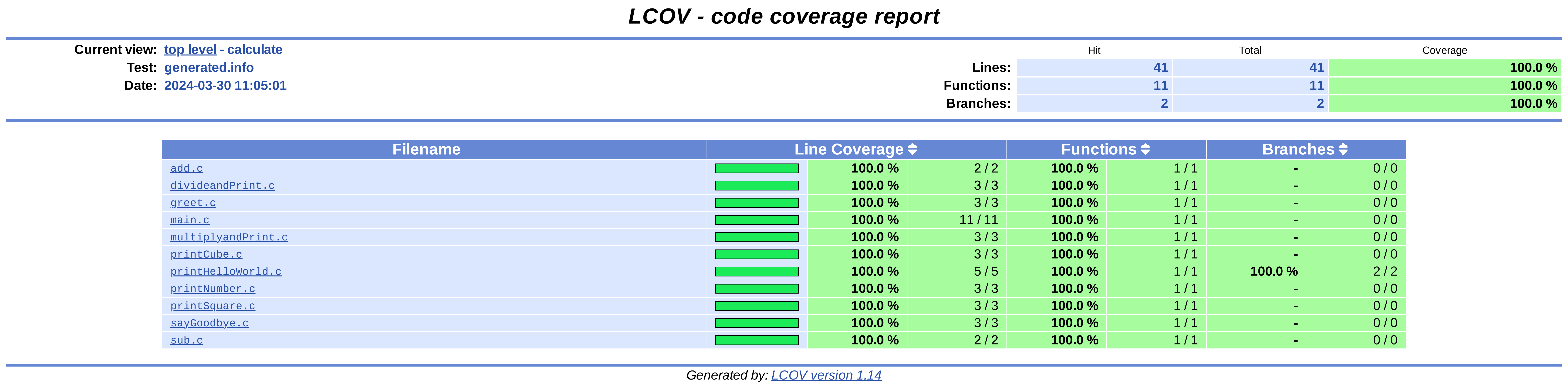


[3] LCOV, https://github.com/linux-test-project/lcov

Method

3. Visualize code coverage and execution flow results

•

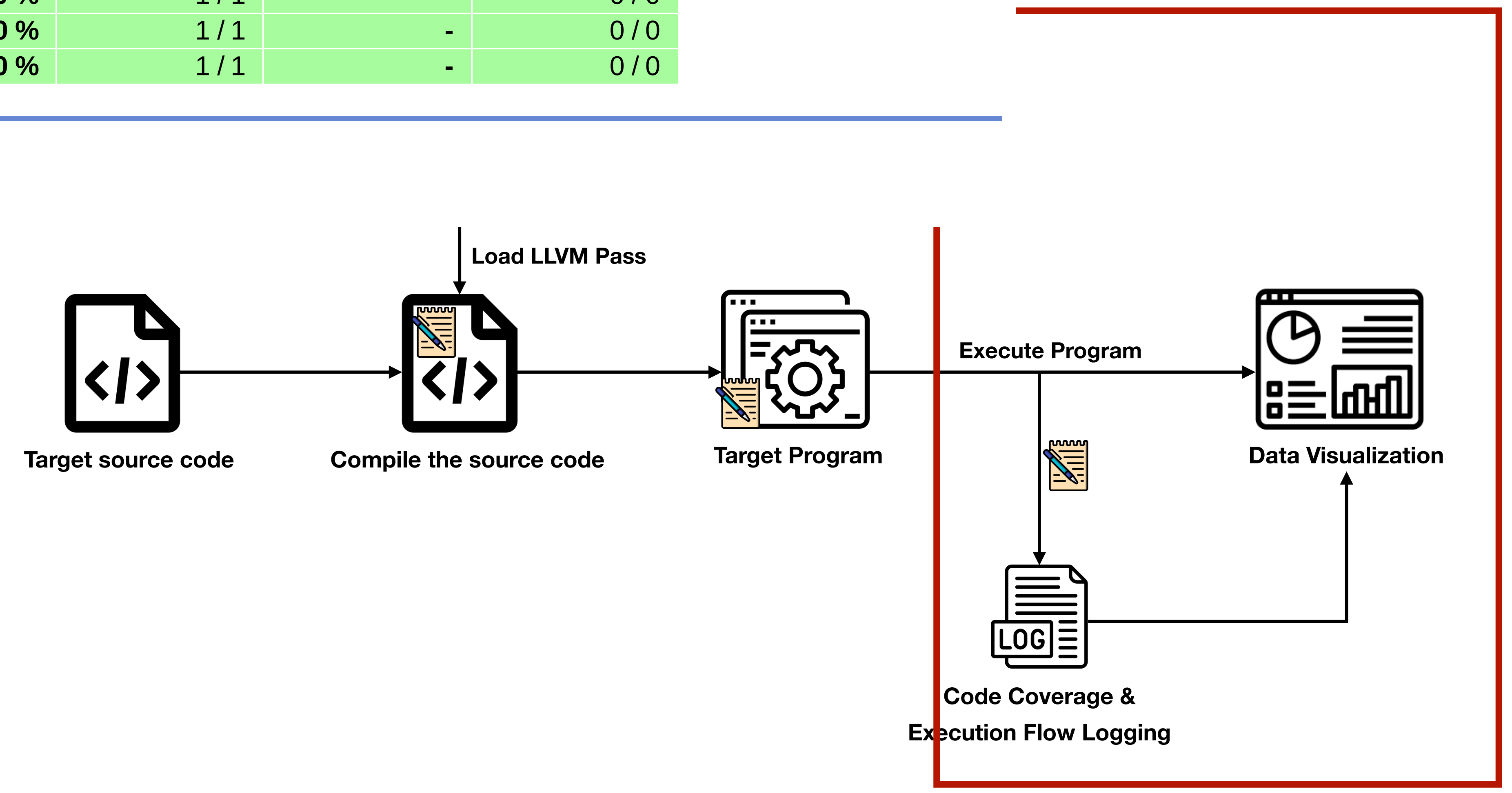


LCOV - code coverage report

Current view:	top level - calculate		Hit	Total	Coverage
Test:	generated.info	**Lines:**	41	41	100.0 %
Date:	2024-03-30 11:05:01	**Functions:**	11	11	100.0 %
		Branches:	2	2	100.0 %

Filename	Line Coverage			Functions		Branches	
add.c	🟩	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
divideandPrint.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
greet.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
main.c	🟩	100.0 %	11 / 11	100.0 %	1 / 1	-	0 / 0
multiplyandPrint.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
printCube.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
printHelloWorld.c	🟩	100.0 %	5 / 5	100.0 %	1 / 1	100.0 %	2 / 2
printNumber.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
printSquare.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
sayGoodbye.c	🟩	100.0 %	3 / 3	100.0 %	1 / 1	-	0 / 0
sub.c	🟩	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0

Generated by: LCOV version 1.14



[3] LCOV, https://github.com/linux-test-project/lcov

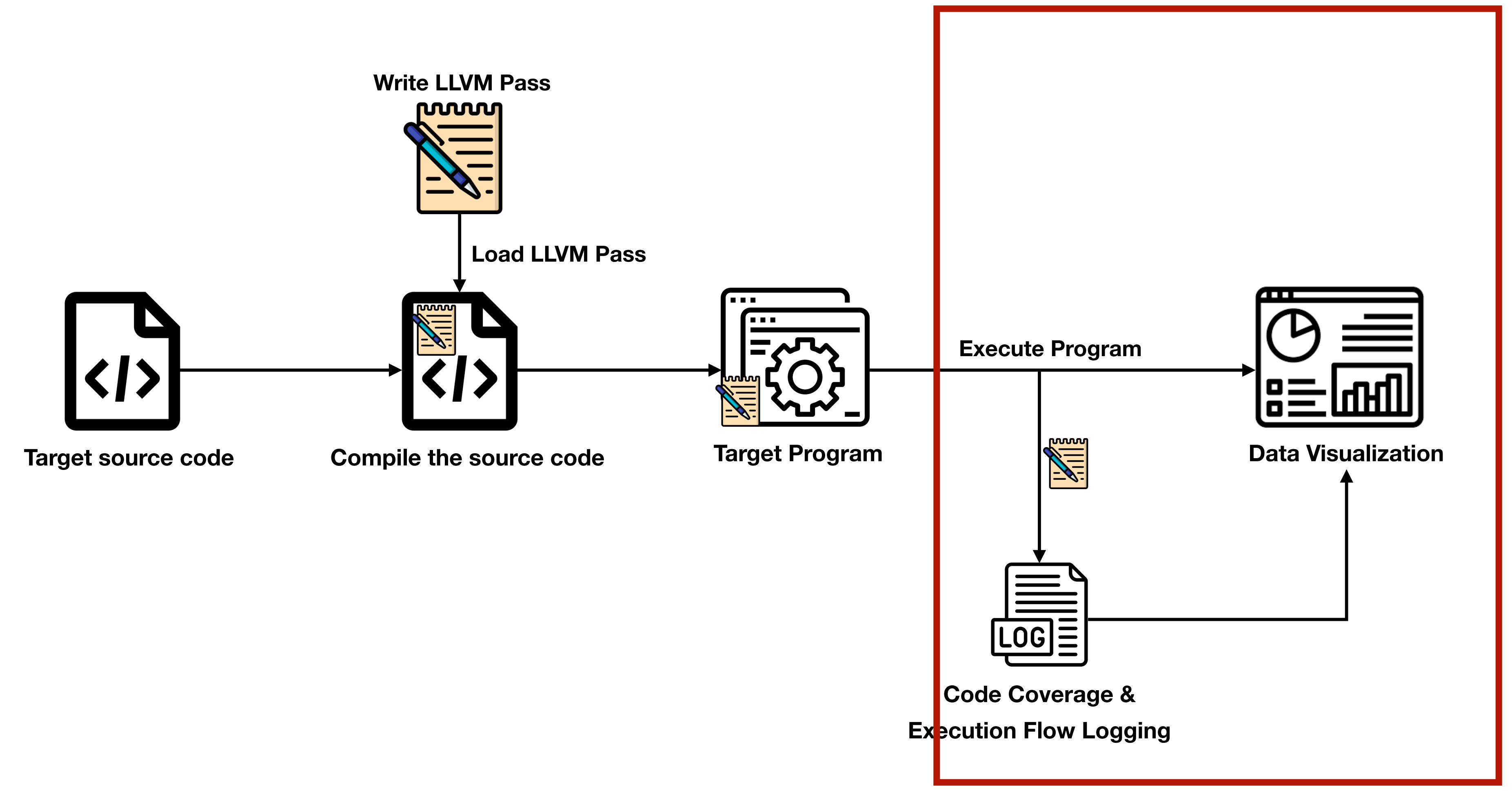
Method

3. Visualize code coverage and execution flow results

- The execution flow will be presented in two main forms for visualization

1) Visualize the execution flow with code coverage

   - Display the code cov & executed flow

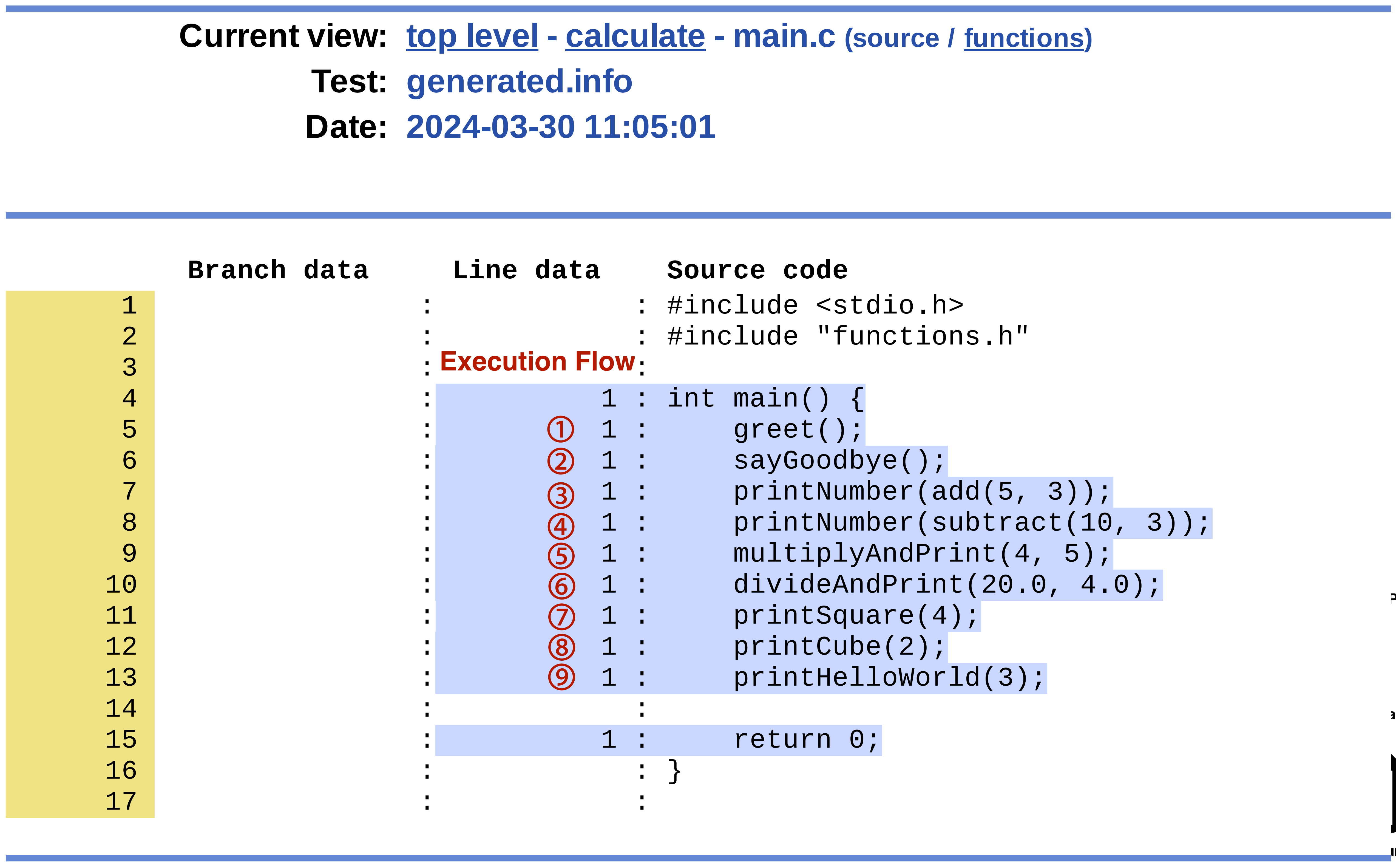


Method

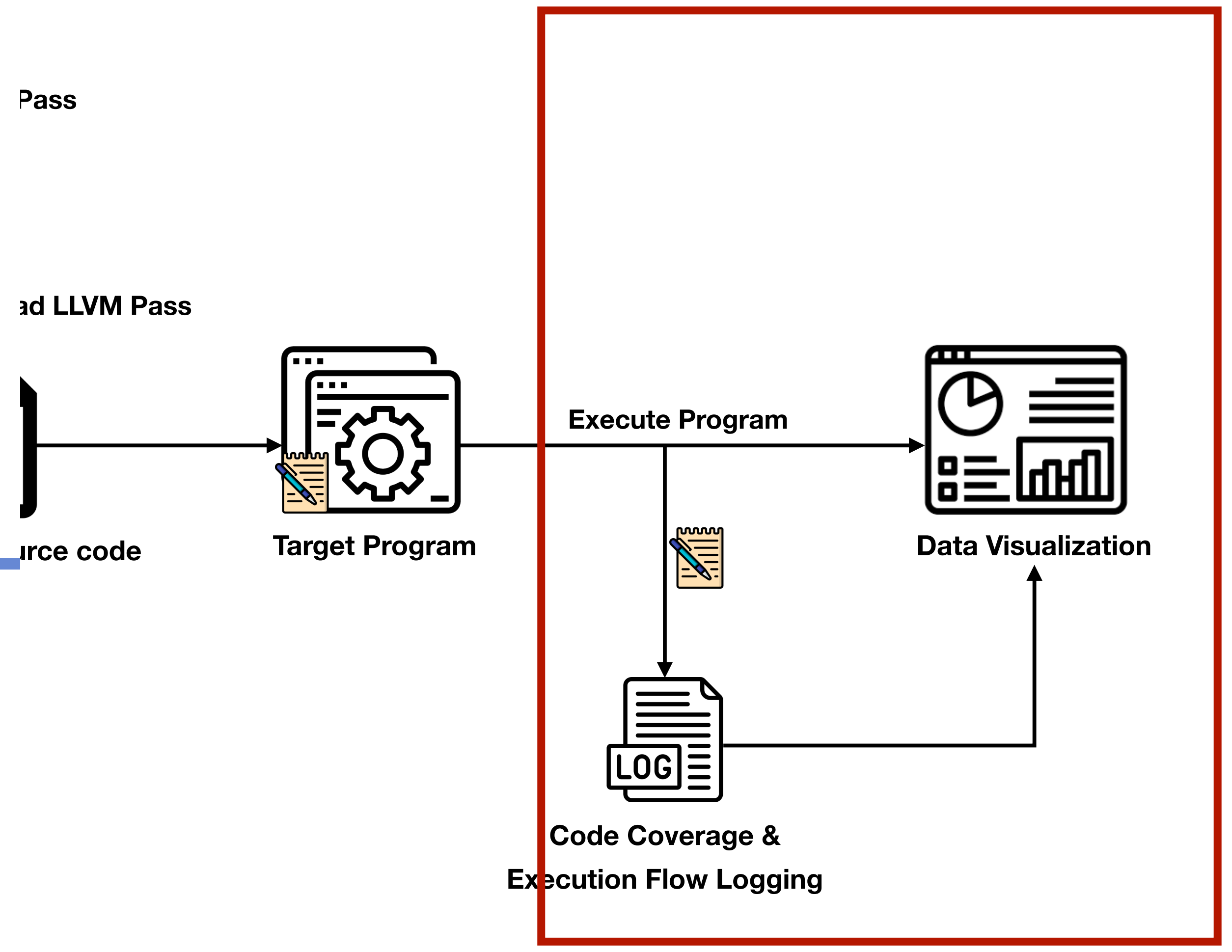
3. Visualize code c

- The execution visualization

1) Visualize the e


Current view: top level - calculate - main.c (source / functions)
Test: generated.info
Date: 2024-03-30 11:05:01

```
Branch data     Line data       Source code
                     :          : #include <stdio.h>
                     :          : #include "functions.h"
                     : Execution Flow :
                     :        1 : int main() {
                     :     ①  1 :     greet();
                     :     ②  1 :     sayGoodbye();
                     :     ③  1 :     printNumber(add(5, 3));
                     :     ④  1 :     printNumber(subtract(10, 3));
                     :     ⑤  1 :     multiplyAndPrint(4, 5);
                     :     ⑥  1 :     divideAndPrint(20.0, 4.0);
                     :     ⑦  1 :     printSquare(4);
                     :     ⑧  1 :     printCube(2);
                     :     ⑨  1 :     printHelloWorld(3);
                     :          :
                     :        1 :     return 0;
                     :          : }
                     :          :
```

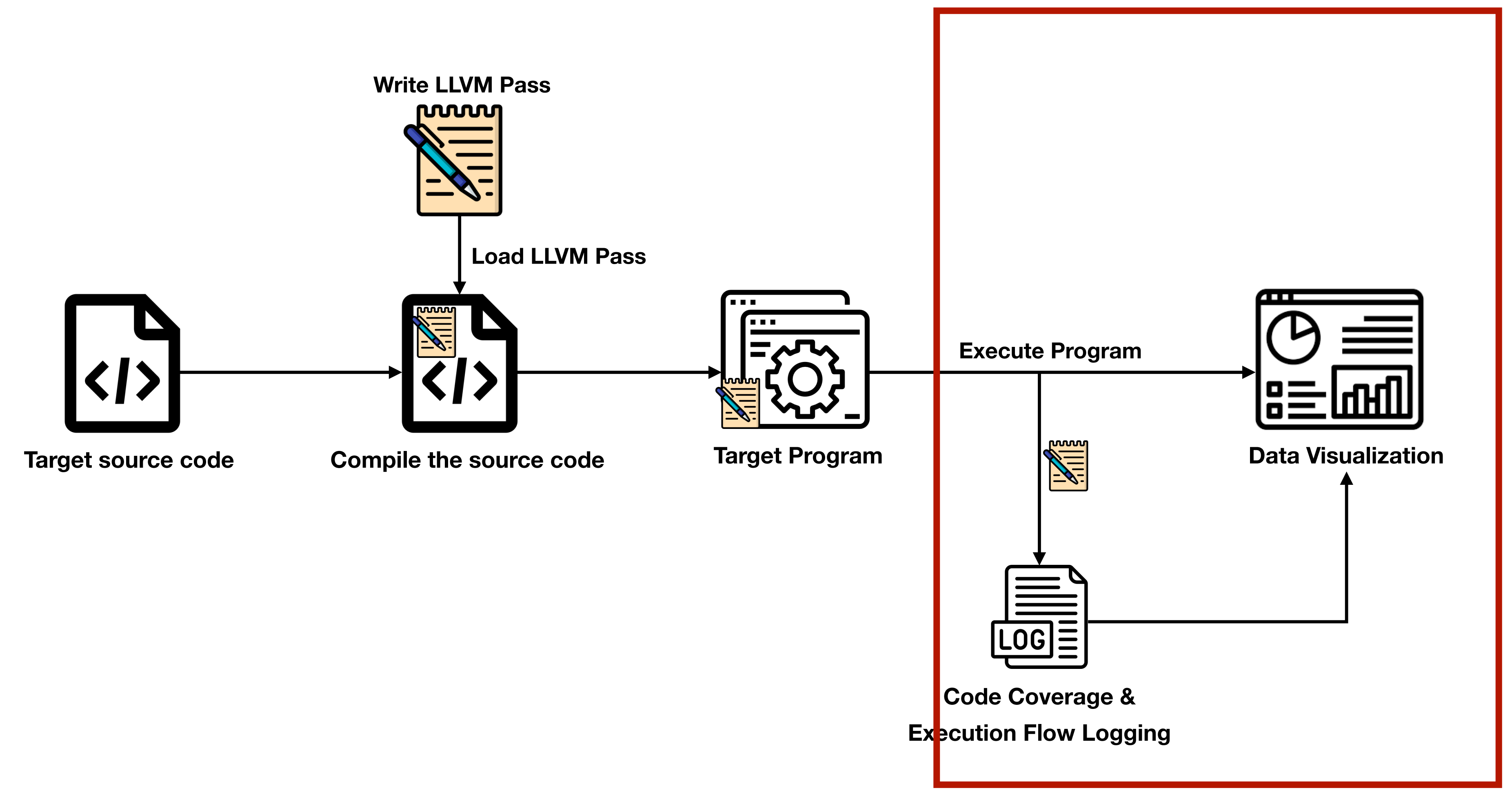


Method

3. Visualize code coverage and execution flow results

- The execution flow will be presented in two main forms for visualization

2) The execution flow in the form of state transition

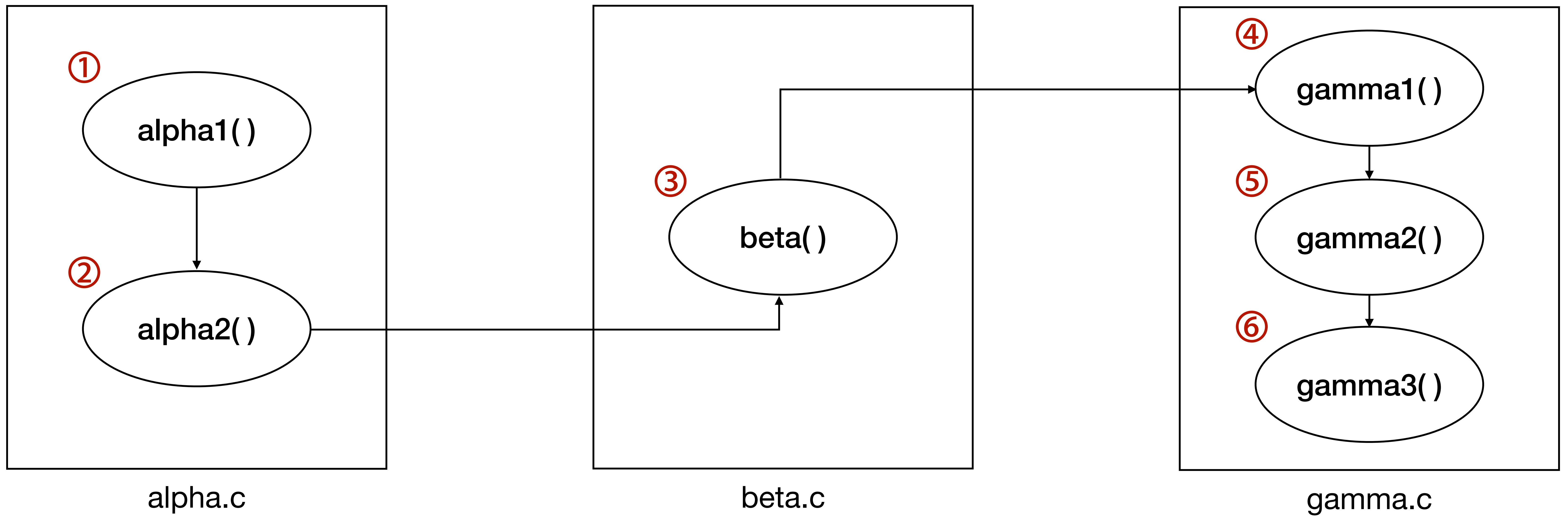


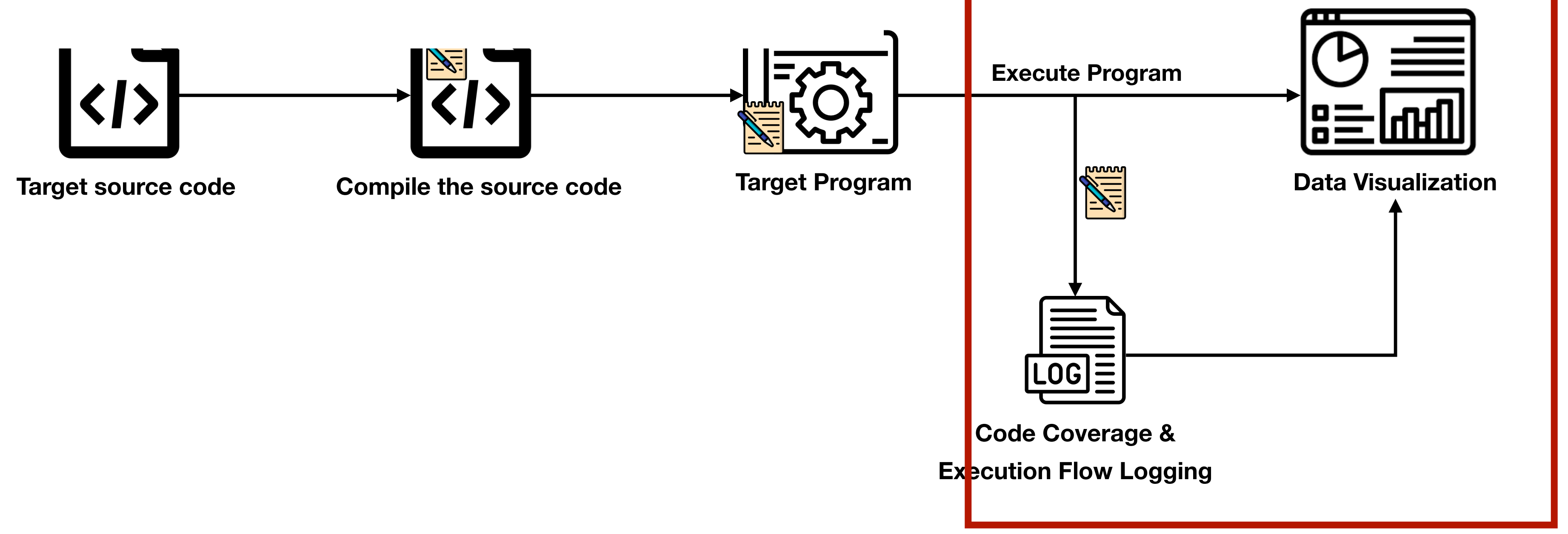
Method

3. Visualiz

- The e ation

2) The e





Expected Results

- Provides combined **code coverage** and **execution flow** information
- **Practical tools** for developers and security researchers
- Provides a **visual view** of the entire flow in the event of an error
- Contributing to the field of software testing through **open-source** disclosure

Evaluation Considerations

- Is code coverage clearly provided according to the input?
- Is the program execution flow clearly provided according to the input?
- Have our research's open-source achieved more than 1 GitHub star?

Researcher Competencies

- I proceed with projects to find **software errors**
  - have the experience to **trace execution flow**
- I have experience in **testing various targets**
  - apartments, smart farms, and satellites
  - Reported **50+** security vulnerabilities (errors)

Questions

Question 1. What do you target?

Questions

Question 1. What do you target?

Answer 1.

- Our target is a program based on LLVM.
- We will operate on various languages such as C, C++, Rust, and Kotlin based on LLVM.

Questions

Question 2. In what form is the system provided?

Questions

Question 2. In what form is the system provided?

Answer 2.

- An LLVM Pass and a Visualization tool will be provided.
- Compile target source code using the LLVM Pass
  - To collect information
- Display information with Visualizer

Questions

Question 3. What are the risks of this research?

Questions

Question 3. What are the risks of this research?

Answer 3.

- Not conducted pre-tests on languages such as Rust or Kotlin through LLVM.
- However, other research has demonstrated the feasibility of using LLVM
  - for the visualization and analysis of various languages[4]

[4] Rust Compiler Development Guide,
https://rustc-dev-guide.rust-lang.org/llvm-coverage-instrumentation.html

Summary

- Current tools do not provide **execution flow**.
- Identifying the execution flow is an **important issue** when analyzing **errors**.
- To solve this problem, we propose **CodeFlowVis**.
- **Visualize** code coverage and execution flow information.
- I have achieved outstanding results in the field of software testing.
- This tool will **contribute** to the field of software testing.