UVD: scalable UAF detector based on on-demand static analysis

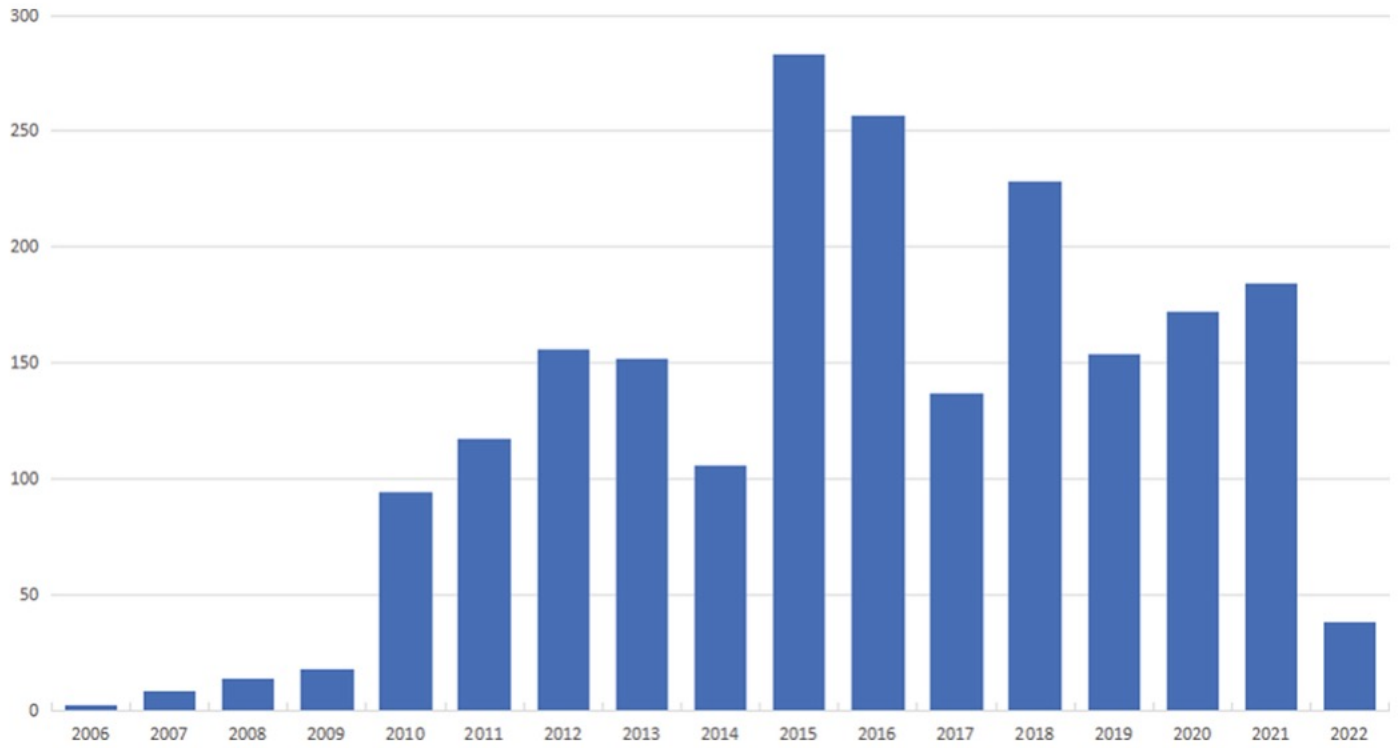
2023.2.1

Presenter: Shangzhi Xu

Background

Background

- C/C++ is the mainstream language for developing system software
- UAF has become one of the most common security vulnerabilities
- It is difficult to detect UAF vulnerabilities



UAF vulnerability statistics reported in the CVE database from 200601 to 202203.

Existing work

Static analysis to detect UAF

- There are less tools only focus on UAF
- Pinpoint & UAFChecker

Pinpoint

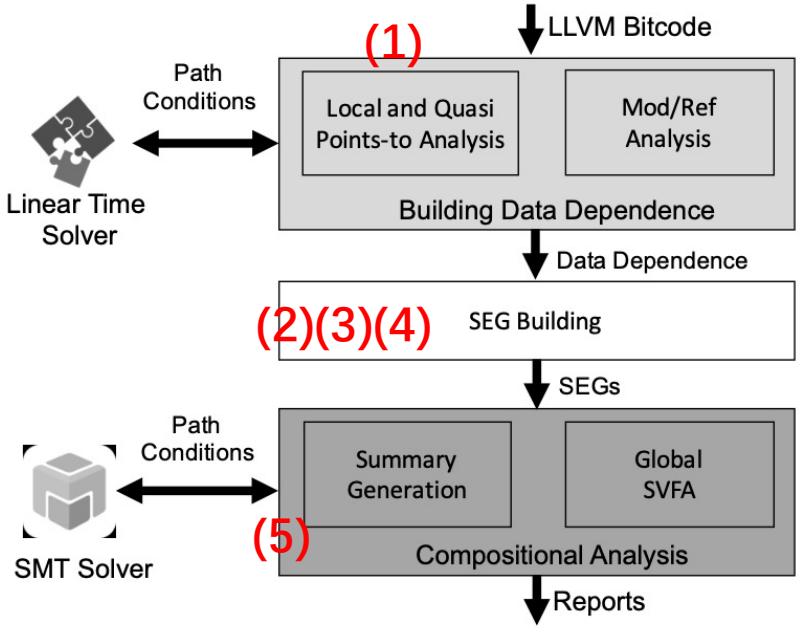
- Use new type of SVFG, the SEG

(1) Start with intra-procedural Path-Sensitive pointer analysis

(2) Create inter-procedural pointer result based on (1)

(3) Conduct intra-procedural analysis to create local SEG

(4) Connect SEG to get inter-procedural SEG based on (3)



Pinpoint

- SEG looks like Figure 1

(5) To detect UAF, start from *free(c)* to see if *printf(*f)* is reachable in the graph

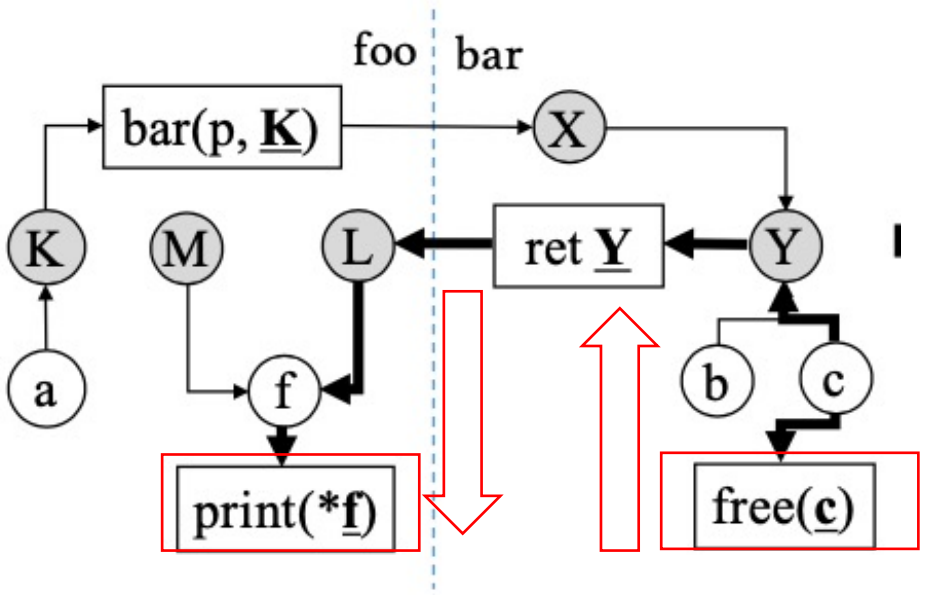


Figure 1

UAFChecker

- Pre-analysis: delete loop and dead code
- Tracker (marked in Figure 2):

  (1) Bottom-up analysis according to call-flow graph

  (2) Static analysis with finite-state machine

  (3) Symbolic execution to delete false-positive

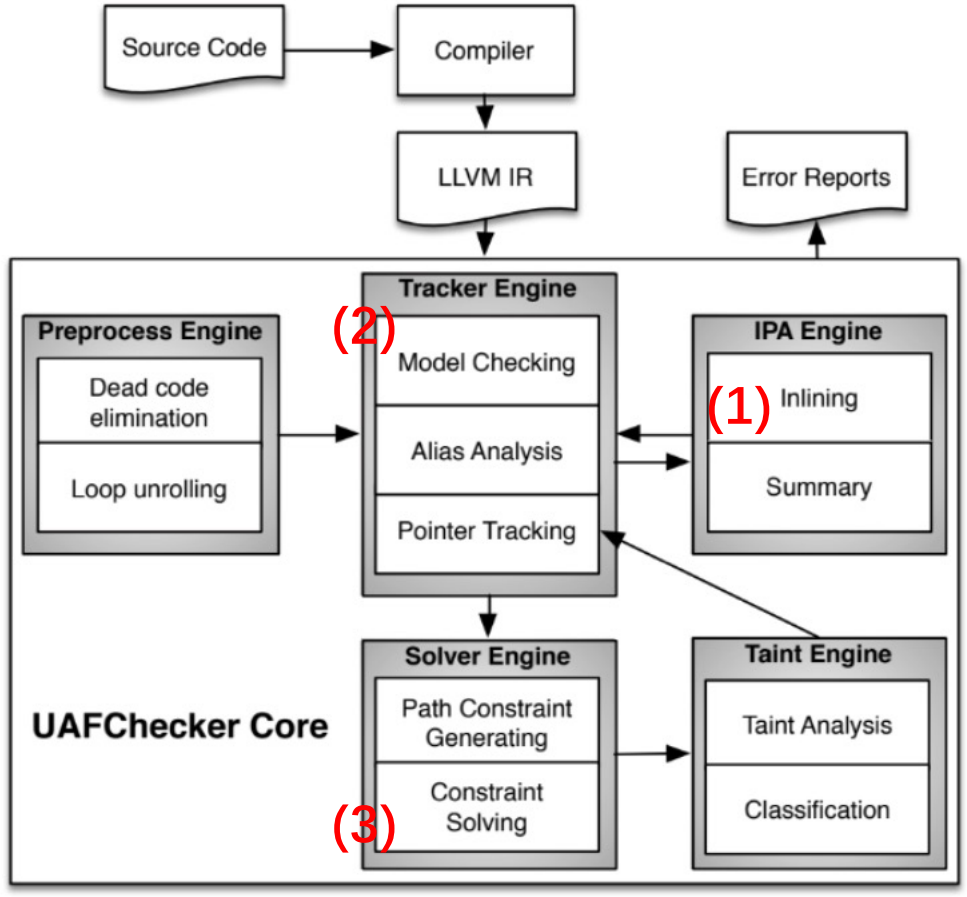


Figure 2

On-demand static analysis

- The most serious problem in static analysis is the scalability
- Many tools performs analysis exhaustively
- Can not handle large programs

Marple

- On-demand path sensitive analysis

(1) Start from potentially vulnerable statement

(2) Query backwardly and solve the path

(3) Report vulnerability

```c
#include <stdio.h>
#include <string.h>
void success() { puts("You Hava already controlled it."); }
void vulnerable() {
  char s[12];
  gets(s);
  puts(s); (1)
  return;
}
int main(int argc, char **argv) {
  vulnerable();
  return 0;
}
```

Buffer over-flow vulnerability

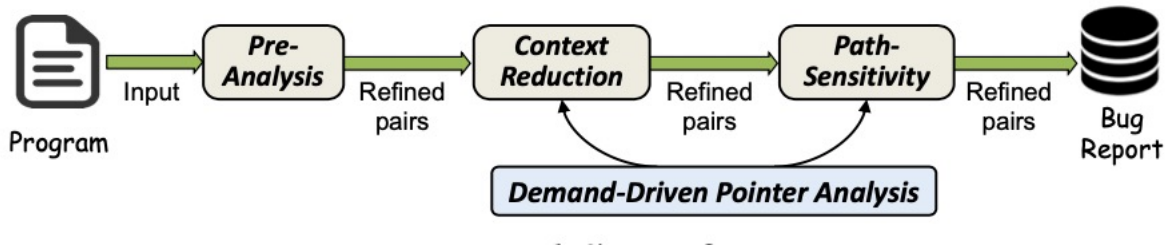
CRED

- Context deletion

(1) Pre analysis to get pointer set

(2) Use set from (1) to delete useless context information

(3) Use path-sensitive to augment calling contexts and improve precision



Buffer over-flow vulnerability

Existing problem

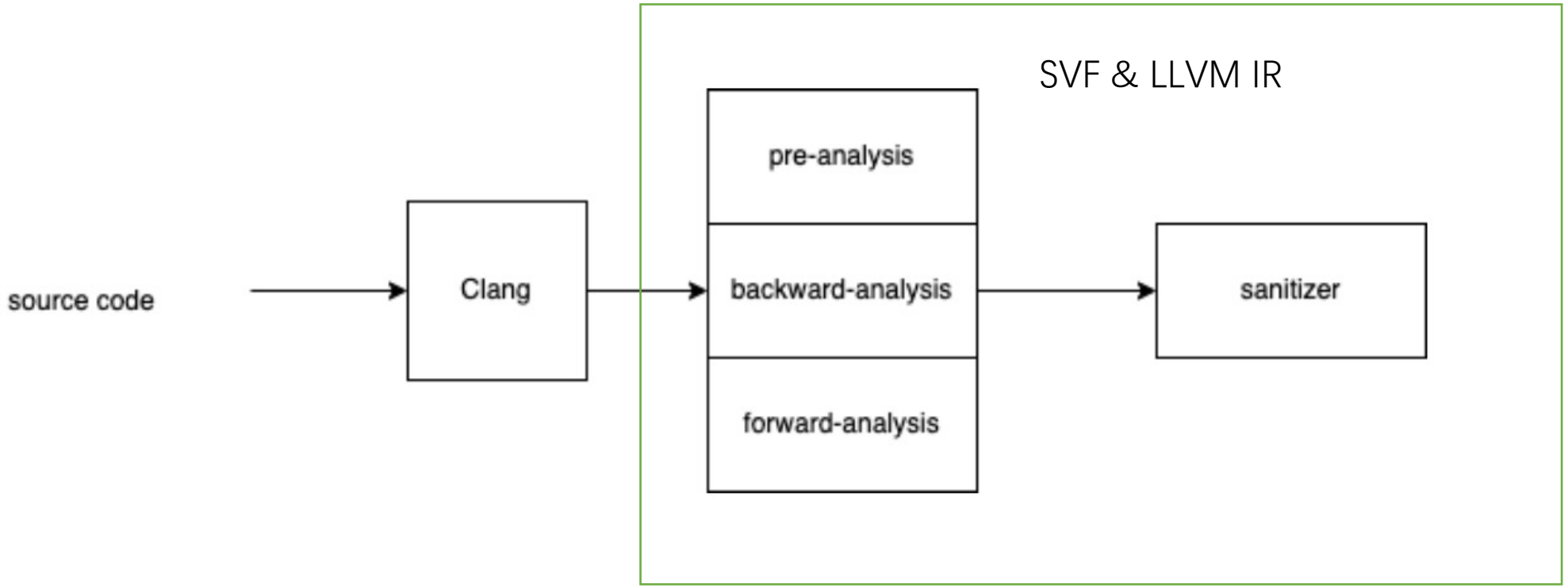
Existing problem

- No scalable on-demand static analysis tool for UAF specifically (pointer set too large/ path too much)

- Many of these tools above are not precise enough

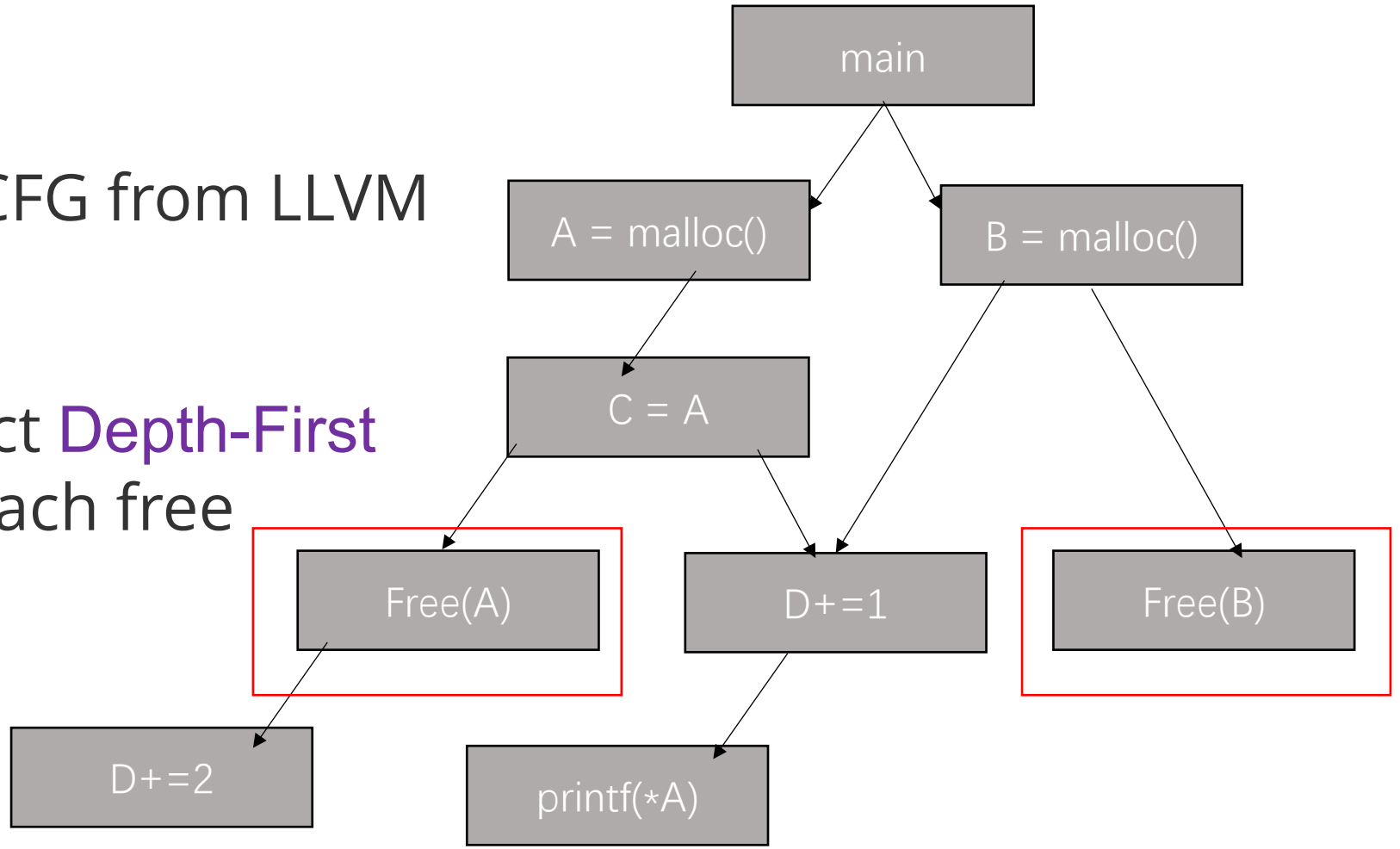
Our approach

Our approach



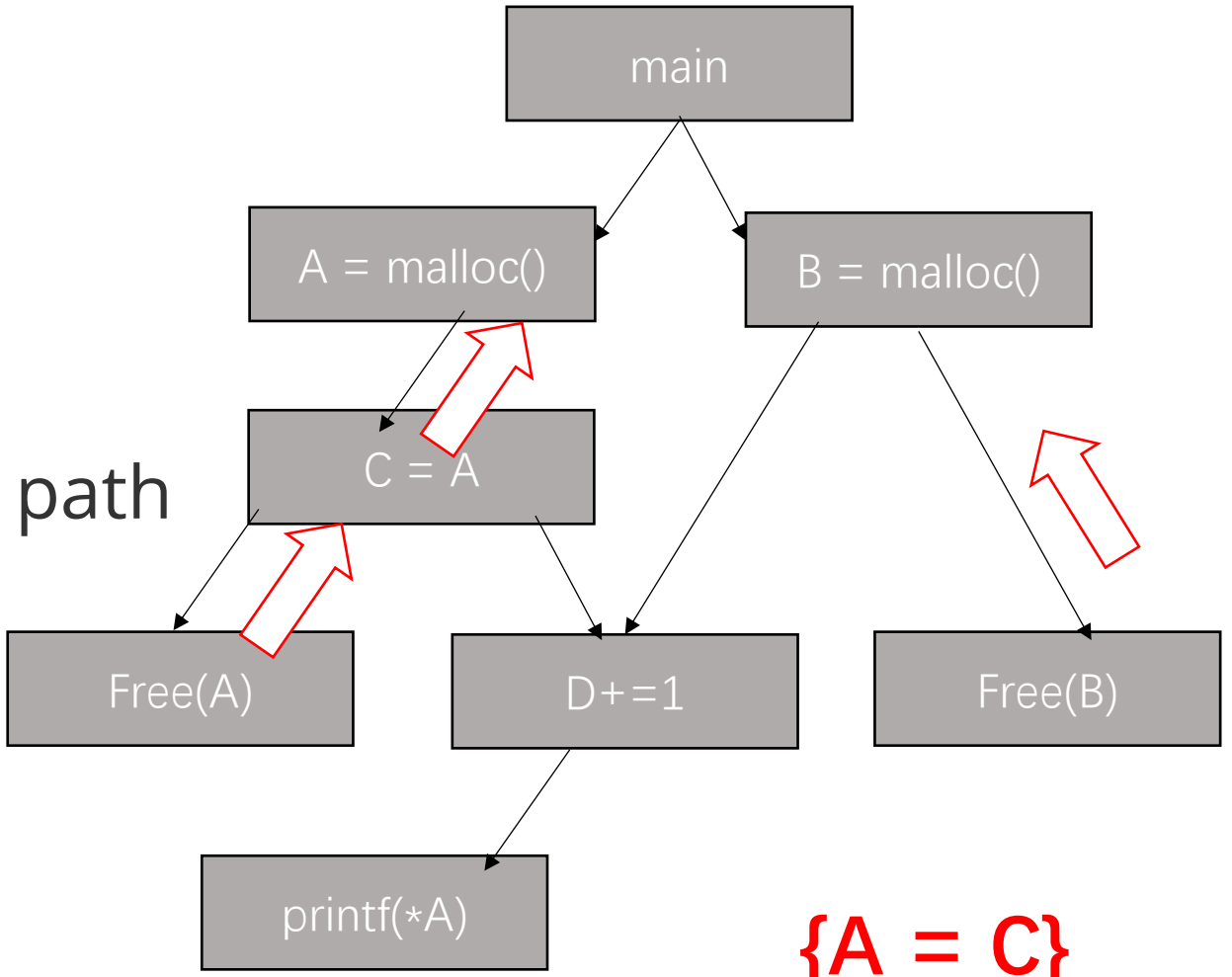
Pre-analysis

- Use SVF to build ICFG from LLVM IR bitcode
- In ICFG, we conduct Depth-First Search to detect each free statement



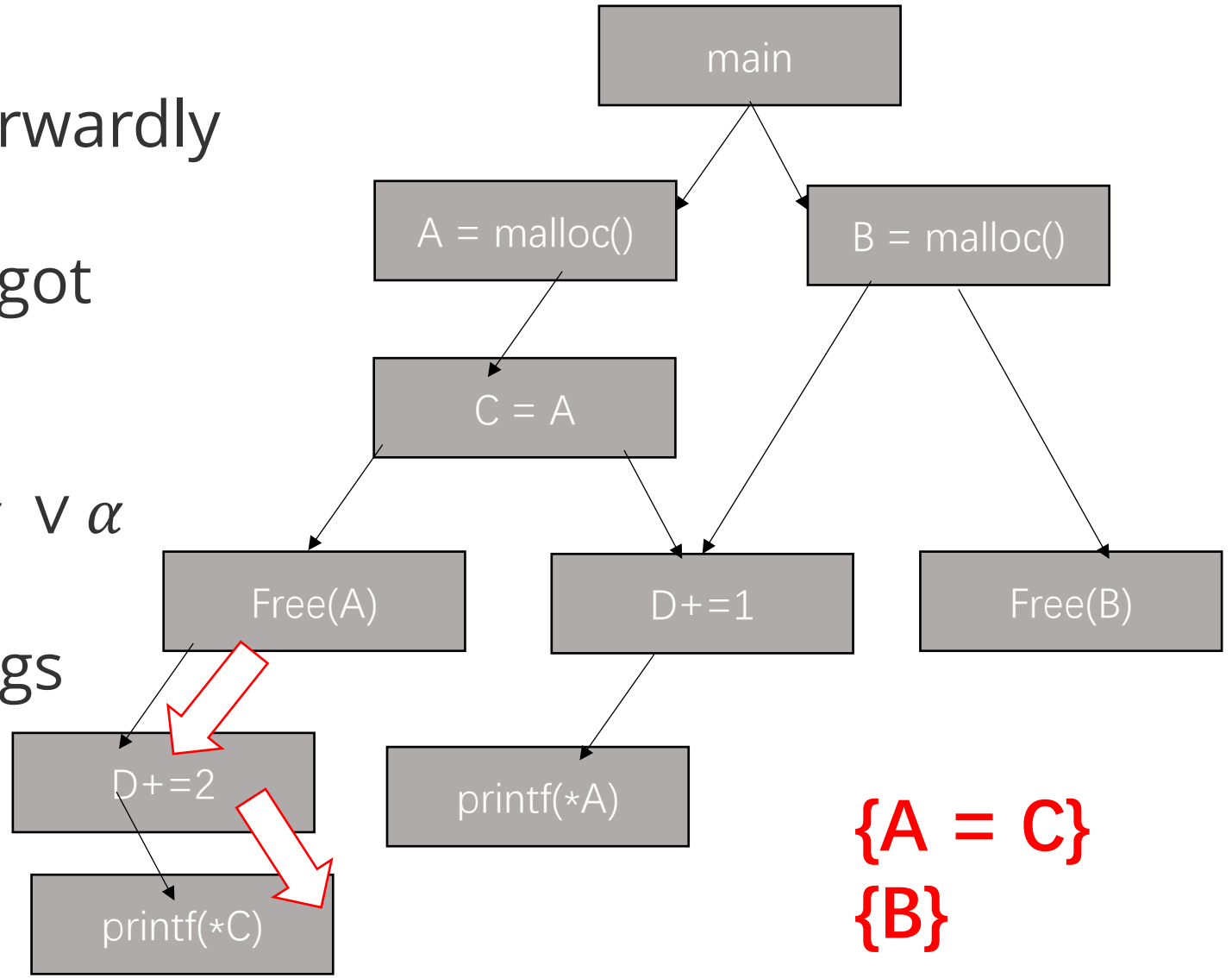
Back-ward analysis

- From each free, search backwardly
- Get point set & solve simple path like $\neg\alpha \vee \alpha$
- Realize backward Andersen pointer analysis



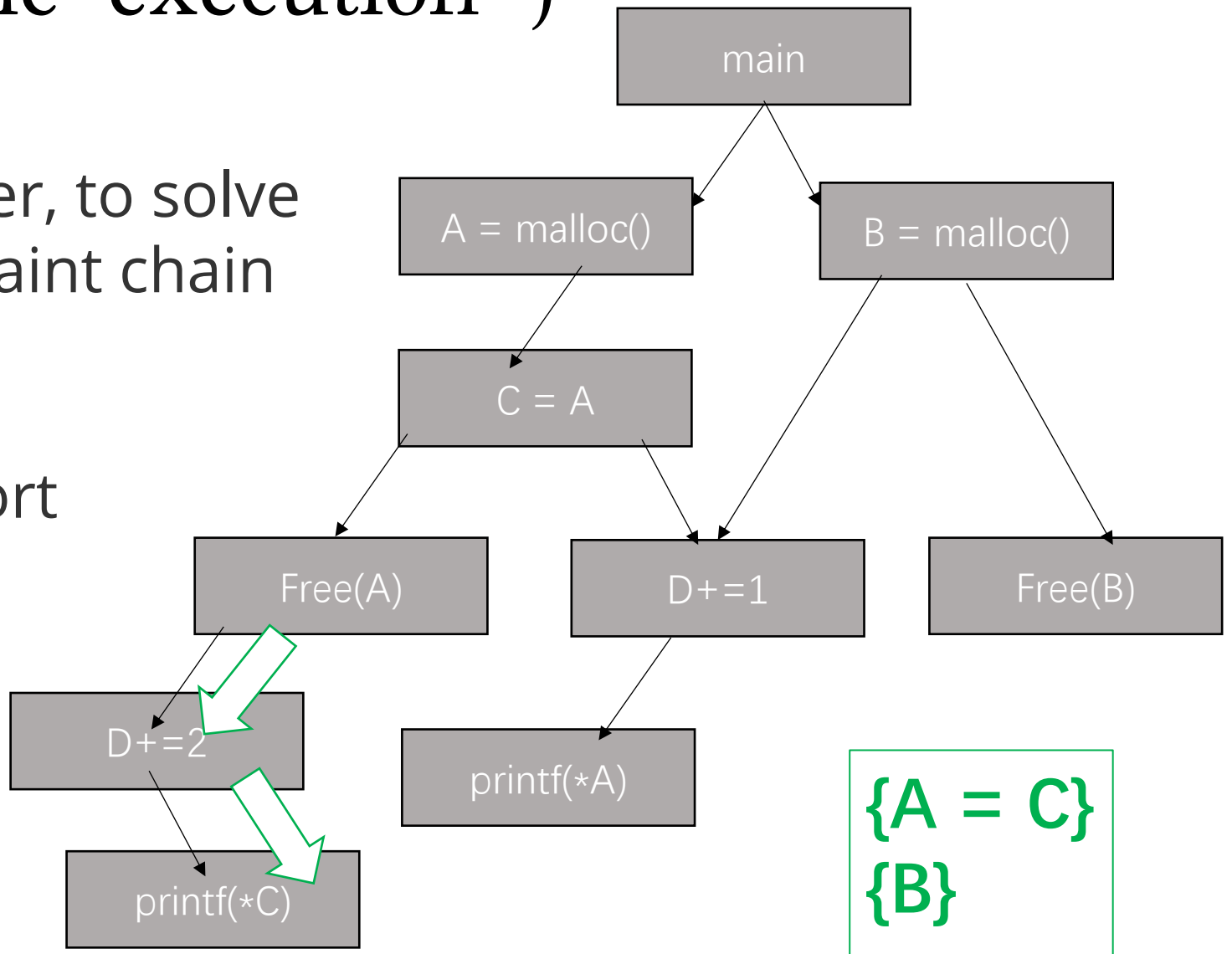
For-ward analysis

- From each free, search forwardly
- Search with the point set got above
- Solve simple path like $\neg\alpha \vee \alpha$
- Report taint chain and bugs



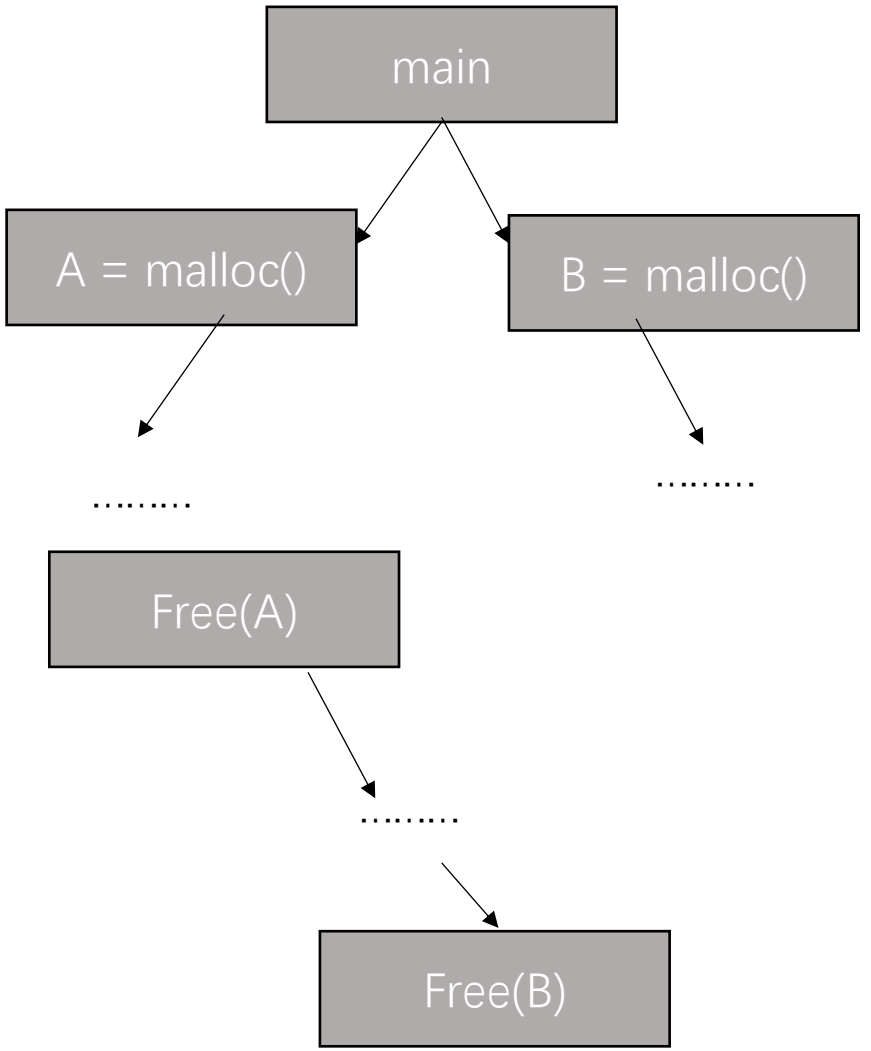
Sanitizer（Symbolic execution）

- Use BDD or SMT solver, to solve difficult path for the taint chain reported above
- Delete infeasible report
- Report again



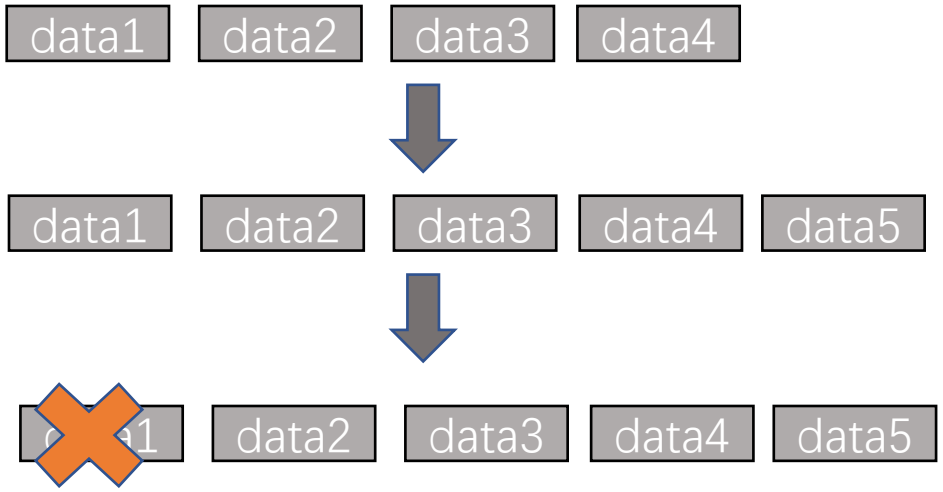
Optimization

- Free(A) and Free(B) share similar path in the ICFG
- After analysis Free(A) backwardly, the path/pointer information we got would be useful when analysis Free(B), so we need to save these data.
- When handling large programs, such data set will get too large



Optimization

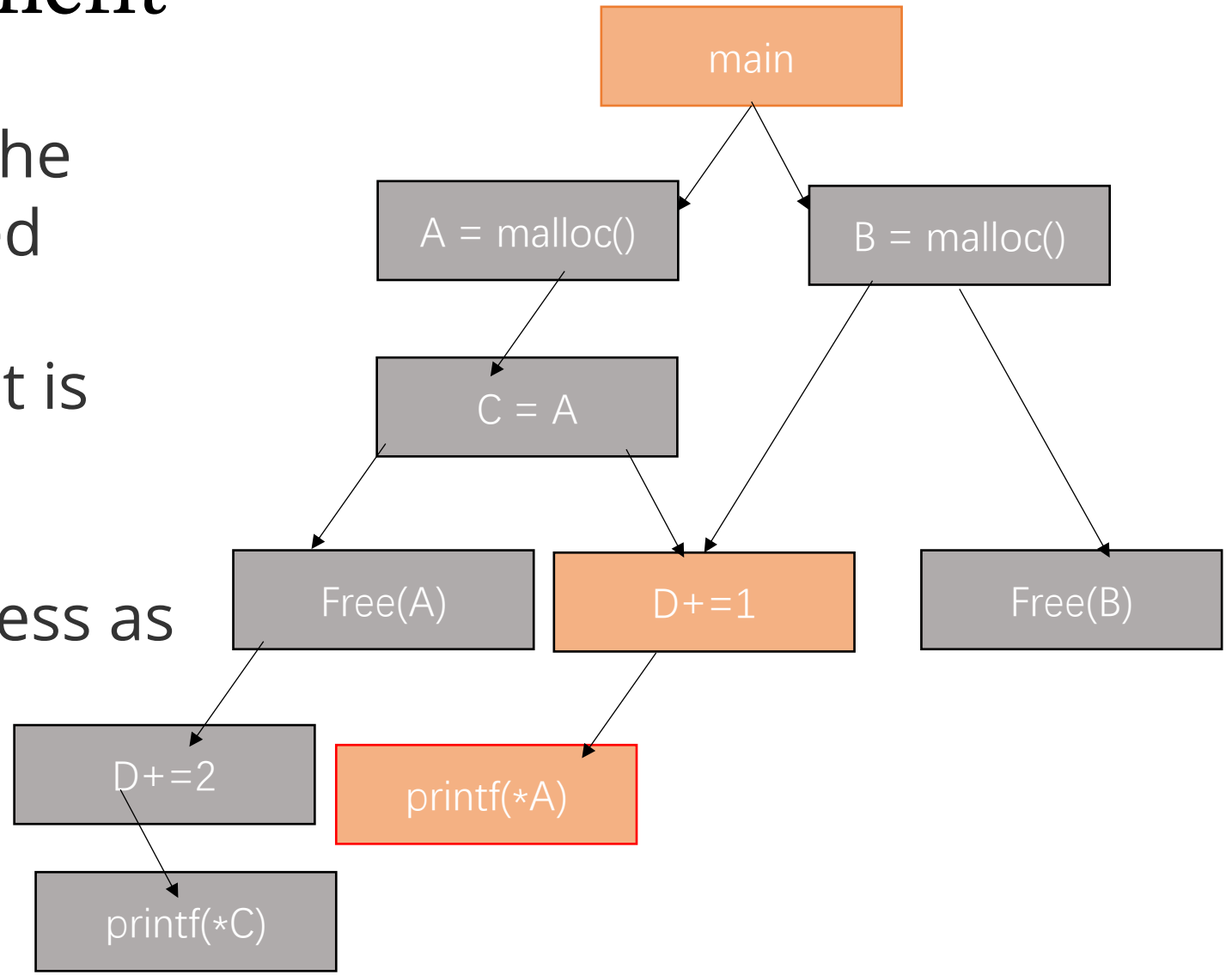
- So, we will set a threshold, if the data set is larger than the threshold, we will delete some of the data based on FIFO



Our advantages

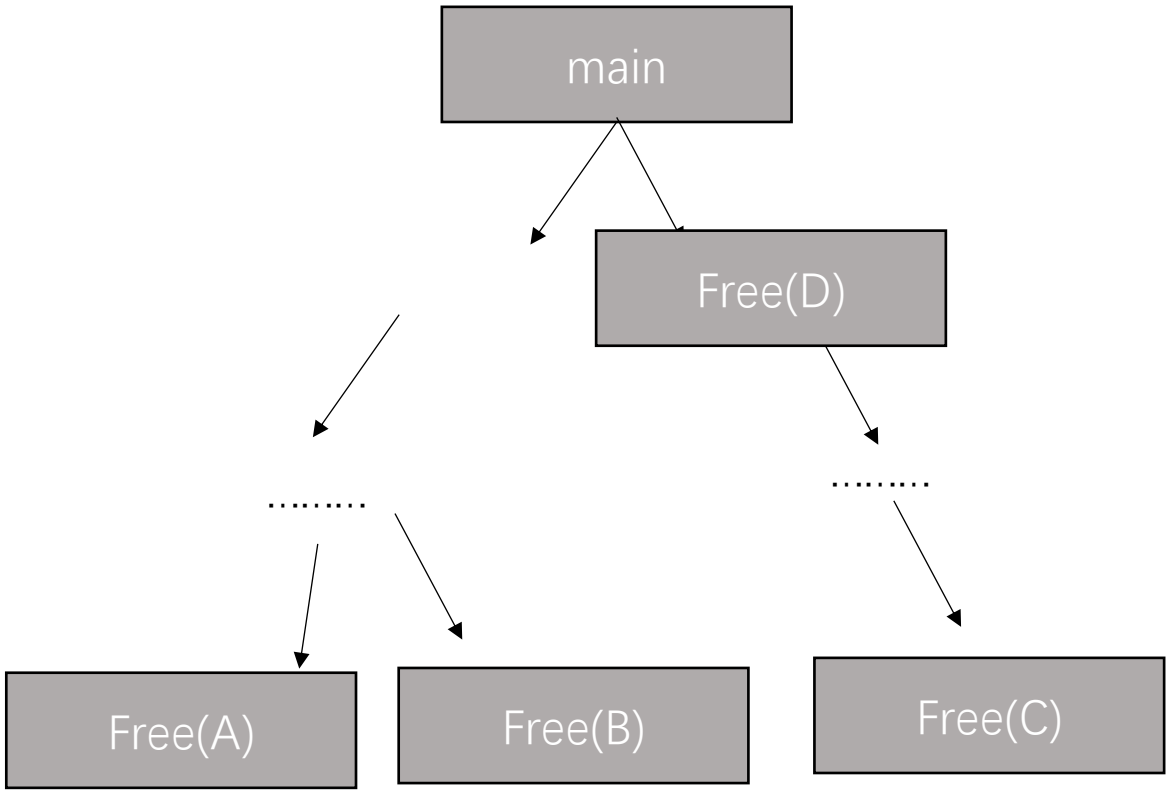
Skip useless statement

- The orange statements in the figure will never be analyzed
- Every variable in pointer set is useful
- Visit each useful nodes as less as possible



Keep scalable data set with almost no overhead

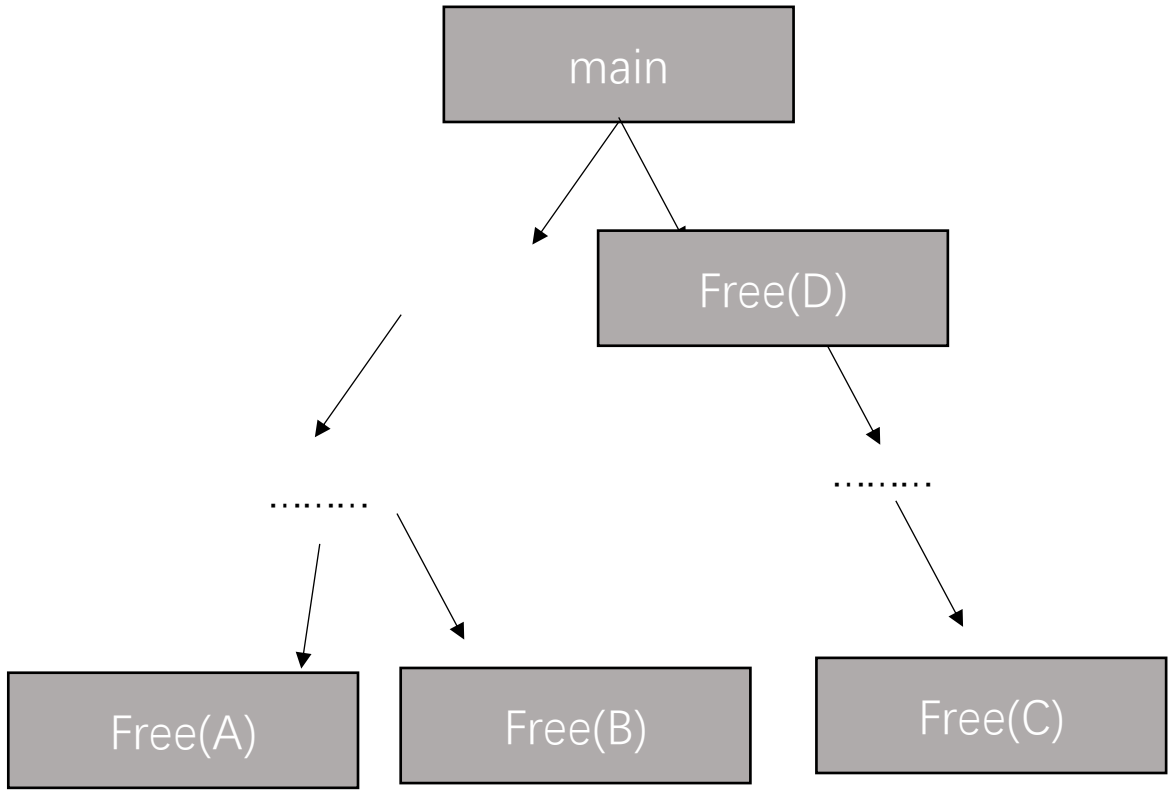
- Delete data based on threshold and FIFO
- It makes sense, as we find the free statement with Depth-First Search, after analyze Free(D), we will automatically start with the other which is most likely to share the same path: Free(C)



Order: free(A) -> free(B) -> free(D)->free(C)

Keep scalable data set with almost no overhead

- When deleting data, we will delete A first, it is less likely for such deletion to effect the analyze of C



Order: free(A) -> free(B) -> free(D)->free(C)

Other advantages

- The Sanitizer with path-sensitive analysis will eliminate false-positive
- Separate Path-sensitive analysis with other analysis to save time, we only conduct time-consuming path analysis for suspect paths.

Thanks