2022 SW 재난센터 여름 정기 워크샵

복잡한 C++ 프로그램 분석을 위한 LLVM IR 분석 - Dynamic object extraction 기법 구현

이아청 박사과정
KAIST 김문주 교수 연구실

1

C++ 동적 분석
(Dynamic object extraction)
이란 무엇인가?

2

Dynamic object extraction의 구현

- Clang AST, LLVM IR 툴 설명 중심

복잡하다!

표준 문서 분량 : 1853쪽

계속된 기능 추가, 더 늘어나는 문서 분량

C++ 버전	표준 분량 (쪽)	증가량
C++11	1338	
C++14	1358	+20 (1.5%)
C++17	1605	+247 (18.2%)
C++20	1853	+248 (15.5%)
C18	520	

C++ 11 발표 : 2011. 8월
GCC C++ 11 구현 완료 : 2013. 3월

```cpp
int (*foo)(int[], int) = +[](int x[], int indx) { return indx[x]; };
```

```cpp
int x = 45;
int * ptr = &(++x); //ok, compile error in C
int * ptr2 = &(x++); // compile error
```

```cpp
struct Foo {
  int func(int and) bitand { return 42; }
  char func(int bitand) and { return 255; }
};

void main() {
  int intvar = 42;
  Foo foo;

  foo.func(42);  // ok
  Foo().func(intvar); //ok
  foo.func(intvar); // compile error
  Foo().func(42); // compile error
}
```

```cpp
template<class T>
struct Y {
  void g(X<T> *p) {
    p->template X<T>::f();
  }
};
```

```cpp
template <typename TF, typename... Ts>
void for_each_arg(TF&& f, Ts&&... xs)
{
    using swallow = int[];
    return (void)swallow{(f(std::forward<Ts>(xs)), 0)...};
}
```

많은 기능

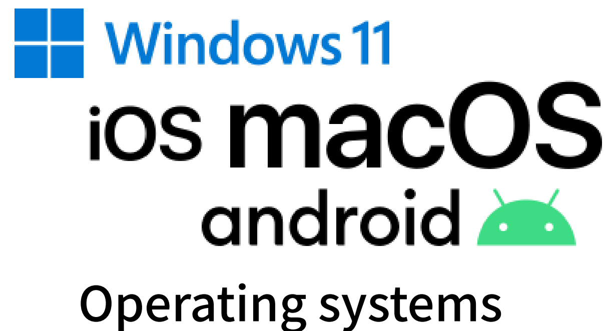
Object-oriented
메모리 관리
STL (Standard Template Library)
Exception
빠른 실행 속도
...

많이 쓴다!

C++ projects

Database engines

Web browsers



Operating systems

Video game engines

하지만 C++에 대한 연구는 부진한 상태!

Google scholar 검색 결과

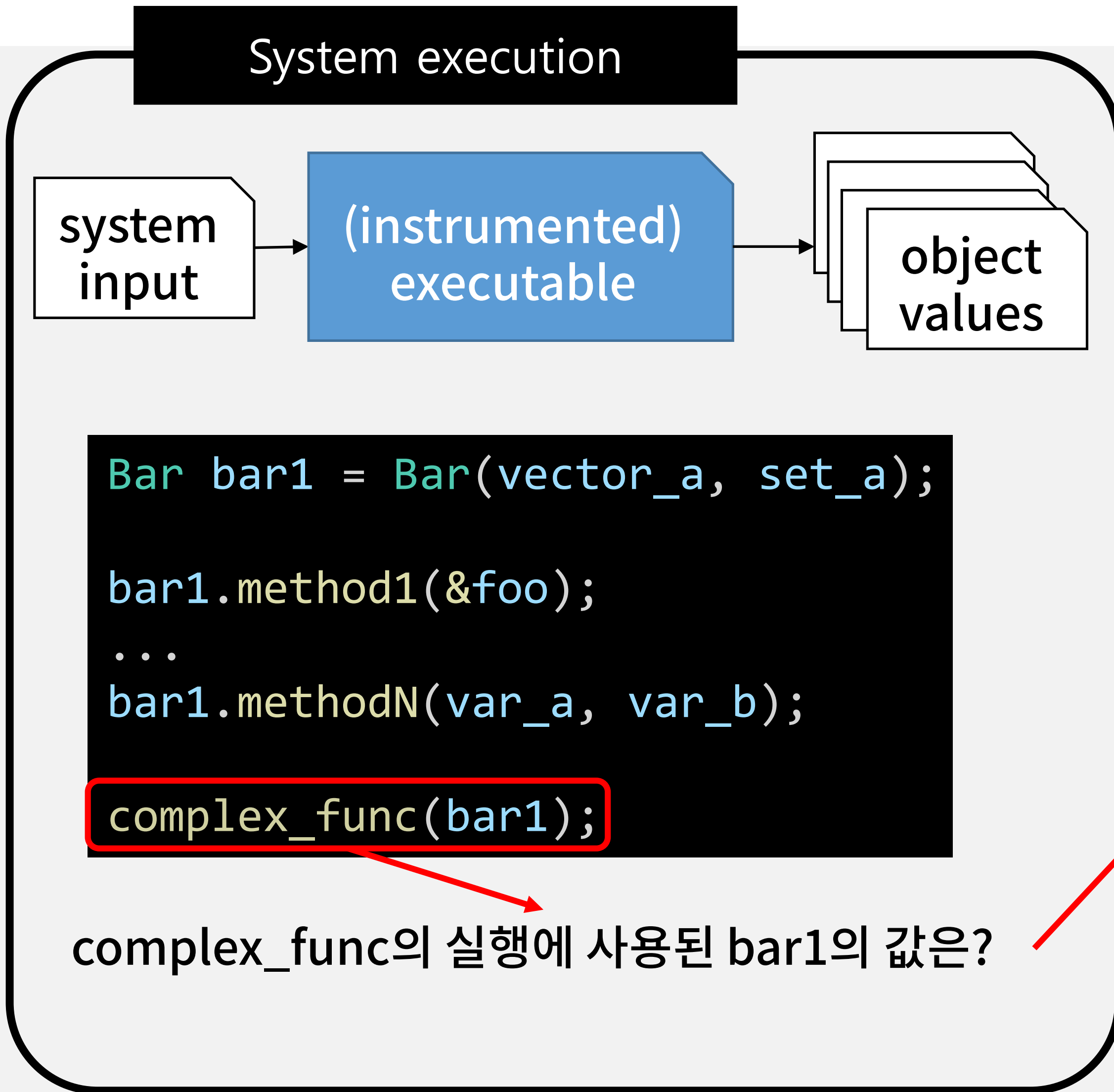
“C++ testing” : 22,900 건
“Java testing” : 1,240,000 건
“C testing” : 6,610,000 건

	Java	C
testing	54배!	289배!
static analysis	37배!	315배!

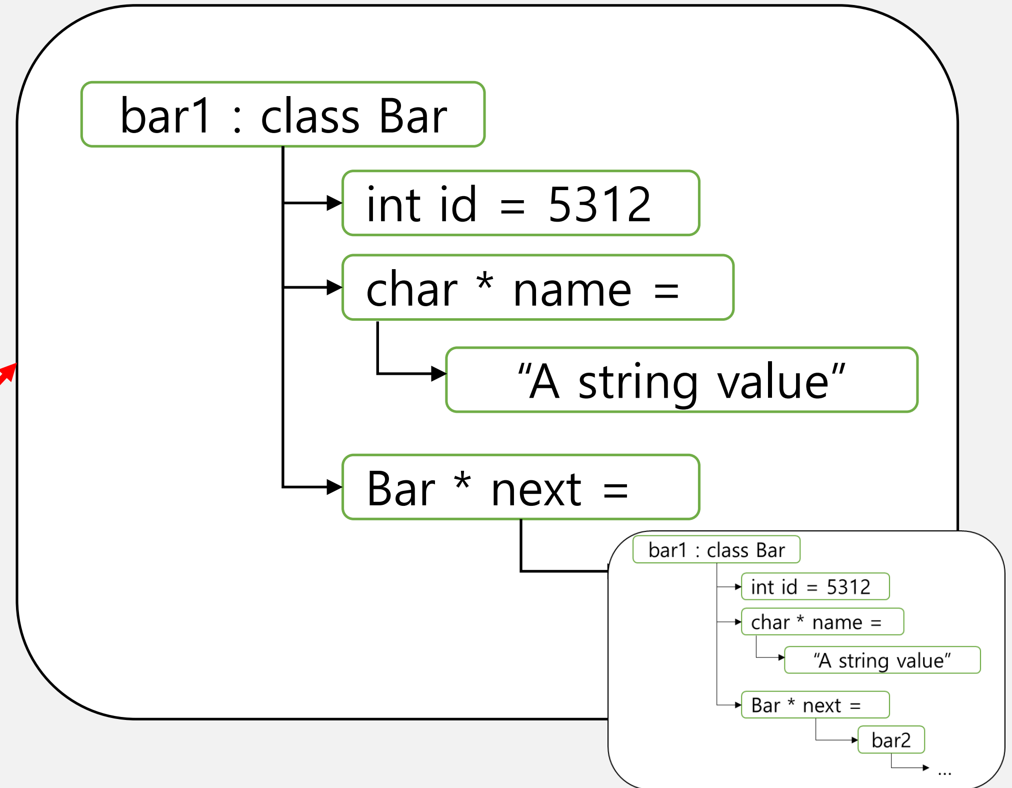
Google scholar 검색 결과

“C++ static analysis” : 15,400 건
“Java static analysis” : 569,000 건
“C static analysis” : 4,850,000 건

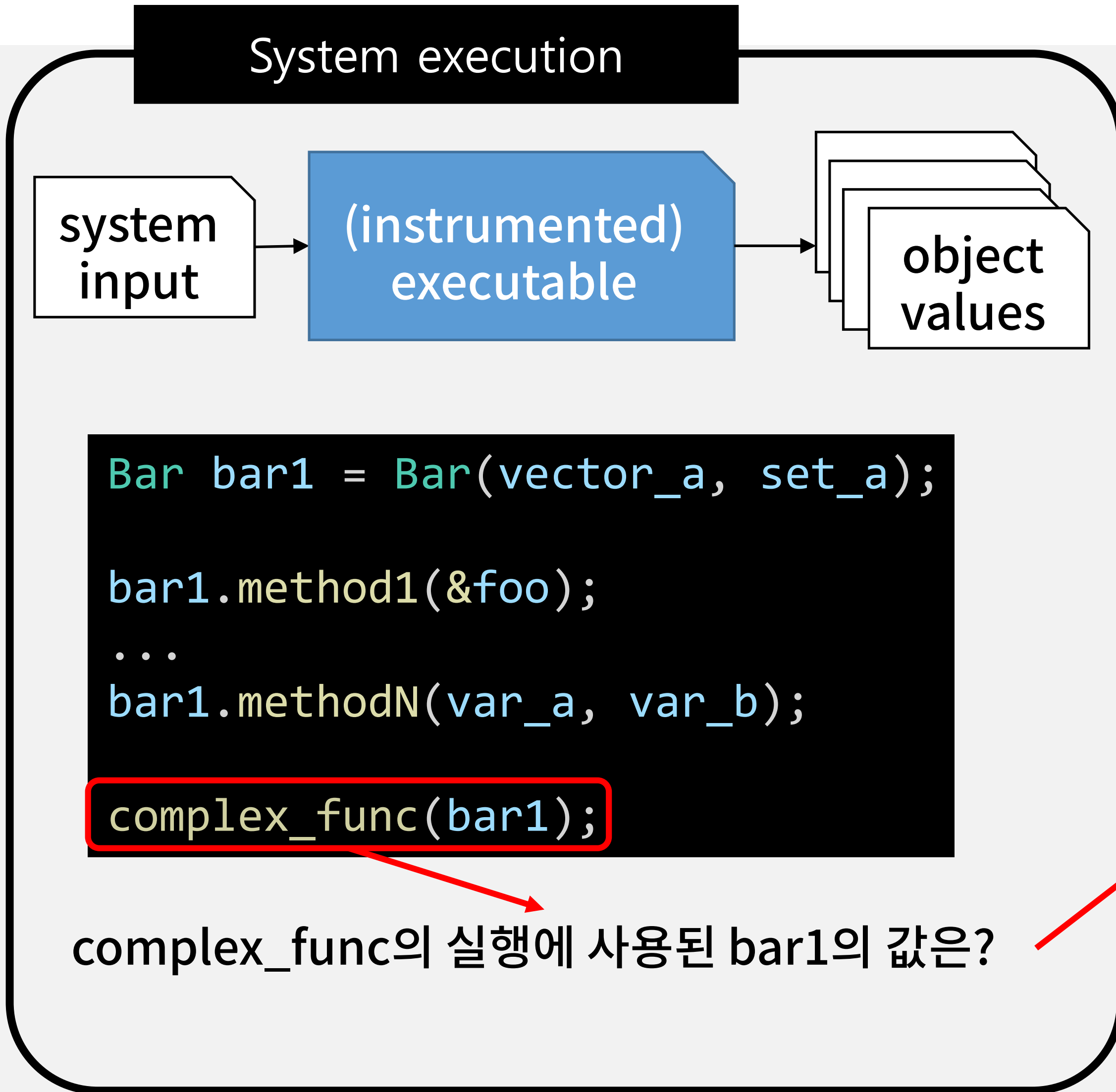
Carving (Dynamic object extraction) & Replaying



실행 도중 object의 값을 추출 및 재연하는 기술 (profiling)

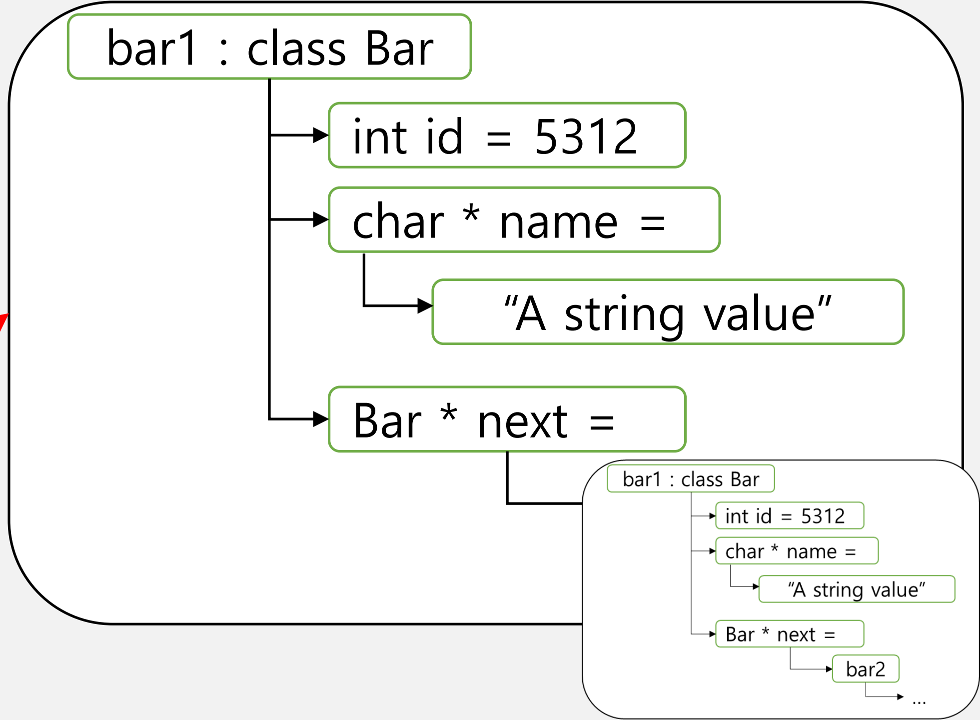


Carving (Dynamic object extraction) & Replaying

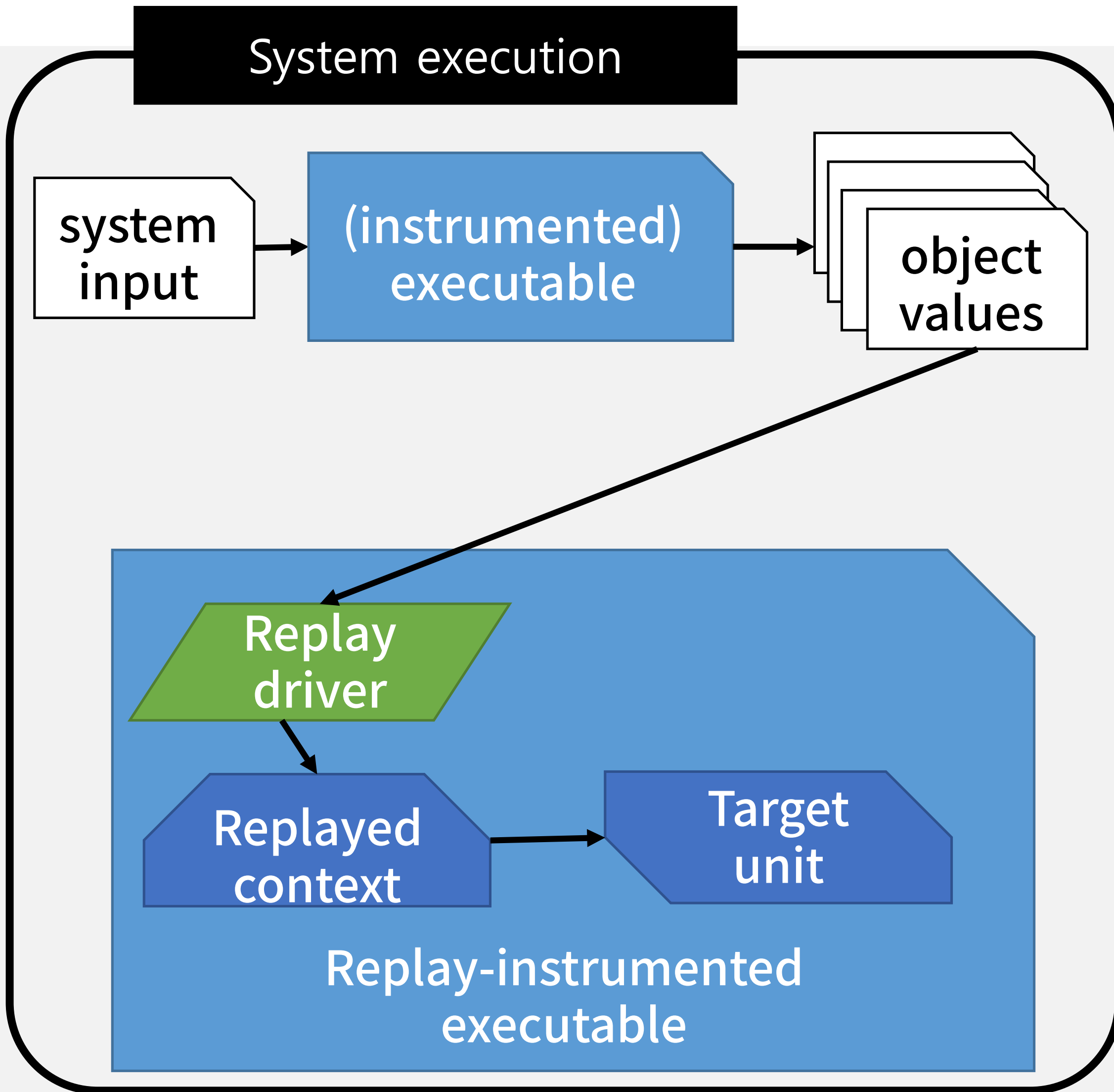


실행 도중 object의 값을 추출 및 재연하는 기술

=> 실행 상태의 메모리 상태를 재연 가능한 수준으로 추출하는 기술



Carving (Dynamic object extraction) & Replaying

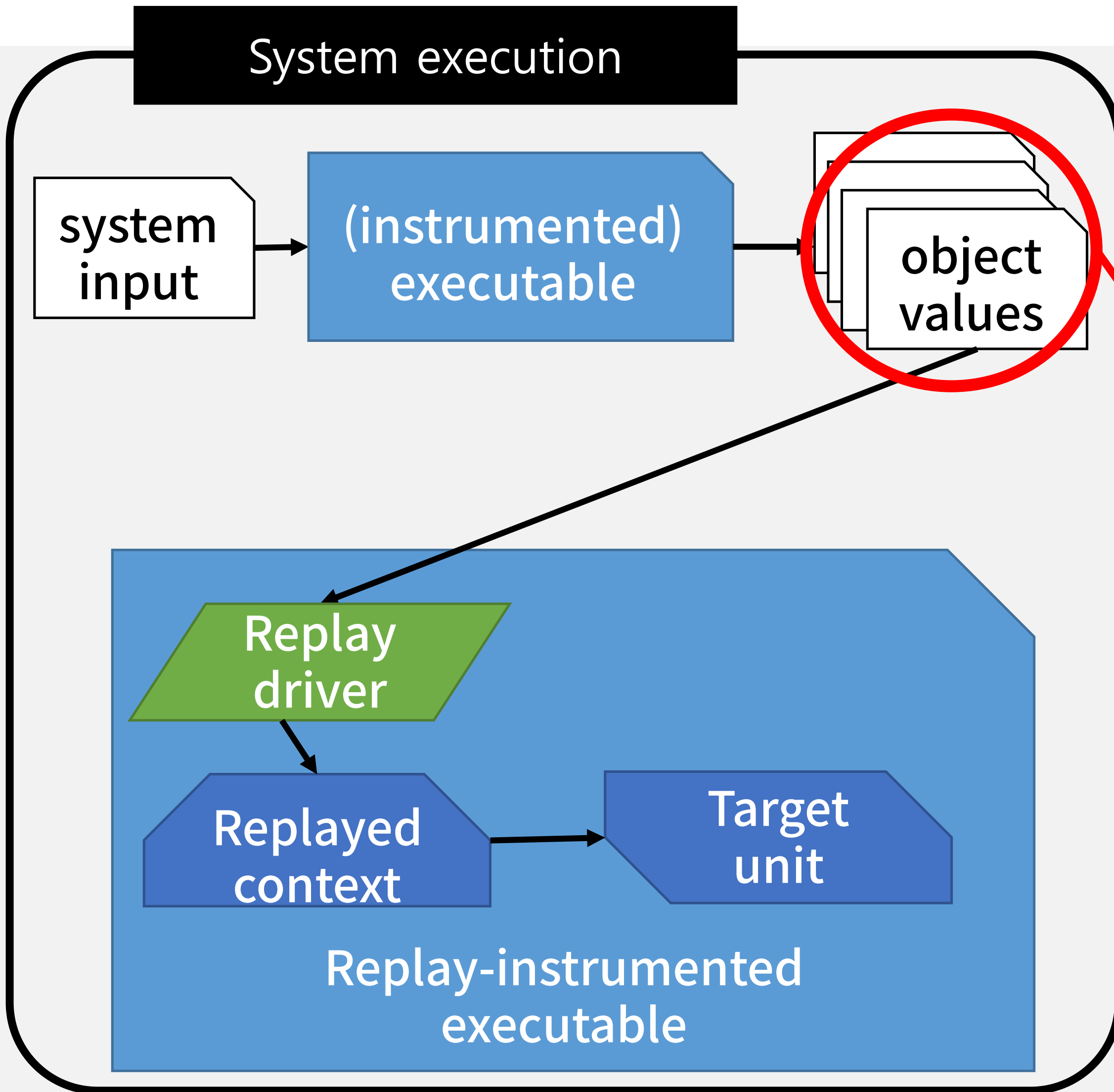


실행 도중 object의 값을 추출 및 재현하는 기술

=> 실행 상태의 메모리 상태를 (유닛 레벨에서) 재연 가능한 수준으로 추출하는 기술

재연시에는 재연용 driver를 이용하여, 추출된 값을 받아 다시 실행하고자 하는 Memory state (Context)를 다시 복구하여 다시 실행할 수 있도록 함.

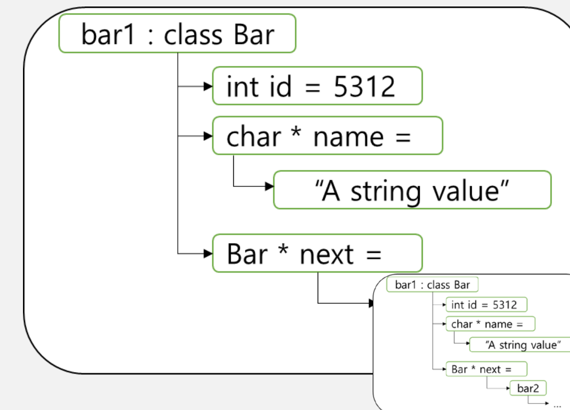
Carving (Dynamic object extraction) & Replaying 예시



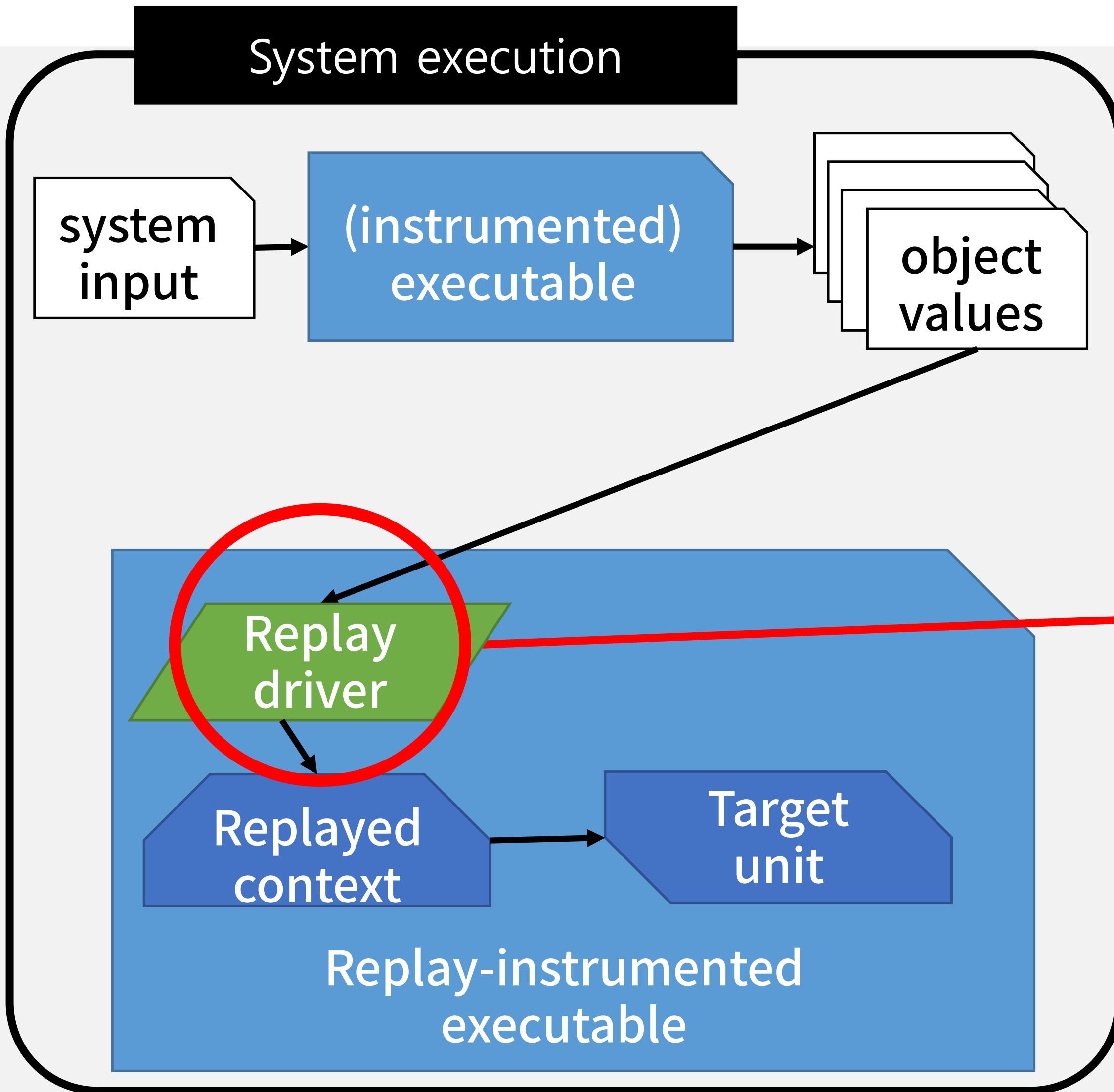

```
complex_func(Bar bar1);
```

```
class Bar {
  int id;
  char * name;
  Bar * next;
};
```

- Complex_func의 파라미터 (bar1)
- 파라미터가 가리키는 objects (name, next)
- 해당 함수가 사용하는 전역 변수



Carving (Dynamic object extraction) & Replaying 예시




```
complex_func(Bar bar1);
```

```
class Bar {
  int id;
  char * name;
  Bar * next;
};
```

```
void complex_func_driver() {
  context * ctx = read_carved_input();
  Bar bar;
  bar.id = ctx->bar1.id;
  bar.name = ctx->bar1.name;
  bar.next = ctx->bar1.next;

  set_global_var(ctx);

  complex_func(bar);
}
```

Carving (Dynamic object extraction) – application (To Do)

- Valid한 object을 매우 많이 얻을 수 있음.

Unit testing, API testing

(Database program)

```
struct sqlite3 {
  sqlite3_vfs *pVfs;
  struct Vdbe *pVdbe;
  CollSeq *pDfltColl;
  sqlite3_mutex *mutex;
  Db *aDb;
  //... 88 fields
}
```

```
sqlite3 * db = sqlite3_open();

sqlite3_prepare(db, "prepare", 2048);
sqlite3_db_config(db, 1, 2);
sqlite3_exec(db, "Insert", insert_call, context());
```

이 db object은 각 API가 제대로 작동할 수 있게 valid하면서도, test가 가능하도록 다양하게 만들 수 있어야 함. (false alarm 방지)

Test case reduction/priortization

Fault localization

System test generation

Carving (Dynamic object extraction) – application (To Do)

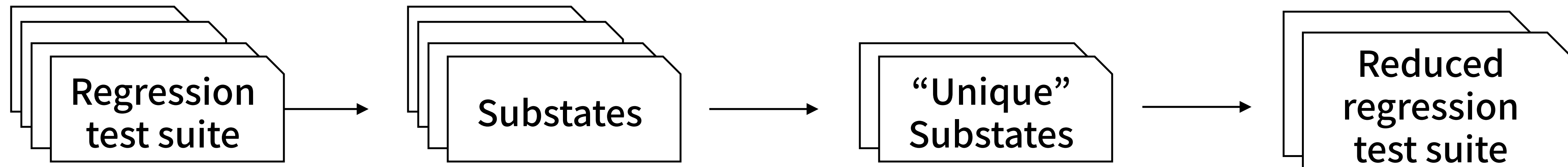
- Valid한 object을 매우 많이 얻을 수 있음.

Unit testing, API testing

Test case reduction/prioritization

Carving은 테스트 실행의 memory state를 추출하는 과정,
각 테스트의 실행을 라인/분기 레벨보다 더 세밀하게 비교가 가능하다.

Carving이 비교적 간단한 Java의 경우,
각 테스트의 substate를 비교하여 테스트를 reduction하는 방법으로
Regression test suite의 효율성 (더 많은 버그, 낮은 테스트 수)을 높인 사례가 존재[Trad, Chadi et al. 2018].



Fault localization

System test generation

Abou Assi, Rawad & Masri, Wes & Trad, Chadi. (2018). Substate Profiling for Effective Test Suite Reduction. 123-134. 10.1109/ISSRE.2018.00023.

Carving (Dynamic object extraction) – application (To Do)

- Valid한 object을 매우 많이 얻을 수 있음.

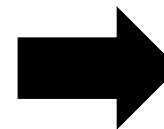
Unit testing, API testing

Test case reduction/priortization

Fault localization

SBFL (Statistical based Fault Localization)은 커버리지를 기반으로
성공한 테스트와 실패한 테스트를 비교하여 Fault가 발생할 가능성이 높은 라인을 도출
(Ex, 여러 실패한 테스트가가 함께 실행한 라인이 수상하다)

	Success tc1	Success tc2	Fail tc1
Line 1	실행	실행	실행하지 않음
Line 2	실행	실행하지 않음	실행



Line1 보다 Line 2에서
Fault가 발생했을 가능성이 높다.

Memory state를 비교하여
실행 결과 성공한 테스트여도, 실패한 테스트와 비슷한 테스트라면,
해당 테스트가 수행한 라인에서도 Fault가 발생했을 가능성이 높지 않을까 추측.

System test generation

Carving (Dynamic object extraction) – application (To Do)

- Valid한 object을 매우 많이 얻을 수 있음.

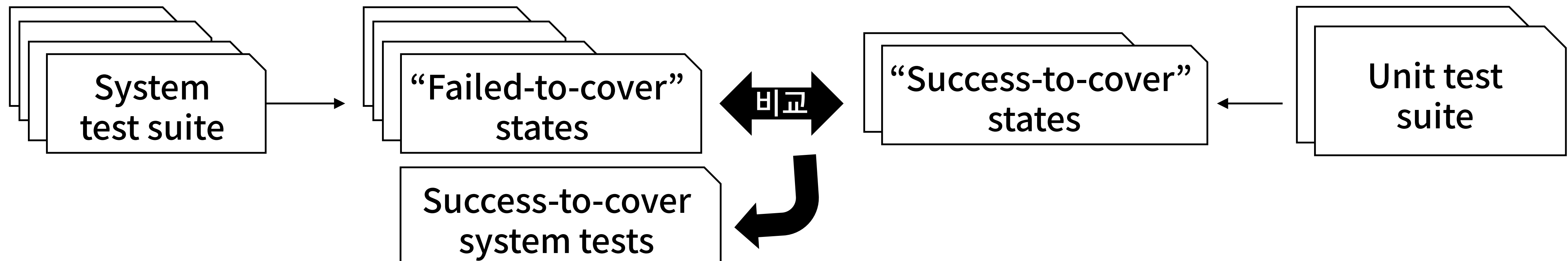
Unit testing, API testing

Test case reduction/priortization

Fault localization

System test generation

시스템 레벨에서 특정 분기에 도달하는 테스트를 만들 수 없었지만,
유닛 레벨에서는 가능했다면, 도달했을 때와 못 했을 때의 Memory state를 비교하여,
어떤 정보가 해당 분기에 도달하는데 중요하게 작동했는지 힌트를 얻을 수 있음.
이를 이용하여 시스템 레벨 테스트를 만드는데 사용할 수 있을 것으로 추측.



1

C++ 동적 분석
(Dynamic object extraction)
이란 무엇인가?

2

Dynamic object extraction의 구현

- Clang AST, LLVM IR 툴 설명 중심

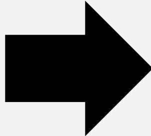
Carving 구현

LLVM IR이 무엇인가?
-> 다음 슬라이드

총 2천 줄 정도의 C++ 코드를 작성,
Insturment 하고자 하는 타겟 프로그램의 IR (Intermediate Representation)을 자동으로 수정하여
원하는 Object의 정보를 파일의 형태로 출력

원본 코드

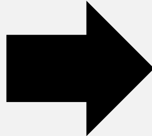
```
define dso_local i32 @foo(%struct._a* %0) #0 !dbg !9 {
  %14 = alloca %struct._a*, align 8
  %15 = bitcast %struct._a** %14 to i8*
  store %struct._a* %0, %struct._a** %14, align 8
  call void @llvm.dbg.declare(metadata %struct._a** %14, metada
  %16 = load %struct._a*, %struct._a** %14, align 8, !dbg !22
  %17 = getelementptr inbounds %struct._a, %struct._a* %16, i32
  %18 = load i8, i8* %17, align 4, !dbg !23
  %19 = sext i8 %18 to i32, !dbg !22
  ret i32 %19, !dbg !24
}
```

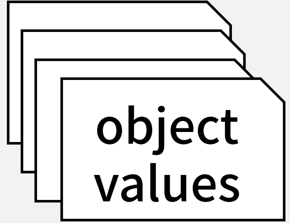


Instrument 된 코드

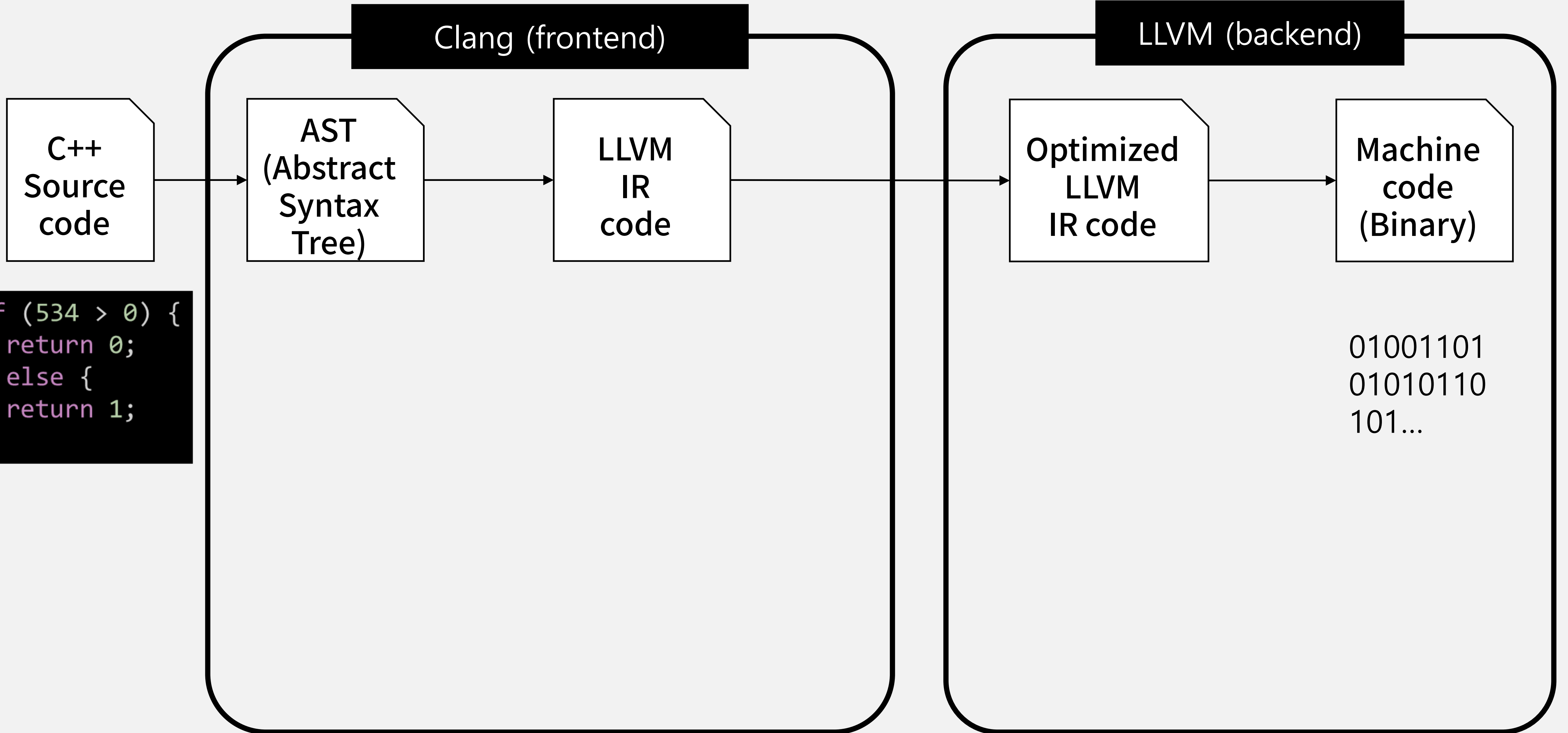
```
define dso_local i32 @foo(%struct._a* %0) #0 !dbg !9 {
  call void @_Z22__carv_func_call_probei(i32 0)
  call void @_Z16__carv_name_pushPc(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @0, i32 0, i32 0))
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = bitcast %struct._a* %0 to i8*
  %5 = call i32 @_Z12Carv_pointerPv(i8* %4)
  %6 = sdiv i32 %5, 8
  %7 = icmp eq i32 %6, 0
  br i1 %7, label %13, label %8

8:                                                ; preds = %8, %1
  %9 = phi i32 [ %3, %1 ], [ %11, %8 ]
  %10 = getelementptr %struct._a, %struct._a* %0, i32 %9
  call void @_Z22__carv_ptr_name_updatei(i32 %9)
  call void @__Carv__struct__a(%struct._a* %10)
  %11 = add i32 %9, 1
  call void @_Z15__carv_name_popv()
  %12 = icmp slt i32 %11, %6
  br i1 %12, label %8, label %13

13:                                               ; preds = %8, %1
  call void @_Z15__carv_name_popv()
  call void @_Z23__update_carved_ptr_idxv()
  %14 = alloca %struct._a*, align 8
  %15 = bitcast %struct._a** %14 to i8*
  call void @_Z21__mem_allocated_probePvi(i8* %15, i32 8)
  store %struct._a* %0, %struct._a** %14, align 8
  call void @llvm.dbg.declare(metadata %struct._a** %14, metadata !20, metadata !DIExpression()), !dbg !21
  %16 = load %struct._a*, %struct._a** %14, align 8, !dbg !22
  %17 = getelementptr inbounds %struct._a, %struct._a* %16, i32 0, i32 1, !dbg !23
  %18 = load i8, i8* %17, align 4, !dbg !23
  %19 = sext i8 %18 to i32, !dbg !22
  call void @_Z16__carv_name_pushPc(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @3, i32 0, i32 0)), !dbg !24
  call void @_Z8Carv_inti(i32 %19), !dbg !24
  call void @_Z15__carv_name_popv(), !dbg !24
  call void @_Z21__carv_func_ret_probePci(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @4, i32 0, i32 0), i32 0), !dbg !24
  %20 = bitcast %struct._a** %14 to i8*, !dbg !24
  call void @_Z28__remove_mem_allocated_probePv(i8* %20), !dbg !24
  ret i32 %19, !dbg !24
}
```

(회색을 제외한 부분이 모두 삽입된 IR 코드)

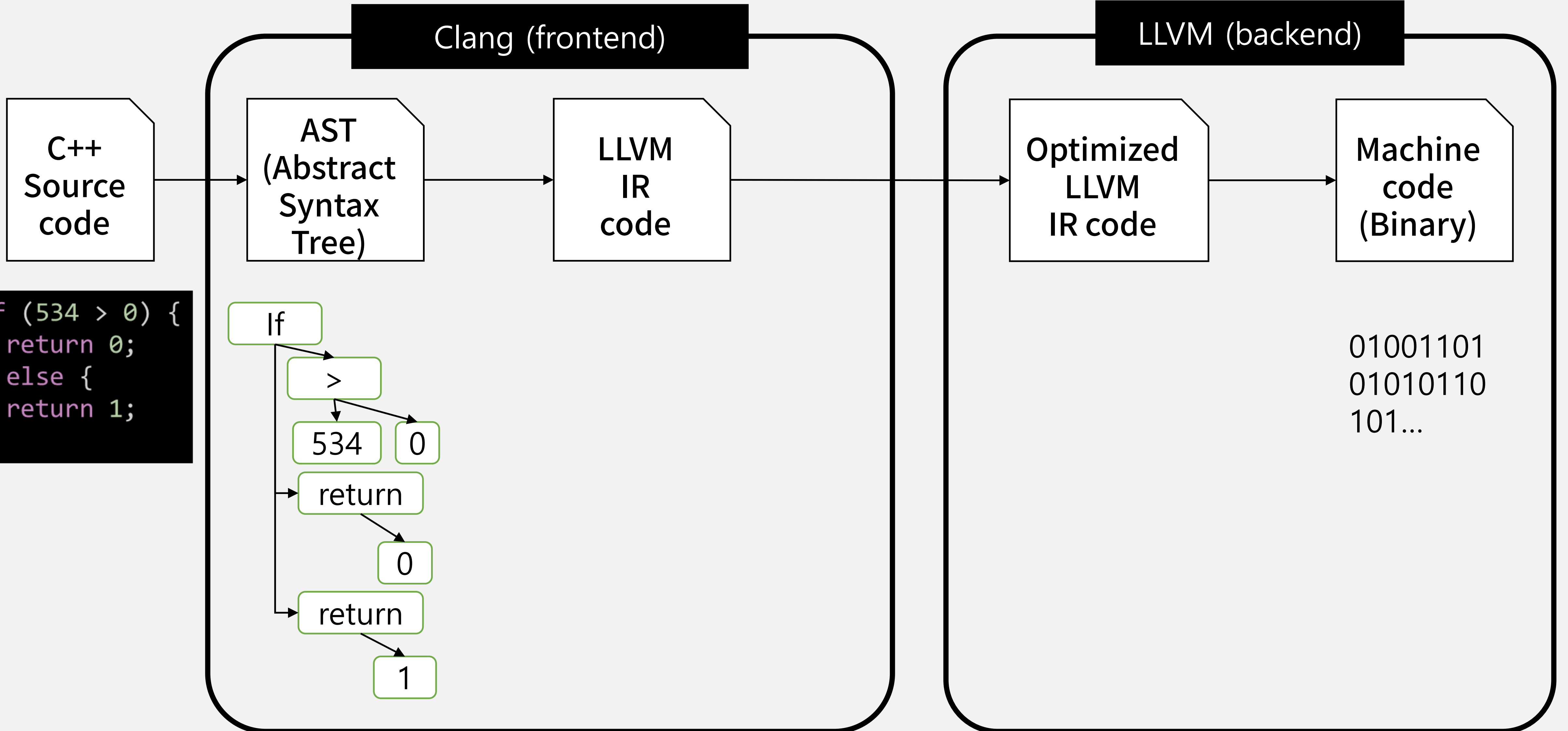




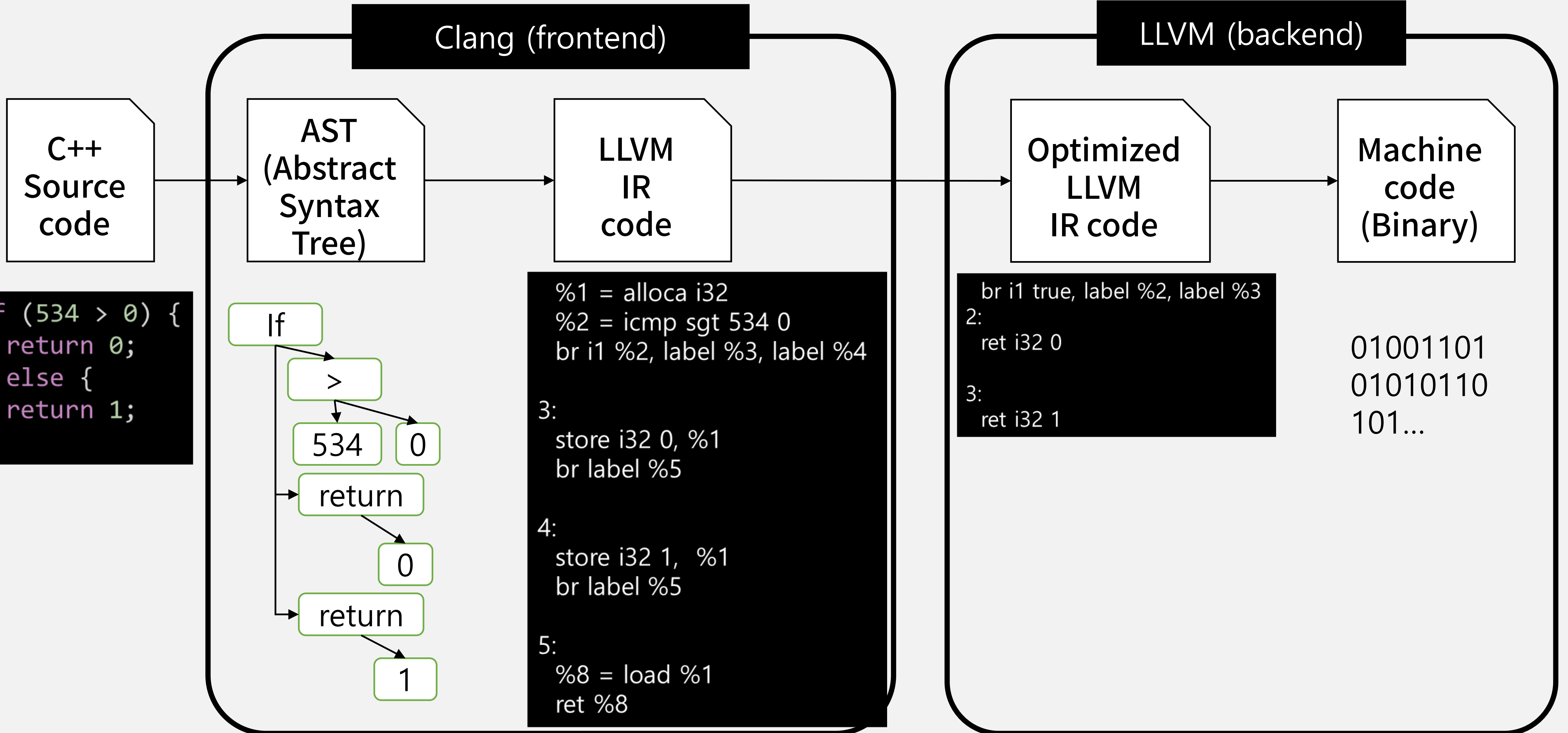
Clang/LLVM C/C++ Compilation



Clang/LLVM C/C++ Compilation



Clang/LLVM C/C++ Compilation



왜 (Clang AST 가 아닌) LLVM IR인가

AST는 원본 소스 코드에 비슷한 형태이지만, IR은 머신 코드에 비슷한 형태.

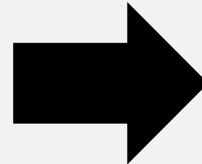
따라서, AST에서 자동 수정을 하기 위해서는 C++ 소스코드의
다양한 Syntax case에 대해 모두 구현작업이 필요하다.

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

이와 같은 Rectangle object를 추출한 후, 다시 재현을 하기 위해서는,
재현을 위한 드라이버 코드 작성이 필요.

Rectangle 정의

```
class Rectangle {
public:
  int width;
  int area;
};
```



재현용 드라이버 코드

```
Rectangle Rectangle_replay() {
  context * ctx = read_input();
  Rectangle rect;
  rect.width = ctx->rect.width;
  rect.area = ctx->rect.area;

  return rect;
}
```

단순히 Rectangle을 만들어서, 이전에 추출한 width,area 값을 넣고,
반환해주는 코드.

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 1. const qualifier

Rectangle 정의

```
class Rectangle {
public:
  int width;
  const int area;
};
```

재현용 드라이버 코드

```
Rectangle Rectangle_replay() {
  context * ctx = read_input();
  Rectangle rect;
  rect.width = ctx->rect.width;
  rect.area = ctx->rect.area;

  return rect;
}
```

Compiler error!

Const를 고려하여 다르게 구현하여야 함.

int const, int * const,
const int *, int * const *, int ** const
int const [10], const int (*) (int, int) 등

반면 IR에서는 const가 모두 제거됨.
자유롭게 읽고 쓰는 것이 가능.

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay

(non-template) add 정의

```
int add(int a, int b) {
 return a + b;
}
```

재현용 드라이버 코드

```
int add_replay() {
  context * ctx = read_input();
  int param1 = ctx->param1;
  int param2 = ctx->param2;
  add(param1, param2);
}
```

단순히 2개의 int를 가지고 와서 add를 실행

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay

add 정의

```
template<typename T>
T add(T a, T b) {
 return a + b;
}
```

재현용 드라이버 코드

```
int add_replay() {
  context * ctx = read_input();
  T param1 = ctx->param1;
  T param2 = ctx->param2;
  add(param1, param2);
}
```

추출하거나, 재현할 때,
각 add함수가 어떤 타입으로
사용되었는지 알아야
정확하게 추출 및 재현을 할 수 있음.

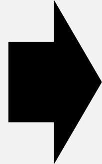
소스 코드에서 add가 무슨 타입으로
사용되는지 알아내야 한다.

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay – 사용한 타입 찾기

add 정의

```
template<typename T>
T add(T a, T b) {
 return a + b;
}
```



add의 사용처는 단순 실행 코드 내에 있을 수도,

```
int arr[24];
c = add(arr[0], arr[3]);
```

모든 실행 코드 탐색 필요

특정 변수 내부에 있을 수도,

```
void * funcs[] = {
  (void *) add<int>,
  (void *) add<short> };
```

모든 Initializer 탐색 필요

다른 template 내부에 있을 수도 있으며,

```
template<typename A>
A add2(A a, A b) {
  return add<A>(a, b);
}
```

(A에 따라 T가 결정)
Recursive하게 탐색 필요

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay – 사용한 타입 찾기

add의 정의는 클래스 정의 내부에 있을 수도 있기 때문에,
클래스가 어떻게 instance화 되느냐에 따라 add의 타입이 정의된다.

```
template <typename A, typename T>
class Shape {
public:
  Shape(A a) { area = a; }
  A add(A a, A b) { return a + b;}
  A area;
  T width;
};
```

마찬가지로 각 클래스가 어떻게 사용되는지도 추적이 필요하다.

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay – 사용한 타입 찾기

Shape 정의

```
template <typename A, typename T>
class Shape {
public:
  Shape(A a) { area = a; }
  A add(A a, A b) { return a + b;}
  A area;
  T width;
};
```

shape의 사용처는 다른 상속 클래스일 수도 있고,

```
class Rectangle : public Shape<B,Q> {
  ...
};
```

shape의 사용처는 다른 friend 클래스일 수도 있고,

```
class Earth {
  friend class Shape<C,F> ;
  ...
};
```

변수 선언, 타입 선언 등에 계속 사용 된다.
(모든 케이스를 탐색하도록 구현해야 한다.)

```
Shape<int, char> b;
```

```
std::vector<Shape<int, int>> bv;
```

```
struct {
  Shape<char, short*> a;
  ...
};
```

Replay 코드 예시 (AST 단에서의 구현 문제 – 매우 많은 case)

고려할 타입 문제 2. generic 함수 replay – 정리

C++ 소스 코드(AST)의 Syntax가 매우 다채로운 만큼,
template이 매우 다양한 syntax에서 사용될 수 있으며,
각각에 대한 구현이 필요하다.

add 정의

```
template<typename T>
T add(T a, T b) {
  return a + b;
}
```

반면 IR단에서는 각자 다른 add<int>, add<short> 등의
concrete한 타입의 전혀 다른 함수로 컴파일되어 사용된다.
사용한 타입에 대한 추적이 불필요.

```
define i32 @_Z3addIiET_S0_S0_(i32 %0, i32 %1)   = add<int>

define i16 @_Z3addIsET_S0_S0_(i16 %0, i16 %1) = add<short>
```

Replay 코드 예시 (AST 단에서의 구현 문제) – 그 외 부차적 문제

3. typedef 확인 – AST 코드 내부에서 typedef로 인해 같은 타입이 다른 타입으로 인식될 수 있음.

```
typedef unsigned long u64;
```

원본 타입 (unsigned long)으로 다룰 수 있도록 작업 필요.

4. lvalue, rvalue 확인

```
void foo (int & a) {};
void foo (int && a) {};
```

같은 타입, 같은 이름의 함수로 보이지만, 둘은 다른 함수.
구분하여 작업 필요.

5. Class access specifiers (private, protected)

```
class Rectangle {
private:
  int width;
  int area;
};
```

```
Rectangle Rectangle_replay() {
  context * ctx = read_input();
  Rectangle rect;
  rect.width = ctx->rect.width;
  rect.area = ctx->rect.area;

  return rect;
}
```

Compiler error!

private 접근 불가.

반면 IR단에서는 typedef, lvalue,rvalue, access specifier 모두 제거 되어 있음.
따라서 IR을 편집하는 방향을 선택.

현재 Carving/Replay 구현 단계

총 3천 줄 정도의 C++ 코드로, 자동으로 타겟 프로그램의 IR을 편집하여 Carving/Replay를 구현했으며,
계속해서 실제 C++ 오픈 소스 프로그램에 적용 하여 테스트중.

```
define void @__Carv__Rectangle(%class.Rectangle* %0) {
entry:
  %1 = getelementptr %0, 0        //Address of Rectangle.width
  %2 = load i32* %1               //Get width value
  call void @Carv_int(i32 %2)     //Write width value
  %3 = getelementptr %0, 1        //Address of Rectangle.area
  %4 = load i32* %3               //Get area value
  call void @Carv_int(i32 %4)     //Write area value
  ret void
}
```

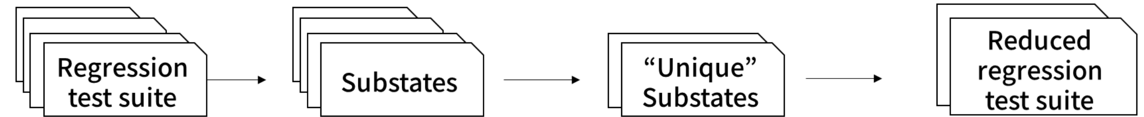
현재 적합한 application을 탐색 중…

Unit testing, API testing

```
sqlite3 * db = sqlite3_open();

sqlite3_prepare(db, "prepare", 2048);
sqlite3_db_config(db, 1, 2);
sqlite3_exec(db, "Insert", insert_call, context());
```

이 db object은 각 API가 제대로 작동할 수 있게 valid하면서도, test가 가능하도록 다양하게 만들 수 있어야 함. (false alarm 방지)

Test case reduction/priortization



Fault localization

SBFL (Statistical based Fault Localization)은 커버리지를 기반으로
성공한 테스트와 실패한 테스트를 비교하여 Fault가 발생할 가능성이 높은 라인을 도출
(Ex, 여러 실패한 테스트가가 함께 실행한 라인이 수상하다)

	Success tc1	Success tc2	Fail tc1
Line 1	실행	실행	실행하지 않음
Line 2	실행	실행하지 않음	실행

Line1 보다 Line 2에서 Fault가 발생했을 가능성이 높다.

Memory state를 비교하여
실행 결과 성공한 테스트여도, 실패한 테스트와 비슷한 테스트라면,
해당 테스트가 수행한 라인에서도 Fault가 발생했을 가능성이 높지 않을까 추측.

System test generation