PLRG
Programming Language
Research Group

형식화된 가정을 사용하여 실행 의미를 보존하는 바이너리 리프터 만들기

박지희, 윤인수, 류석영

2025 한국정보과학회 프로그래밍언어연구회 여름학교

2025. 08. 20.

바이너리 리프터란?

```
55 48 89 E5
48 83 EC 10
89 7D FC 89
75 F8 89 55
F4 83 7D F4
00 7E 11 8B
```

기계어

바이너리 리프터

```
Function 1
 Input (A, B):
  Output = A + B
```

고수준 중간 언어

- 정적 분석
- 동적 분석
- 사람이 직접 분석 및 보안 감사

이상: 정확성을 가진 바이너리 리프터

```c
int alloc_and_set() {
  int a[2] = {0, 0};
  a[0] = 1;
  return a[0];
}
```

원본 C 코드

```
55 48 89 e5
48 81 ec a8
02 00 00 c7
45 fc 01 00
00 00 8b 45
fc c9 c3 cc
```

기계어

```
Function alloc_and_set
 Input ():
  Output = 1
```

실행 의미가 보존된 중간 언어

현실 1: 일부 상황에서만 정확성을 가진 바이너리 리프터

```c
int alloc_and_set(i) {
  int a[2] = {0, 0};
  a[i] = 1;
  return a[0];
}
```

원본 C 코드

```
55 48 89 e5
48 81 ec b8
02 00 00 48
89 bd d8 fc
ff ff c7 84
45 e0 fc ff
```

기계어

```
Function alloc_and_set
 Input (A):
  Output =
    if (A == 0) 1
    else 0
```

실행 의미가 일부만 보존된 중간 언어

```
alloc_and_set(2)
```

현실 2: 정확성을 가지지 않는 바이너리 리프터

```
int alloc_and_set() {
  int a[2] = {0, 0};
  a[0] = 1;
  asm("lea
       $backdoor,
       %rbp");
  return a[0];}
```

원본 C 코드

```
55 48 89 e5
48 81 ec b8
02 00 00 48
89 bd d8 fc
ff ff c7 84
45 e0 fc ff
```

기계어

```
Function alloc_and_set
 Input ():
  Output = 1
```

실행 의미가 전혀 다른 중간 언어

문제의 원인: 리프터의 불완전한 가정

- 리프팅이 잘 동작하는 대상 (암시적, 원본 코드에 대한 가정):
  1. 의미가 잘 정의된 C 코드
  2. 일반적인 컴파일러 (GCC, LLVM)
  3. 일반적인 컴파일 옵션 (O0-O2)
- 하지만 리프팅 대상이 **이 가정을 만족하는지 확인할 방법이 없음**

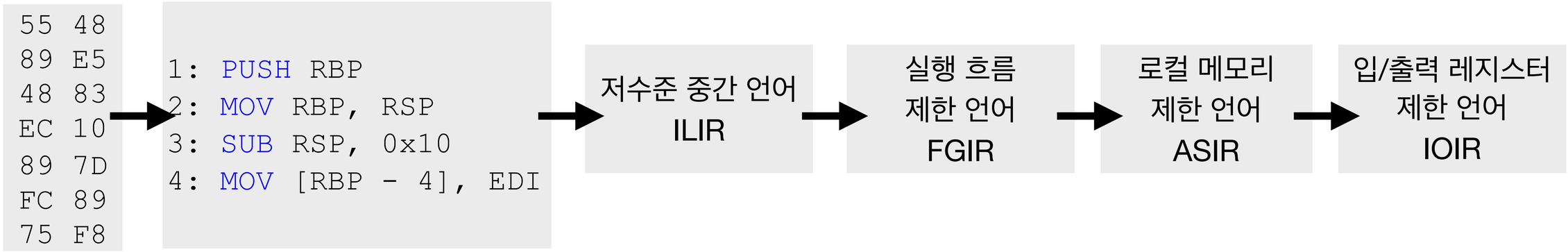
문제의 해결: 리프터의 가정을 형식화하고 정확성 증명

1. **기계어를 만드는 원본 코드 및 컴파일러에 대한 가정이 아니라, 대상 프로그램의 실행 성질에 대한 가정으로 형식화**

2. 실행 의미 보존의 기준으로서 가정에 맞는 실행 흐름 모방 (Filtered-Simulation) 을 정의하고, 가정이 항상 맞을 경우 일반적인 실행 의미 보존의 기준인 쌍모방 (Bisimulation) 과 같다는 것을 증명

형식화된 가정을 통한 바이너리 리프팅

디스어셈블리 명령어 단위의 리프팅 실행 흐름 분석 스택 사용 분석 레지스터 사용 분석



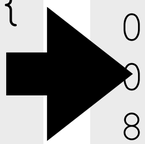
기계어 어셈블리어

형식화된 가정을 통한 바이너리 리프팅 - 예시

```
void mali (a, b) { … }
void nop () {}

int alloc_and_set(a, b) {
  void (*t)() = nop;
  int buf[1] = {0};
  buf[a] = b;
  t();
  return buf[0];
}
```

원본 C 코드



```
55 48
89 e5
48 81
ec b8
02 00
00 48
89 bd
d8 fc
ff ff
c7 84
45 e0
fc ff
```

기계어

함수 입력

a=0 b=1	a=4 b=mali	a=100 b=0	a=-1 b=mali

기계어의 실행 의미

nop 호출 → 정상 종료	nop 호출 → mali 호출 → 메모리 접근 오류	함수는 정상 종료/호출 중인 함수의 변수값 훼손	mali 호출 → 정상 종료

형식화된 가정을 통한 바이너리 리프팅 - 예시

```
int alloc_and_set
(a, b) {
  void (*t)() = nop;
  int buf[1] = {0};
  buf[a] = b;
  t();
  return buf[0];
}
```

원본 C 코드

```
55 48
89 e5
48 81
ec b8
02 00
00 48
89 bd
d8 fc
ff ff
c7 84
45 e0
fc ff
```

기계어

```
mov [rsp-8], nop
mov [rsp-16], 0
mov [rsp-16+rdi], rsi
mov rax, [rsp-8]
call rax
mov rax, [rsp-16]
leave
ret
```

어셈블리어

```
alloc_and_set:
  store (rsp - 8) nop
  store (rsp - 16) 0
  store (rsp - 16 + rdi) rsi
  load temp (rsp - 8)
  assign rsp (rsp - 16)
  store rsp after_call
  jump temp

after_call:
  load rax (rsp - 16)
  load ret_addr rsp
  assign rsp (rsp + 8)
  jump ret_addr
```

ILIR

저수준 중간 언어: ILIR

ILIR 코드

```
alloc_and_set:
  store (rsp - 8) nop
  store (rsp - 16) 0
  store (rsp - 16 + rdi) rsi
  load temp (rsp - 8)
  assign rsp (rsp - 16)
  store rsp after_call
  jump temp

after_call:
  load rax (rsp - 16)
  load ret_addr rsp
  assign rsp (rsp + 8)
  jump ret_addr
```

함수 입력

a=0 b=1	a=4 b=mali	a=100 b=0	a=-1 b=mali
nop 호출 → 정상 종료	nop 호출 → mali 호출 → 메모리 접근 오류	함수는 정상 종료/호출 중인 함수의 변수값 훼손	mali 호출 → 정상 종료

ILIR의 실행 의미

형식화된 가정: 없음

- 하나의 어셈블리 명령어를 하드웨어 중립적인 명령어의 조합으로 표현함

함수 및 그 실행 흐름 범위가 제한된 언어: FGIR

FGIR 코드

```
alloc_and_set:
  store (rsp - 8) nop
  store (rsp - 16) 0
  store (rsp - 16 + rdi) rsi
  load temp (rsp - 8)
  assign rsp (rsp - 16)
  store rsp after_call
  call temp [→ after_call]

after_call:
  load rax (rsp - 16)
  load ret_addr rsp
  assign rsp (rsp + 8)
  ret ret_addr
```

함수 입력

a=0 b=1	a=4 b=mali	a=100 b=0	a=-1 b=mali

FGIR의 실행 의미

nop 호출 → 정상 종료	nop 호출 → ret에서 가정 위반으로 정지	함수는 정상 종료/호출 중인 함수의 변수값 훼손	mali 호출 → 정상 종료

형식화된 가정: [→ addr]

- 이 호출 명령어와 대응되는 반환 명령어의 타겟 주소는 항상 addr일 것이다

함수의 로컬 메모리를 제한하는 언어: ASIR

ASIR 코드

```
alloc_and_set(Stack[-16:0]):
  store stack[-8] nop
  store stack[-16] 0
  store stack[-16+rdi] rsi
  load temp stack[-8]
  assign rsp (rsp - 16)
  store stack[-24] after_call
  call temp [→ after_call]

after_call(Stack[-16:0]):
  load rax stack[-16]
  load ret_addr stack[0]
  assign rsp (rsp + 8)
  ret ret_addr
```

함수 입력

a=0 b=1	a=4 b=mali	a=100 b=0	a=-1 b=mali

ASIR의 실행 의미

nop 호출 → 정상 종료	nop 호출 → ret에서 가정 위반으로 정지	[-16+rdi] 접근에서 가정 위반으로 정지	mali 호출 → 정상 종료

형식화된 가정: Stack[a:b]

- 이 블록에서 사용되는 로컬 스택 메모리의 최소/최대 주소는 a/b이다

함수의 입력/출력 레지스터를 제한하는 언어: IOIR

IOIR 코드

```
alloc_and_set(Stack[-16:0])
(Input [a, b, rsp]):
  store stack[-8] nop
  store stack[-16] 0
  store stack[-16+a] b
  load temp stack[-8]
  assign rsp (rsp - 16)
  store stack[-24] after_call
  call temp [→ after_call]
  (Reads []/ Writes [])
after_call(Stack[-16:0]):
  load x stack[-16]
  load ret_addr stack[0]
  assign rsp (rsp + 8)
  ret ret_addr
  (Output [x, rsp])
```

함수 입력

a=0 b=1	a=4 b=mali	a=100 b=0	a=-1 b=mali

IOIR의 실행 의미

nop 호출 → 정상 종료	nop 호출 → ret에서 가정 위반으로 정지	[-16+a] 접근에서 가정 위반으로 정지	mali 호출에서 가정 위반으로 정지

형식화된 가정: (Input / Output / Reads / Writes) R1, R2, …

- Input/Output: 이 함수 내에서는 R1, R2, … 의 값만 읽는다/변경한다
- Reads/Writes: 호출하는 함수 내에서는 R1, R2, …의 값만 읽는다/변경한다

문제의 해결: 리프터의 가정을 형식화하고 정확성 증명

1. 기계어를 만드는 원본 코드 및 컴파일러에 대한 가정이 아니라, 대상 프로그램의 실행 성질에 대한 가정으로 형식화

2. **실행 의미 보존의 기준으로서 가정에 맞는 실행 흐름 모방 (Filtered-Simulation) 을 정의하고, 가정이 항상 맞을 경우 일반적인 실행 의미 보존의 기준인 쌍모방 (Bisimulation)과 같다는 것을 증명**

가정에 맞는 실행 흐름 모방 (Filtered-Simulation)

FGIR

```
alloc_and_set:
  store (rsp - 8) nop
  store (rsp - 16) 0
  store (rsp - 16 + rdi) rsi
  load temp (rsp - 8)
  assign rsp (rsp - 16)
  store rsp after_call
  call temp [→ after_call]

after_call:
  load rax (rsp - 16)
  assign rsp (rsp + 16)
  load ret_addr rsp
  ret ret_addr
```

2

a=0
b=1

ASIR

```
alloc_and_set(Stack[-16:0]):
  store stack[-8] nop
  store stack[-16] 0
  store stack[-16+rdi] rsi
  load temp stack[-8]
  assign rsp (rsp - 16)
  store stack[-24] after_call
  call temp [→ after_call]

after_call(Stack[-16:0]):
  load rax stack[-16]
  load ret_addr stack[0]
  assign rsp (rsp + 8)
  ret ret_addr
```

1

a=0
b=1

- 고수준 언어가 실행 의미에 의해 가정 위반 없이 한 단계 실행되면, 대응되는 저수준 언어는 항상 한 단계 실행되고 대응 관계를 유지함

가정에 맞는 실행 흐름 모방 (Filtered-Simulation)

FGIR

```
alloc_and_set:
  store (rsp - 8) nop
  store (rsp - 16) 0
  store (rsp - 16 + rdi) rsi
  load temp (rsp - 8)
  assign rsp (rsp - 16)
  store rsp after_call
  call temp [→ after_call]

after_call:
  load rax (rsp - 16)
  assign rsp (rsp + 16)
  load ret_addr rsp
  ret ret_addr
```

1

a=100
b=0

ASIR

```
alloc_and_set(Stack[-16:0]):
  store stack[-8] nop
  store stack[-16] 0
  store stack[-16+rdi] rsi
  load temp stack[-8]
  assign rsp (rsp - 16)
  store stack[-24] after_call
  call temp [→ after_call]

after_call(Stack[-16:0]):
  load rax stack[-16]
  load ret_addr stack[0]
  assign rsp (rsp + 8)
  ret ret_addr
```

2

a=100
b=0

가정 위반으로 정지

- 저수준 언어가 한 단계 실행되면, 고수준 언어가 한 단계 실행되고 대응 관계를 유지하거나 가정을 위반하여 정지함. 가정이 모든 실행에 대해 맞는 경우, 쌍모방(Bisimulation) 관계가 성립

구현 및 평가

제안한 형식화된 가정을 사용한 바이너리 리프팅 도구 FIBLE

(**FI**ltered **B**inary **L**ifting and **E**xecution)을 구현함.

1. 표현력 (Expressiveness) : 잘 정의된 C 코드에 대해서는 제안한 리프터로 표현 가능한가? (가정 위반이 없는가?)

2. 정확성 (Correctness): 구현이 정말 제안한 실행 의미 보존 기준을 만족하는가?

형식화된 가정을 통한 바이너리 리프팅의 표현력

- 잘 정의된 C코드 모음으로 두 종류의 벤치마크 선택
  1. Coreutils + musl-libc: 37개 프로그램
     - 테스트가 있는 프로그램 중 실행기에서 모사할 수 있는 시스템 호출만 사용하는 프로그램들을 선정 (ls, mv, cat, wc, ..)
  2. DARPA CGC dataset: 44개 프로그램
     - DARPA에서 만든 바이너리 자동 분석 챌린지를 위한 데이터셋으로, 다양한 종류의 프로그램을 포함 (체스 엔진, HTTP 프로토콜, 몬테카를로 시뮬레이션, ..)
- 모두 GCC 9.4.0 -O0 옵션을 이용해 컴파일

형식화된 가정을 통한 바이너리 리프팅의 표현력

각 중간 언어에 대한 실행기 구현 및 시스템 호출 모사 후, 주어진 테스트를 실행하며 유추한 가정을 위반하는지 확인

- 결과: IOIR에서 두 벤치마크의 테스트 모두 70% 이상 통과, 특히 실행 흐름 복원 (FGIR)의 경우 Coreutils 100% 통과

Dataset	# Total	# FGIR passed	# ASIR passed	# IOIR passed
Coreutils	37	37 (100.0 %)	32 (86.5 %)	32 (86.5 %)
CGC	44	41 (93.2 %)	31 (70.5 %)	31 (70.5 %)

형식화된 가정을 통한 바이너리 리프팅의 표현력

가정을 위반하여 테스트에 통과하지 못한 프로그램들에 대한 조사 결과:

- 잘 정의되지 않은 동작 사용 (5), 구현 정의 동작 사용 (8), 부정확한 가정 유추 (2)

Type	# Violated		Example
	Coreutils	CGC	
Uninitialized stack	0	5	`int x; int y = x;`
Compare abstract address with concrete address	4	4	`if (&x < &global)`
Imprecise stack bound analysis	1	1	✗

형식화된 가정을 통한 바이너리 리프팅의 정확성

- 가정에 맞는 실행 흐름 모방 성질을 확인하는 검사기 구현:
  - 모든 단계의 프로그램을 동시에 실행해, 매 실행 단계마다 각 프로그램의 실행 상태를 비교하여 정의한 모방 관계를 만족하는지 확인

- 표현력 실험과 같은 벤치마크에서 테스트한 결과, 모든 실행에서 가정에 맞는 실행 흐름 모방 성질이 만족됨을 확인함

결론

- 문제: 암시적 가정 하에 정확성을 목표로 하는 바이너리 리프터들이 개발되었지만, 가정도 모호하고 변환의 정확성을 증명할 방법도 없음
- 제안: 형식화된 가정을 사용한 바이너리 리프팅 과정을 정의하고, 변환의 정확성을 가정에 맞는 실행 흐름 모방이라는 개념으로 정의
- 구현 및 실험을 통해 형식화된 바이너리 리프팅이 실제 바이너리를 잘 표현한다는 것을 확인

Bridging the Gap between Real-World and Formal Binary Lifting through Filtered-Simulation

JIHEE PARK, KAIST, South Korea
INSU YUN, KAIST, South Korea
SUKYOUNG RYU, KAIST, South Korea

Binary lifting is a key component in binary analysis tools. In order to guarantee the correctness of binary lifting, researchers have proposed various formally verified lifters. However, such formally verified lifters have too strict requirements on binary, which do not sufficiently reflect real-world lifters. In addition, real-world lifters use heuristic-based assumptions to lift binary code, which makes it difficult to guarantee the correctness of the lifted code using formal methods. In this paper, we propose a new interpretation of the correctness of real-world binary lifting. We formalize the process of binary lifting with heuristic-based assumptions used in real-world lifters by dividing it into a series of transformations, where each transformation represents a lift with new abstraction features. We define the correctness of each transformation as *filtered-simulation*, which is a variant of bi-simulation, between programs before and after transformation. We present three essential transformations in binary lifting and formalize them: (1) control flow graph reconstruction, (2) abstract stack reconstruction, and (3) function input/output identification. We implement our approach for x86-64 Linux binaries, named FIBLE, and demonstrate that it can correctly lift Coreutils and CGC datasets compiled with GCC.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**; • **Theory of computation** → **Program semantics**.

Additional Key Words and Phrases: Binary lifting, formal semantics

ACM Reference Format:
Jihee Park, Insu Yun, and Sukyoung Ryu. 2025. Bridging the Gap between Real-World and Formal Binary Lifting through Filtered-Simulation. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 112 (April 2025), 29 pages. https://doi.org/10.1145/3720524

1 Introduction

Binary analysis is essential for analyzing programs that do not have source code, such as COTS (Commercial Off-The-Shelf) software, malware, and legacy software. By using binary analysis, one can build a variety of techniques on programs without source code, including vulnerability analysis [14], malware detection [15], binary rewriting, optimization [20], and more.

Binary analysis frameworks offer a set of tools and techniques for analyzing binary code. Most of these frameworks utilize Intermediate Representations (IRs) that are well-suited for binary code analysis. They provide various high-level abstractions that are not inherently present in the binary code, such as Control Flow Graphs (CFGs), functions, and global/local variables. These abstractions enable developers to design precise and efficient analysis tools [6, 38].

Authors' Contact Information: Jihee Park, jiheepark@kaist.ac.kr, KAIST, Daejeon, South Korea; Insu Yun, insuyun@kaist.ac.kr, KAIST, Daejeon, South Korea; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea.

ACM 2475-1421/2025/4-ART112
https://doi.org/10.1145/3720524

OOPSLA 2025

바이너리 리프팅을 통한 기계어 검증

Machine Code Verification Using Binary Lifting

박지희 류석영
PLRG@KAIST

1. 동기: 자동화된 기계어 검증의 어려움

프로그램 자동 검증에 특화된 논리(Hoare Logic 등)를 기계어 검증에 사용할 때의 문제:

문제 1: indirect jump의 복잡성

```
call_to_input:     switch_table:          return:
 callq  %rdi        jmpq *table(,%rdi,8)    retq
```

다양하게 쓰이는 indirect jump를 하나의 rule로 처리하기 위해서는 복잡한 reasoning principle이 필요

문제 2: memory layout의 부재

```
write_anywhere:
 movq  $1, (%rdi)
```
{??} write_anywhere {safe}

컴파일러 정보가 없기 때문에 특정 write이 이후 실행 흐름에 문제를 만드는지 쉽게 알 수 없음

2. 아이디어: 바이너리 리프팅

기계어를 입력으로 받아, 고급 언어 기능을 복원하는 대신 만족해야 할 가정을 포함하는 중간 언어 생성

```
method add[stack range=[-16..+8]]
          (rsp, rbp, rdi, rsi):
[pushq+0]: tmp_1 = rbp;
[pushq+1]: rsp = rsp - 8;
…
[retq+0]: rsp = rsp + 8;
[retq+1]: return rax;
```

해결 1: indirect jump를 call / intra-jump / return으로 나눠 각각 다른 reasoning principle 적용

해결 2: 메모리 접근이 각 함수의 stack range 안으로 접근하는 것을 검증하여 실행 흐름에 영향을 끼치지 않는지 확인 가능

3. 리프팅 후 검증의 건전성 증명

- 변환의 올바름: 변환 전후 프로그램 사이의 bisimulation을 통한 의미 보존

  * Bisimulation: 두 프로그램의 상태가 대응될 때 한 쪽에서 진행된 상태가 다른 쪽에서 진행된 상태와 계속 대응되는 성질을 가진 relation

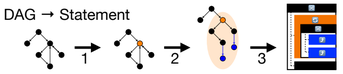
- 변환된 프로그램 사이의 검증 호환성: flow-chart 기반 검증에서 되는 건 리프팅 후 Hoare logic으로 검증 가능

Lean을 통해 변환 정의 및 검증 호환성 증명 진행 중

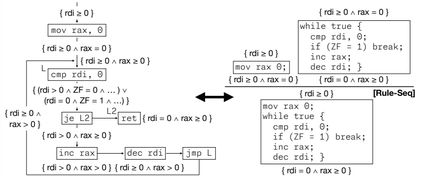
3.1. 변환의 올바름 증명

Control Flow Graph(CFG) 형태의 프로그램을 여러 단계를 통해 statement 형태의 프로그램으로 변환 및 bisimulation을 정의함

1. CFG Loop head detection
2. CFG → Directed Acyclic Graph (DAG)
3. DAG → Statement



3.2. 변환된 프로그램 사이의 검증 호환성 증명

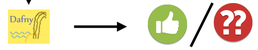


프로그램 변환 시 CFG node와 statement에 대한 대응을 통해 한 쪽 프로그램에 대한 성질을 다른 쪽으로 변환

4. 적용: Dafny로 성질 자동 검증

리프팅된 중간 언어를 입력으로 받아, 같은 의미를 가진 Dafny 코드를 생성한 뒤 함수가 만족해야 할 입력의 조건/출력의 성질 작성

```
method add(st: State.T, RDI: Value.T, RSI: Value.T)
 returns (st: State.T, RAX: Value.T)
 requires Int32(RDI) && Int32(RSI)
 requires Int32SignedBound(ToInt(RDI) + ToInt(RSI))
 ensures Int32(RAX) && ToInt(RAX) == ToInt(RDI) + ToInt(RSI)
```



검증 실패 시, 힌트를 추가하거나 바이너리 리프팅 과정에 추가된 가정을 수정하여 재시도

5. 연구 진행 상황

- 리프팅 후 검증의 건전성 증명: 첫 번째 변환 단계 정의 및 대응 증명 중
- Dafny로 성질 자동 검증: 기본적인 arithmetic 함수 및 memcpy에 대한 검증 성공

KAIST

PLRG Programming Language Research Group

후속 연구 (포스터 세션)