# 한양대학교 프로그램 합성 및 프로그램 분석 연구소개

이 우 석

한양대학교 ERICA 소프트웨어학부

2022 소프트웨어재난연구센터 겨울 정기 워크샵

2022. 02. 10

HANYANG UNIVERSITY

# Contents

- **프로그램 합성 소개**
- 진행중인 합성 및 분석 연구들 소개
  - 합성기반 동형암호 프로그램 최적화
  - 안드로이드 인스턴트 앱 자동생성
  - 재귀호출 프로그램 합성
  - 동형암호 기반 개인정보 유출없는 정적 분석

# 프로그램 합성

- 사용자가 **원하는** (생김새 + 행동) 프로그램을 **자동** 생성하는 기술

프로그래밍 할 줄 몰라도
표현할 수 있어야
(예: 입출력 예제)


- 1945년부터 언급 (앨런 튜링)

> *Instruction tables will have to be made up by mathematicians with computing experience and perhaps a certain puzzle-solving ability… This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.* [1]

실현될 경우 개발의 무거운 짐을 덜어줄 것으로 기대되는, 전산학의 성배(holy grail)

# 프로그램 합성과 연관기술 비교 (합성 vs. 컴파일러)

```
let rec insert x xs =
    match xs with
    | [] -> x :: []
    | h :: t ->
if x <= h then x :: xs
else h :: (insert x t)
```

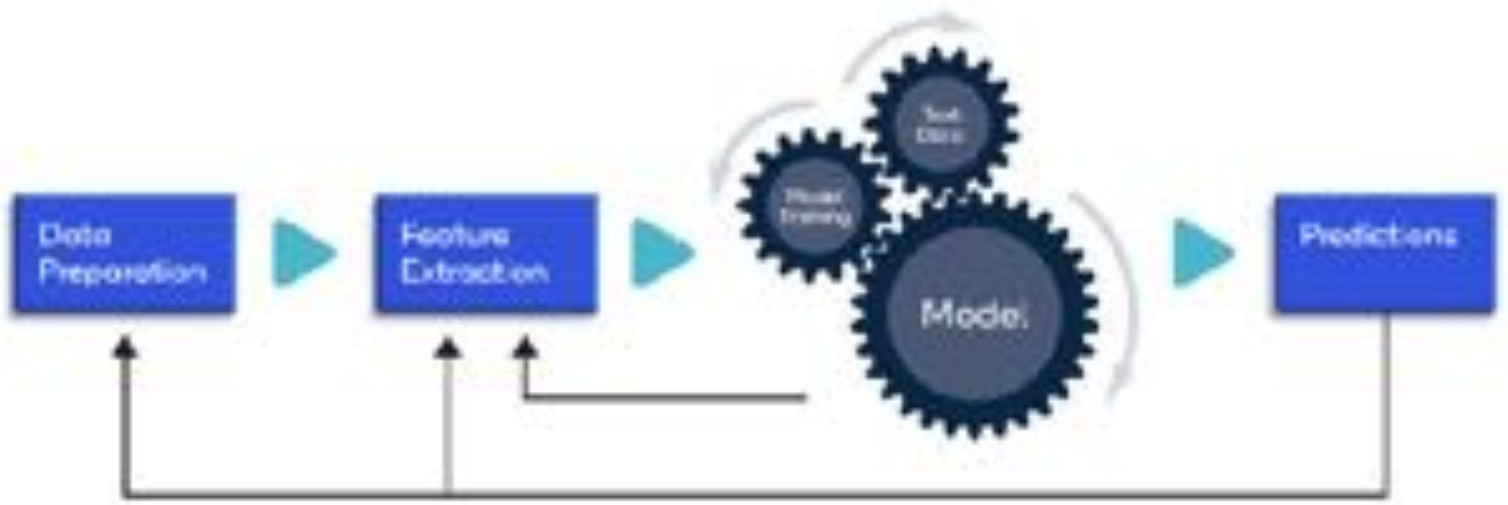
**OCaml**

```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

**어셈블리**

- **공통점**: 고 수준 언어로 쓰인 개념을 실행 가능한 프로그램으로 변환
- **차이점**:
- 컴파일러: 탐색 과정 없이 단순 변환
- 합성:“어떻게” 주어진 목표를 달성할지 컴퓨터가 **탐색**

# 프로그램 합성과 연관기술 비교
# (합성 vs. 기계학습)




- **공통점**: 주어진 입출력 데이터에 맞게 작동하는 함수 찾기

- **차이점**:

  - 기계학습: 튜링완전turing complete하지 않은 대상(결정트리, 인공신경망 구조 등) 학습, (대개)결과물 **해독 불가**, 주어진 데이터에 꼭 맞지 않아도 됨 (노이즈 허용).

  - 합성: 일반 프로그램 대상(튜링 완전), 해독 가능, 주어진 데이터에 꼭 맞아야.

# 프로그램 합성의 두 방향

| | 엄밀한 합성 | 느슨한 합성 |
|---|---|---|
| **대두 시기** | 1960년대 | 2010년대 (딥러닝의 출현과 함께) |
| **생김새 및 행동제약 조건 만족여부** | **항상 만족** | **만족 못시킬 수 있음** |
| **속도** | 느림 | 빠름 |
| **방법** | **알고리즘 수행** | **신경망 추론** |
| **대표적 예** | Excel FlashFill | GitHub Copilot |
| **발표되는 학회** | POPL, PLDI, ICSE, FSE, …<br>(PL 및 SE 저명학회) | NIPS, ICML, AAAI, …<br>(ML 저명학회) |

# 느슨한 합성의 예 — GitHub Copilot

```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

- 자연어 (+ 입출력 예제) → Python 코드
- OpenAI의 GPT-3 를 코드 생성에 맞게 특화
- 164개 합성 문제 중 28%에 대해서만 맞는 프로그램 생성 (문법에 맞지 않는 프로그램 생성 가능)
- 생성되는 프로그램 크기 ↑ → 오류 확률 기하급수적으로 증가

# 프로그램 합성의 두 방향

| | 엄밀한 합성 | 느슨한 합성 |
|---|---|---|
| 대두 시기 | 1960년대 | 2010년대 (딥러닝의 출현과 함께) |
| 생김새 및 행동제약 조건 만족여부 | 항상 만족 | 만족 못시키는 경우 흔함 |
| 속도 | 느림 | 빠름 |
| 방법 | 알고리즘 수행 | 신경망 추론 |
| 대표적 예 | Excel FlashFill | GitHub Copilot |
| 발표되는 학회 | POPL, PLDI, ICSE, FSE, … (PL 및 SE 저명학회) | NIPS, ICML, AAAI, … (ML 저명학회) |

**우리의 주된 연구 관심**

# Contents

- 프로그램 합성 소개
- 진행중인 합성 및 분석 연구들 소개
  - **합성기반 동형암호 프로그램 최적화 (with 서울대)**
  - 안드로이드 인스턴트 앱 자동생성
  - 재귀호출 프로그램 합성
  - 동형암호 기반 개인정보 유출없는 정적 분석

# References

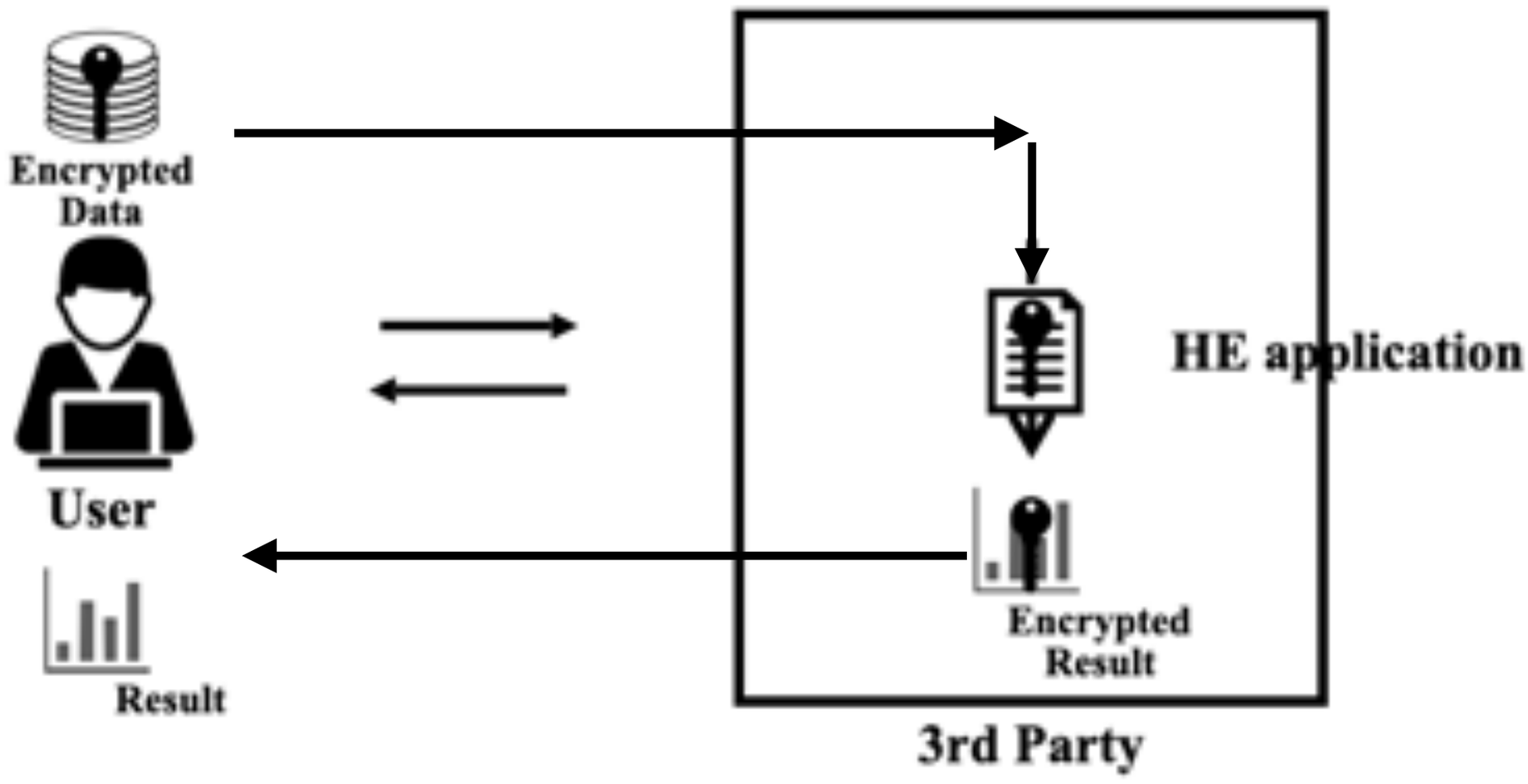

- 이동권, 이우석, 오학주, 이광근, Optimizing Homomorphic Evaluation Circuits by Program Synthesis, Term Rewriting, ACM PLDI 2020

- Optimizing Homomorphic Evaluation Circuits by Program Synthesis, Term Rewriting, and Time-Bounded Exhaustive Search, In submission to ACM TOPLAS

  - 본 연구실에서 개발한 최신 합성기 Duet (POPL'21)을 사용하고 동일식 모두 모으기(equality saturation)을 적용하여 성능 개선

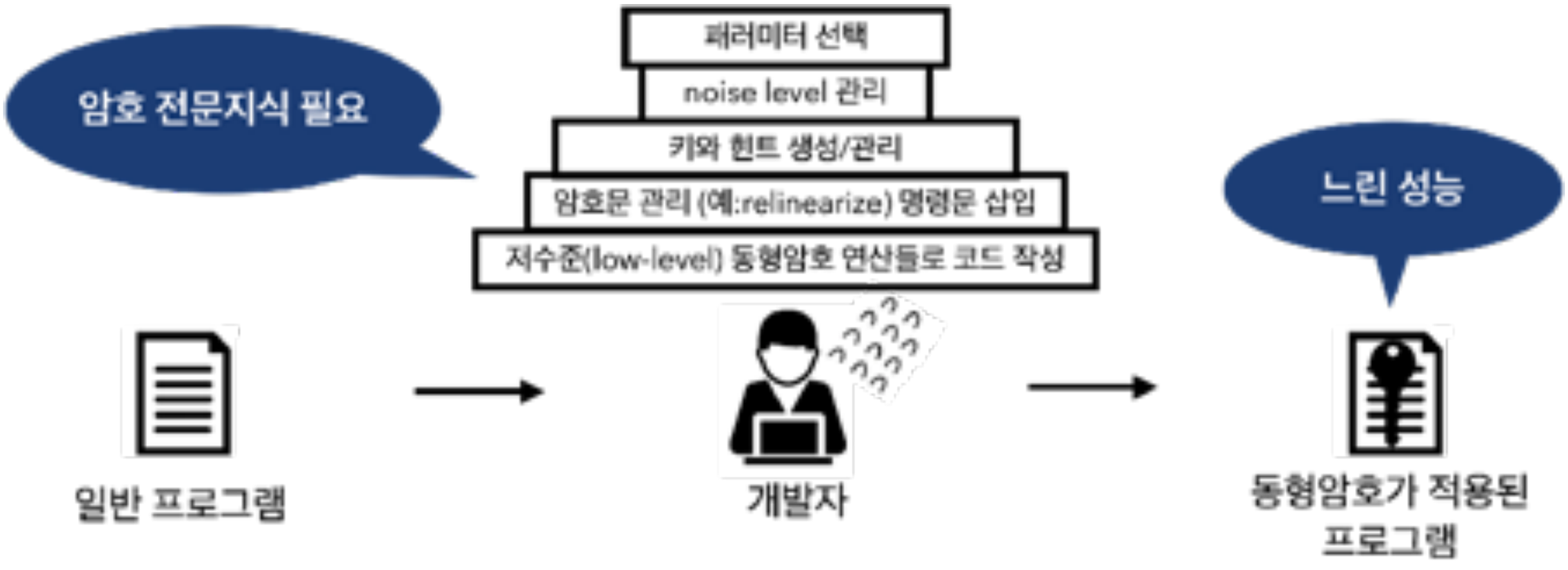
# 동형암호(Homomorphic Encryption)

- 암호화된 데이터에 임의의 연산 수행 가능한 암호
- 데이터보안 100% 보장



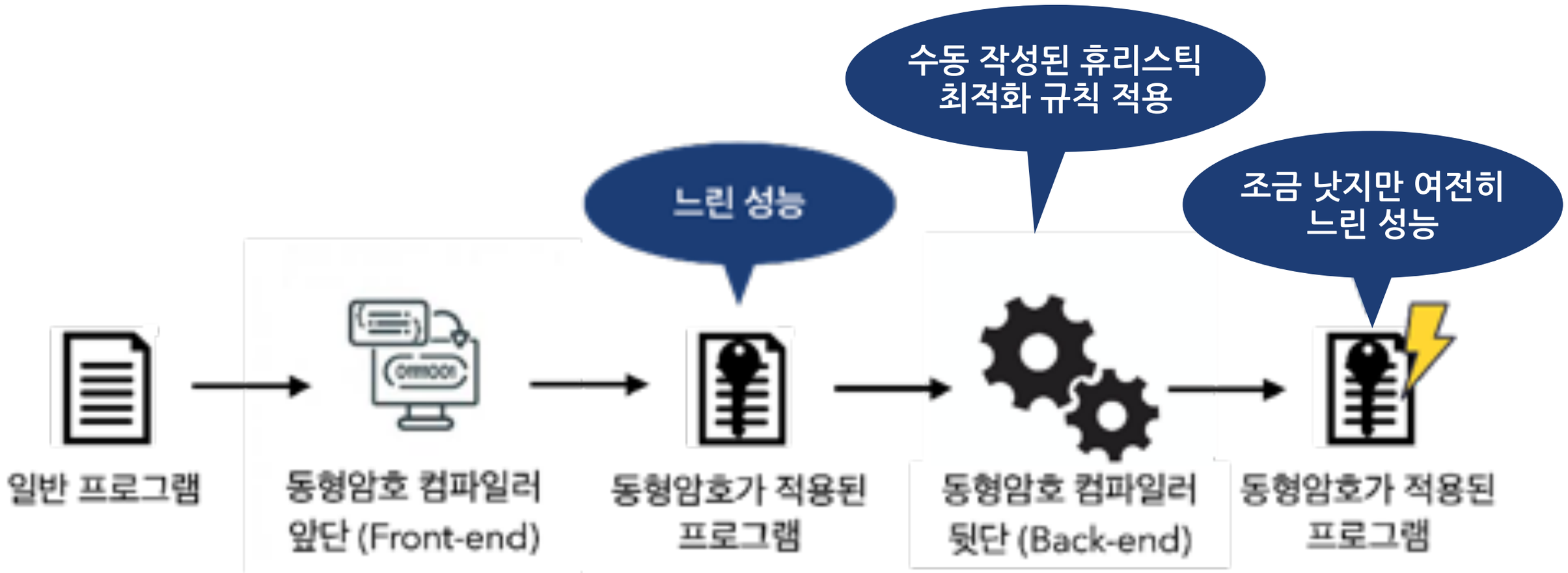
# 동형암호 프로그램 개발

암호학적 지식 필수 (갖춰도 어려움!)



# 동형암호 컴파일러

- 평문 프로그램으로부터 동형암호 프로그램 자동 생성
- 최적화: 수동 작성된 휴리스틱 기반



# 수동작성 vs. 동형암호 컴파일러 이용 시 코드 (두 암호화된 정수 덧셈)

```
#include "FHE.h"
#include "EncryptedArray.h"
#include <NTL/lzz_pXFactoring.h>
#include <fstream>
#include <sstream>
#include <sys/time.h>

int main(int argc, char **argv)
{
    long m=0, p=2, r=1; // Native plaintext space
                        // Computations will be 'modulo p'
    long L=16;          // Levels
    long c=3;           // Columns in key switching matrix
    long w=64;          // Hamming weight of secret key
    long d=0;
    long security = 128;
    ZZX G;
    m = FindM(security,L,c,p, d, 0, 0);
    FHEcontext context(m, p, r);
    buildModChain(context, L, c);
    FHESecKey secretKey(context);
    const FHEPubKey& publicKey = secretKey;
    G = context.alMod.getFactorsOverZZ()[0];
    secretKey.GenSecKey(w);
    addSome1DMatrices(secretKey);
    EncryptedArray ea(context, G);
    vector<long> v1;
    v1.push_back(atoi(argv[1]));
    Ctxt ct1(publicKey);
    ea.encrypt(ct1, publicKey, v1);
    v2.push_back(atoi(argv[2]));
    Ctxt ct2(publicKey);
    ea.encrypt(ct2, publicKey, v2);
}
```

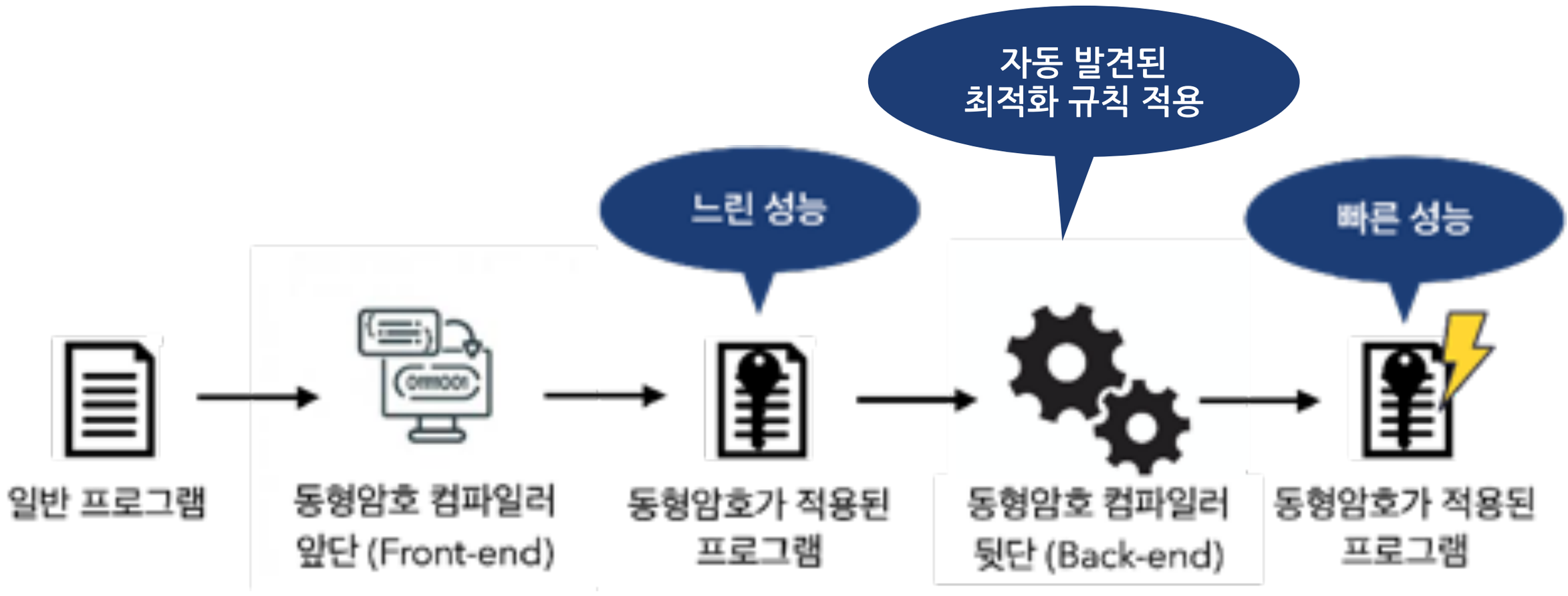
HElib (동형암호 라이브러리) 이용 수동작성

```
#include <iostream>
#include <fstream>
#include <integer.hxx>

int main()
{
        Integer8 a, b, c;

        cin >> a;
        cin >> b;
        c = a + b;

        cout << c;
        FINALIZE_CIRCUIT(blif_name);
}
```

Cingulata (동형암호 컴파일러) 이용시

# 프로그램 합성 기반 최적화

- 사람이 찾기 힘든 최적화 규칙 발견 자동 발견 (by 합성)
- 기존 휴리스틱 기반 기법보다 우수한 최적화 성능



# 간단한 동형암호 스킴

- Based on approximate common divisor problem
- $p$ : integer as a secret key
- $q$ : random integer
- $r$ ( $\ll |p|$ ) : random noise for security

$$Enc_p(\ \mu \in \{0,1\}\ ) = pq + 2r + \mu$$
$$Dec_p(c) = (c \bmod p) \bmod 2$$
$$Dec_p(Enc_p(\mu)) = Dec_p(pq + 2r + \mu) = \mu$$

- For ciphertexts $\underline{\mu_i} \leftarrow Enc_p(\mu_i)$, the following holds

$$Dec_p(\underline{\mu_1} + \underline{\mu_2}) = \mu_1 + \mu_2$$
$$Dec_p(\underline{\mu_1} \times \underline{\mu_2}) = \mu_1 \times \mu_2$$

- The scheme can evaluate all boolean circuits as $+$ and $\times$ in $\mathbb{Z}_2 = \{0,1\}$ are equal to XOR and AND

# 장애물: 증가하는 **Noise**

- Noise increases during homomorphic operations.
- For $\underline{\mu_i} = pq_i + 2r_i + \mu_i$

$$\underline{\mu_1} + \underline{\mu_2} = p(q_1 + q_2) + \boxed{2(r_1 + r_2) + (\mu_1 + \mu_2)} \text{ double increase}$$

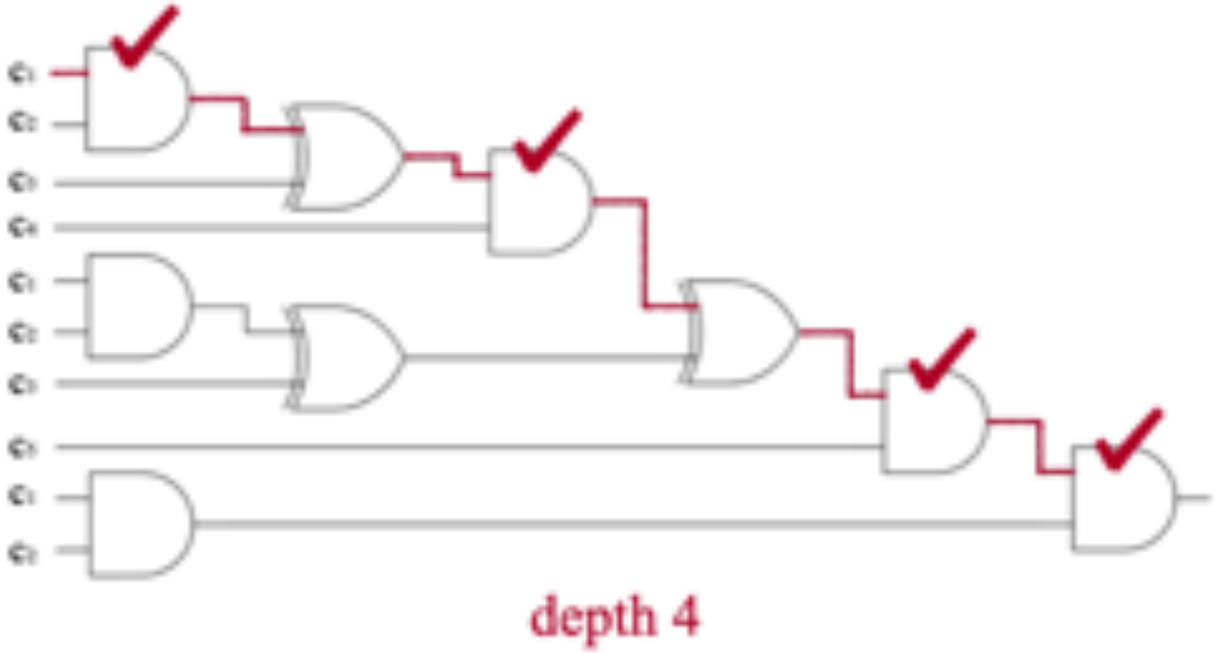
$$\underline{\mu_1} \times \underline{\mu_2} = p(pq_1q_2 + \cdots) + \boxed{2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + (\mu_1 \times \mu_2)} \text{ quadratic increase}$$

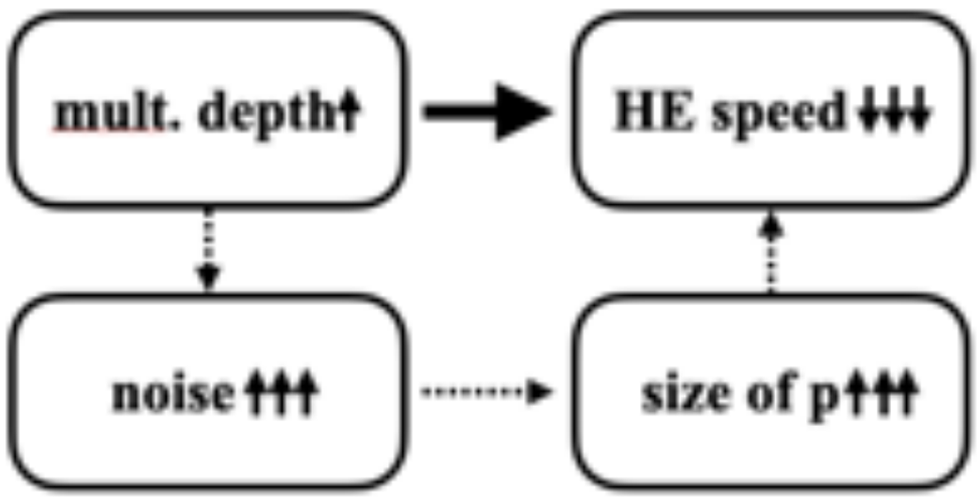
noise

- **if (noise $> p$) then incorrect results**

# 곱셈 깊이: 가장 중요한 성능 지표

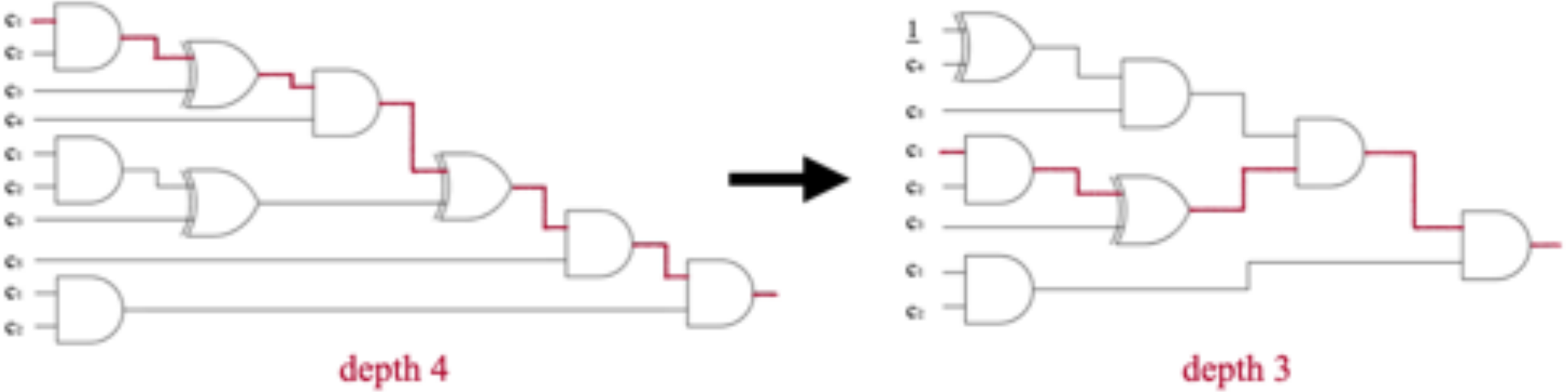
- 곱셈 깊이: 입력에서 출력으로 이르는 길 중 가장 많은 누적 곱셈 연산 횟수



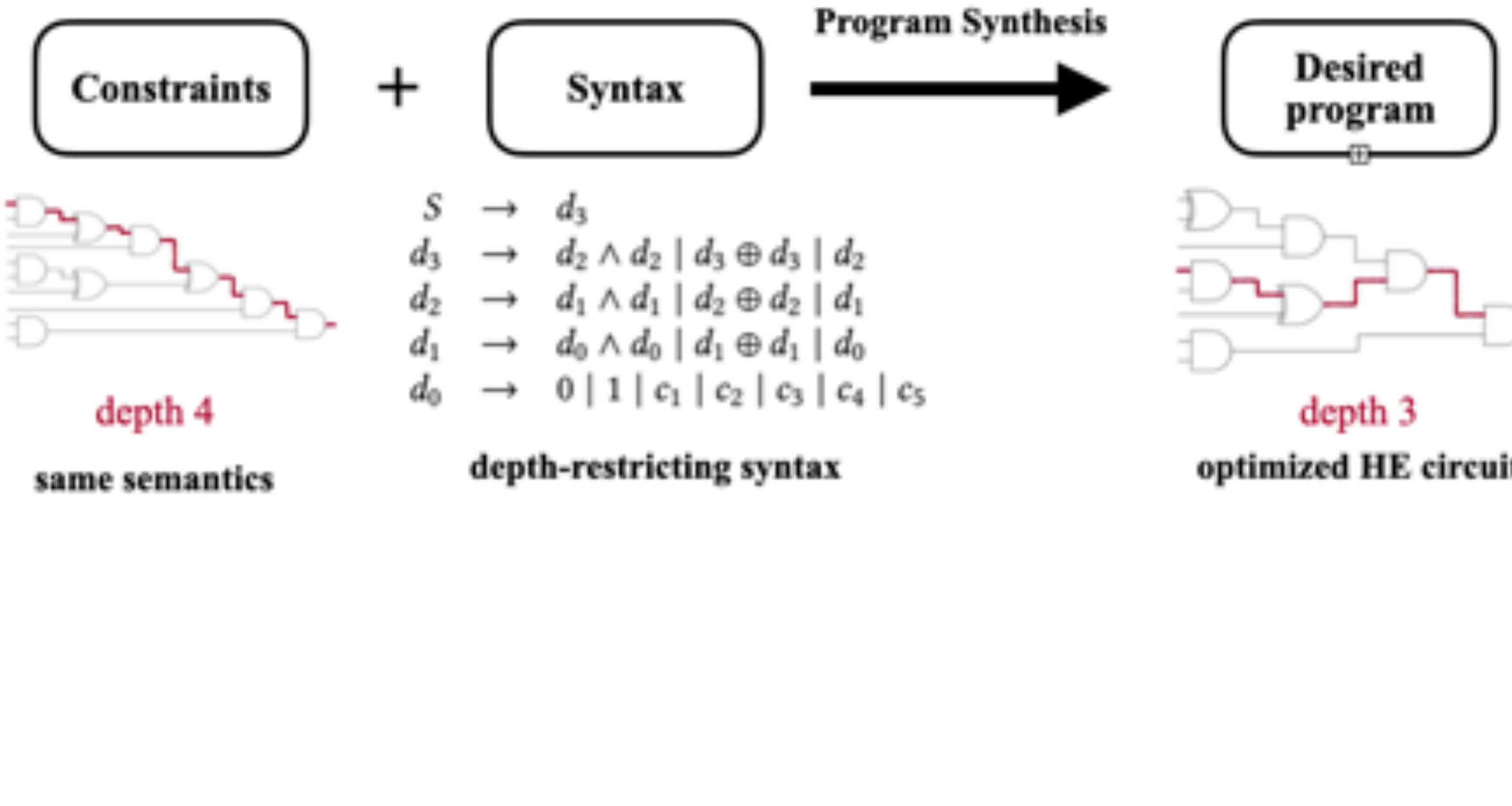



# 동형암호 최적화 = 곱셈깊이 축소

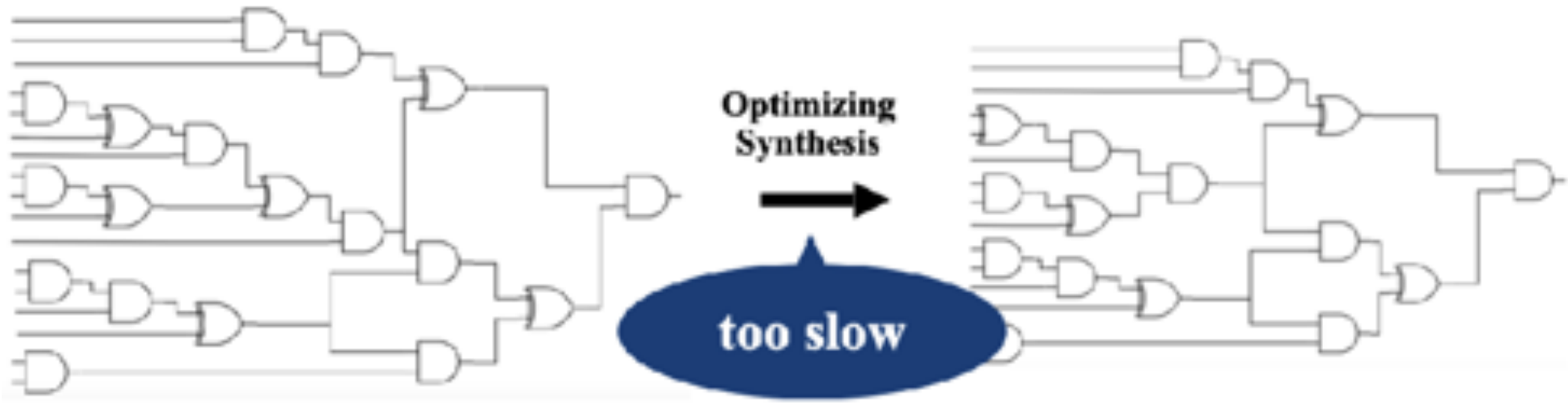
- 의미는 동일하며 더 적은 곱셈 깊이를 갖는 회로 찾기



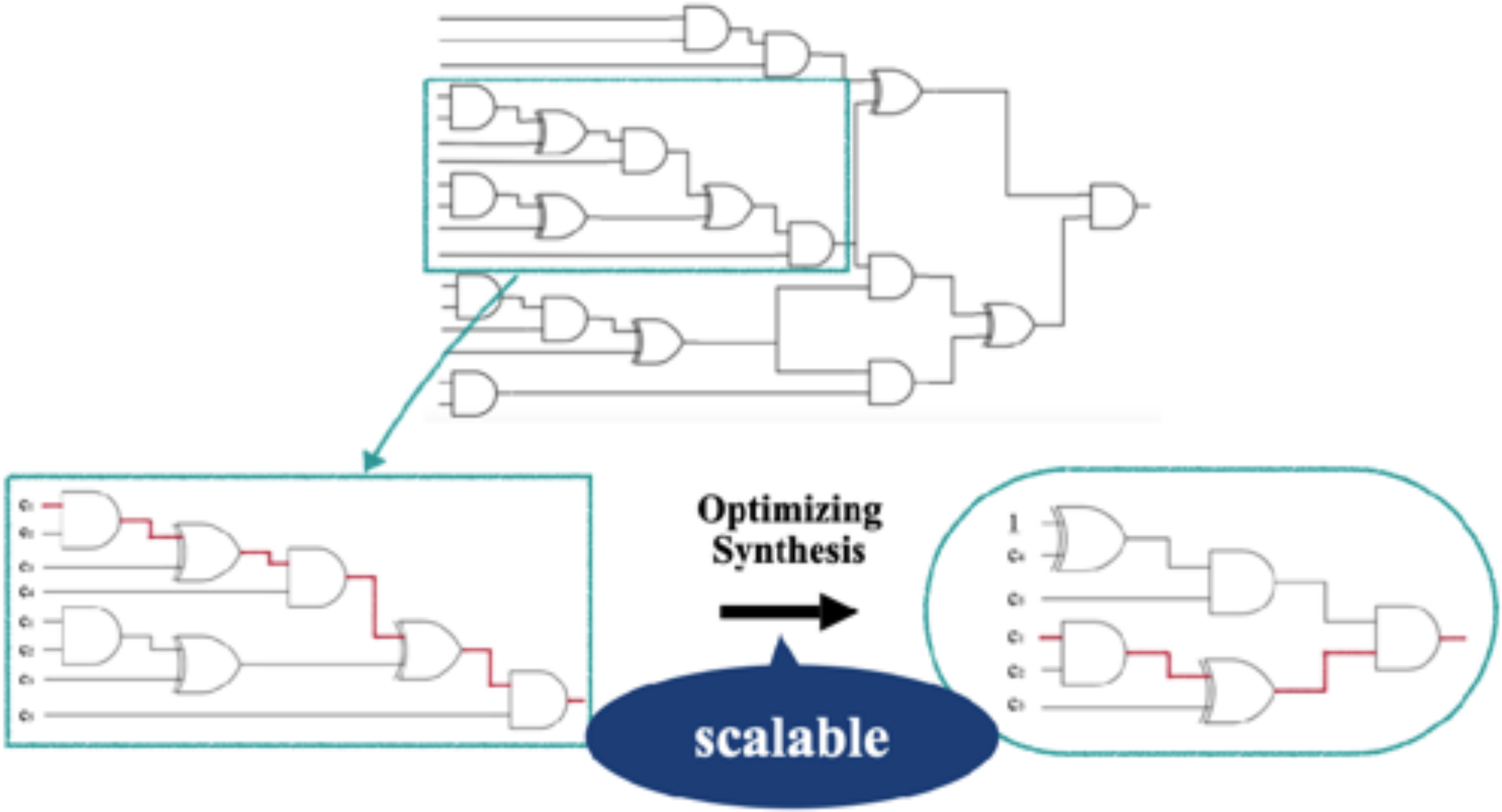
# 프로그램 합성 기반 최적화



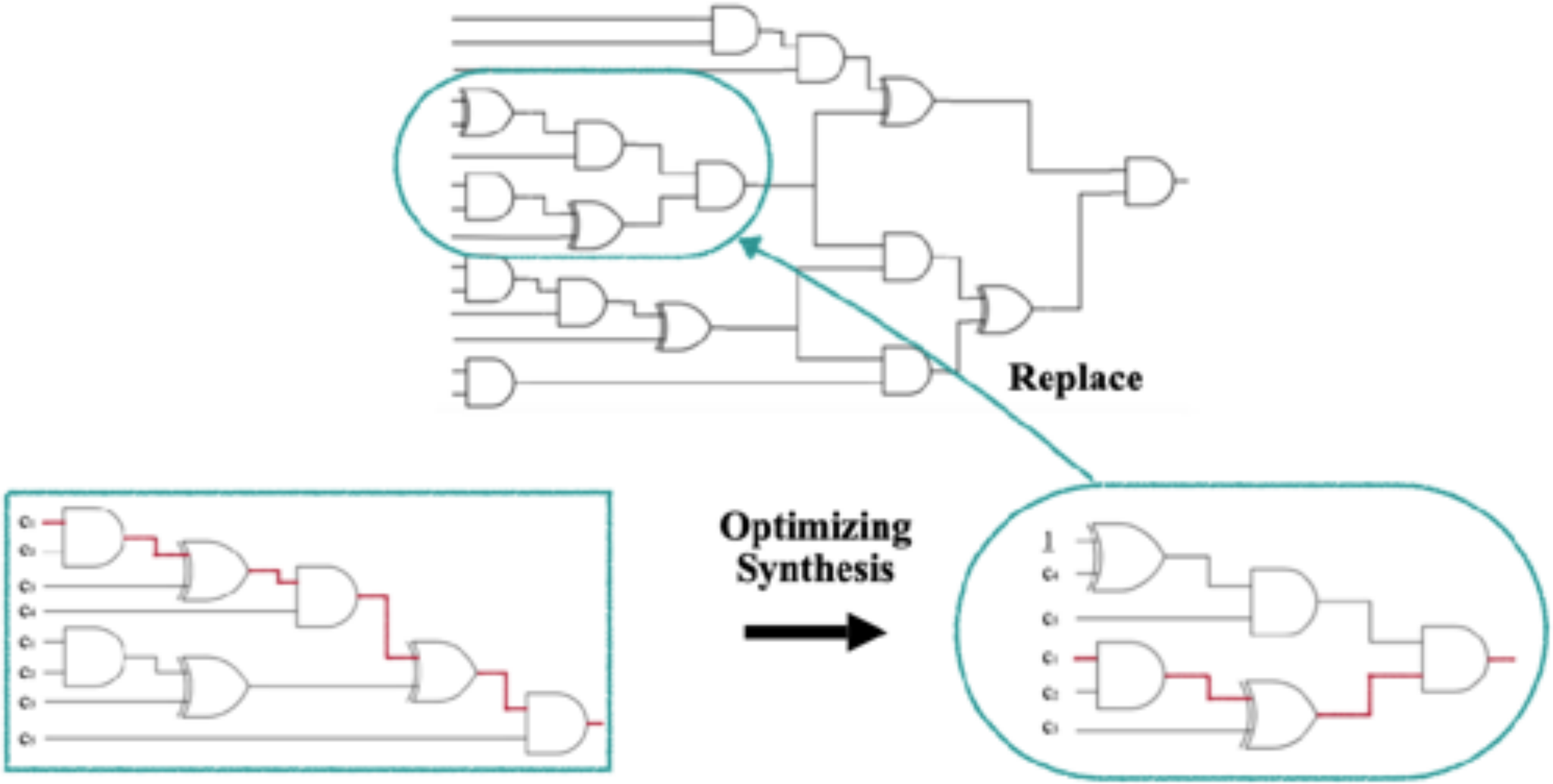
# 문제: 회로가 일반적으로 너무 큼



# 해결책 I: 분할정복



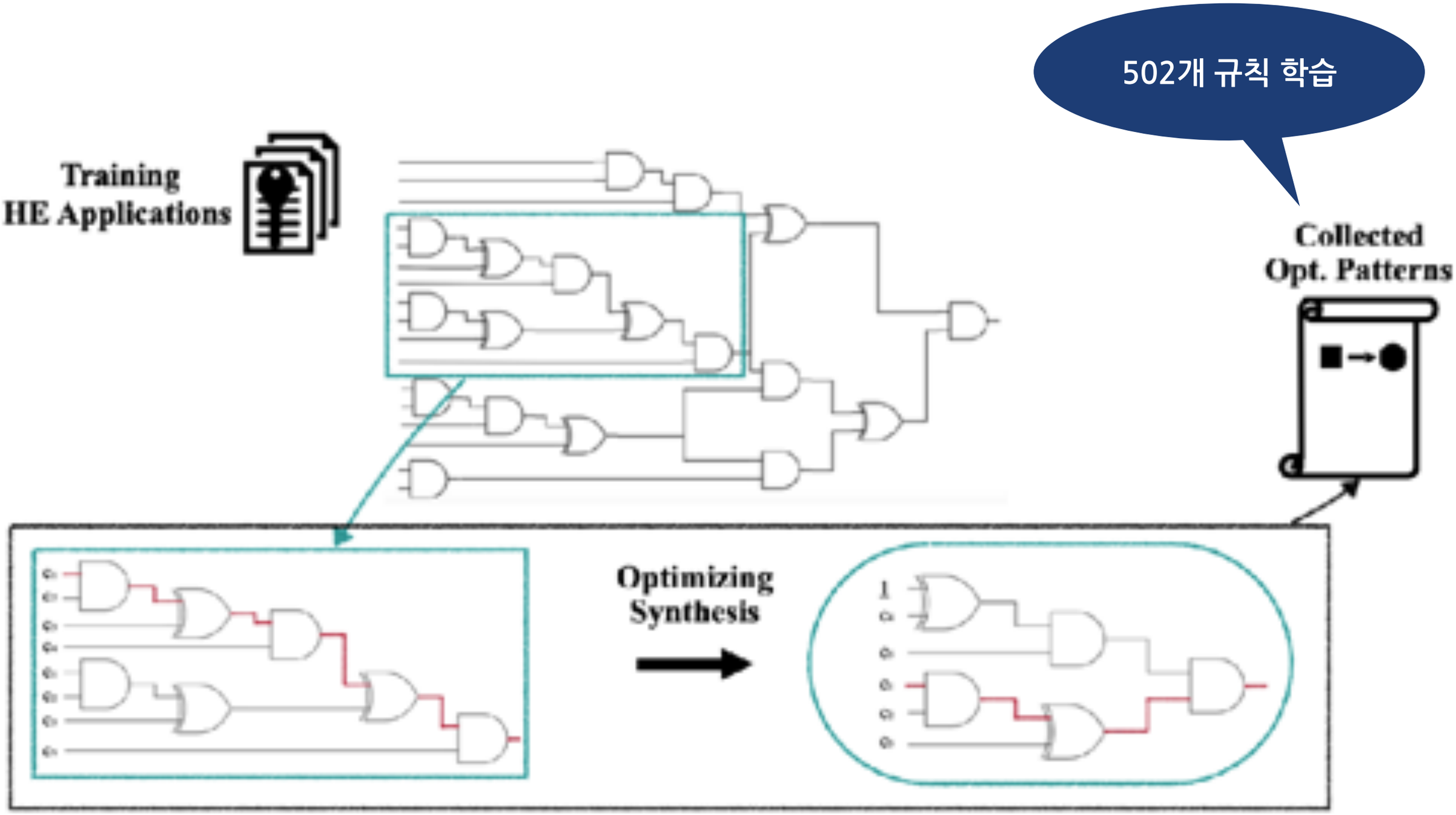
# 해결책1: 분할정복



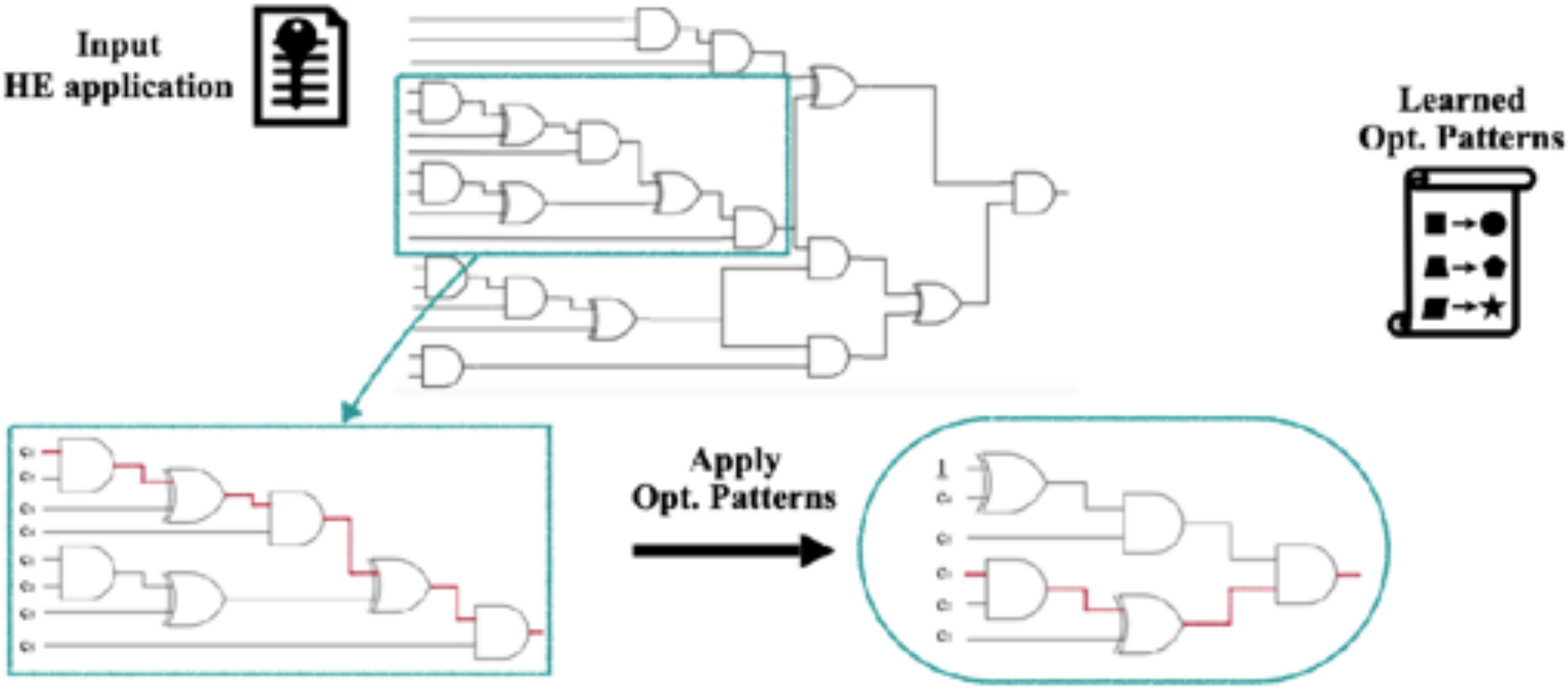
# 해결책2: 합성한 최적화 규칙 재사용

- 오프라인 학습
  - 성공적인 합성 사례들을 모아 최적화 규칙 사전 편찬
- 온라인 최적화
  - 새로운 입력회로에 대해 사전에 있는 최적화 규칙들 적용

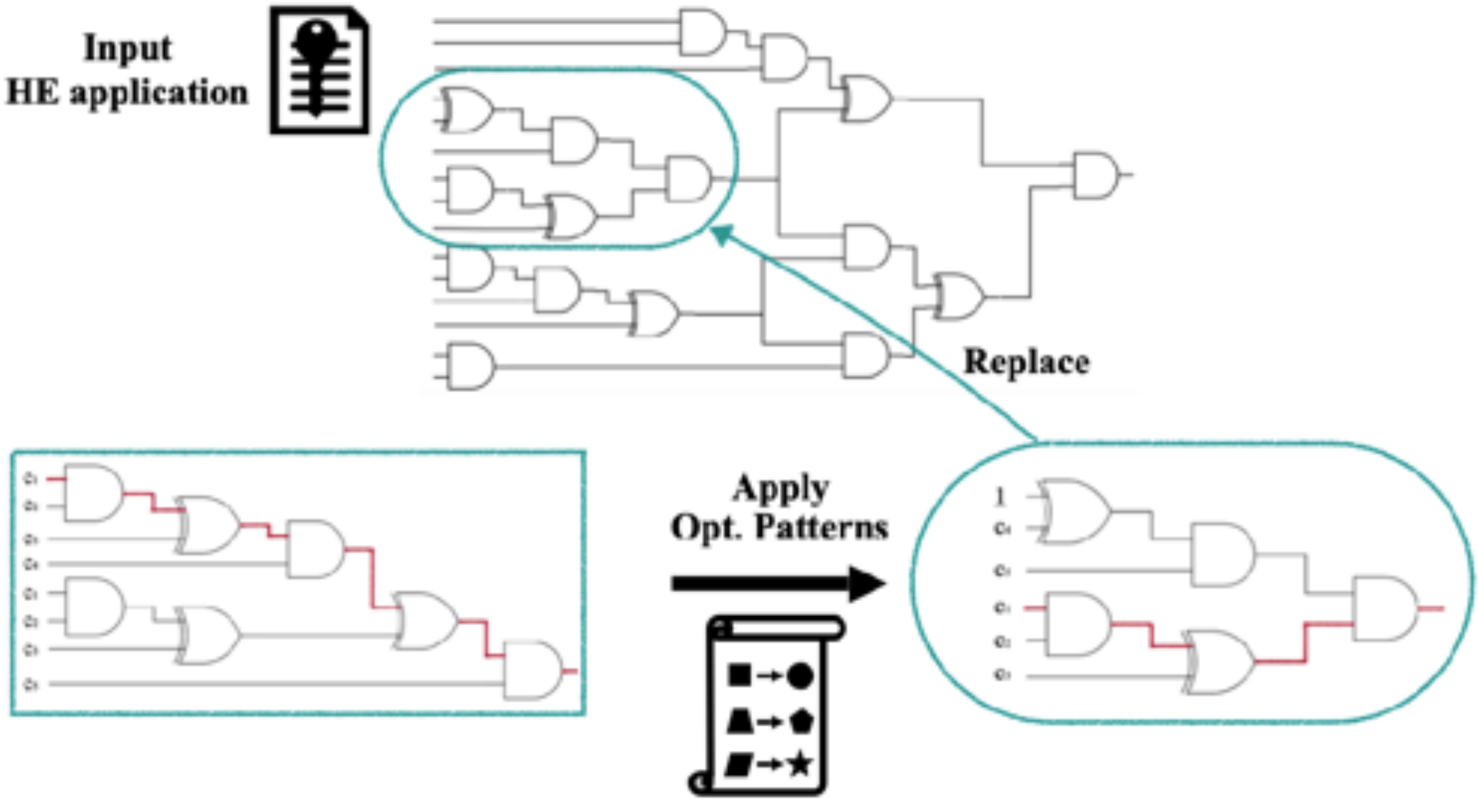
# 오프라인 학습


502개 규칙 학습
Training
HE Applications
Collected
Opt. Patterns
Optimizing
Synthesis

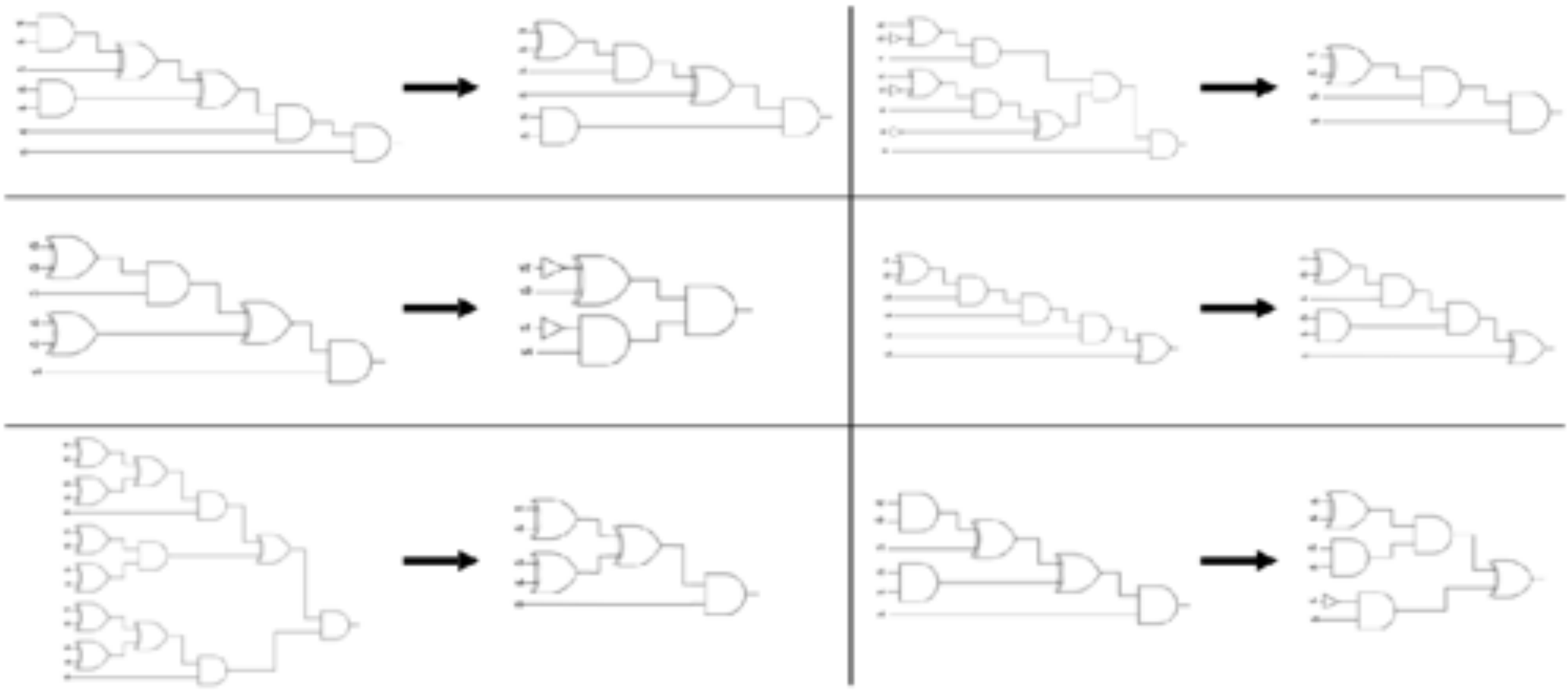
# 온라인 최적화


Input
HE application
Learned
Opt. Patterns
Apply
Opt. Patterns

# 온라인 최적화

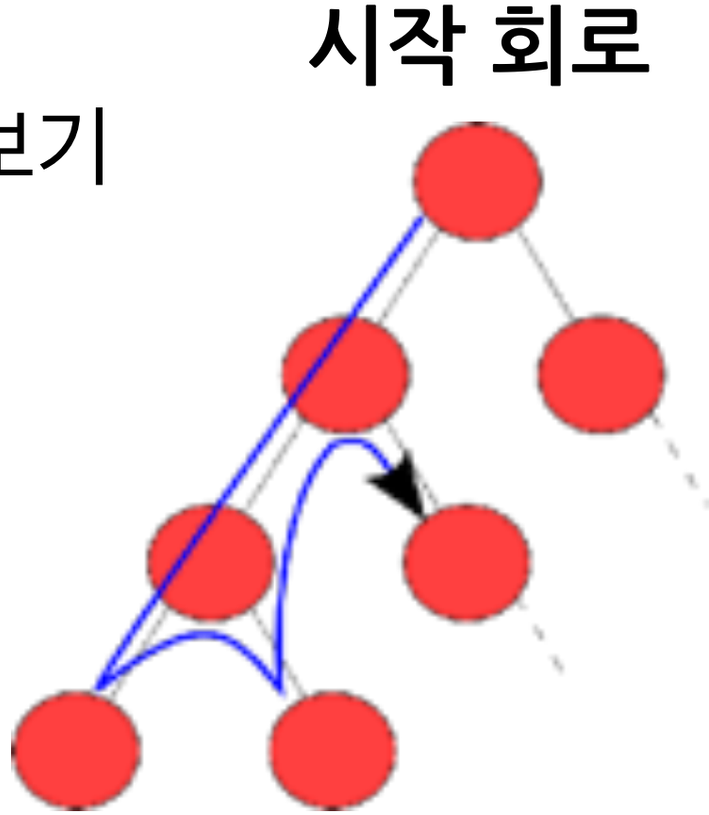


# 합성된 최적화 규칙들 예



# 더 잘할 수 없을까?

- 최적화 규칙을 임의의 순서로 적용하는 것은 최적에 못미치는 결과를 생성할 수 있음
  - 왜냐하면 서로 다른 최적화 규칙들이 서로가 사용되는 것을 방해할 수 있기 때문
- 해결책: 모든 가능한 순서 다 적용해보기



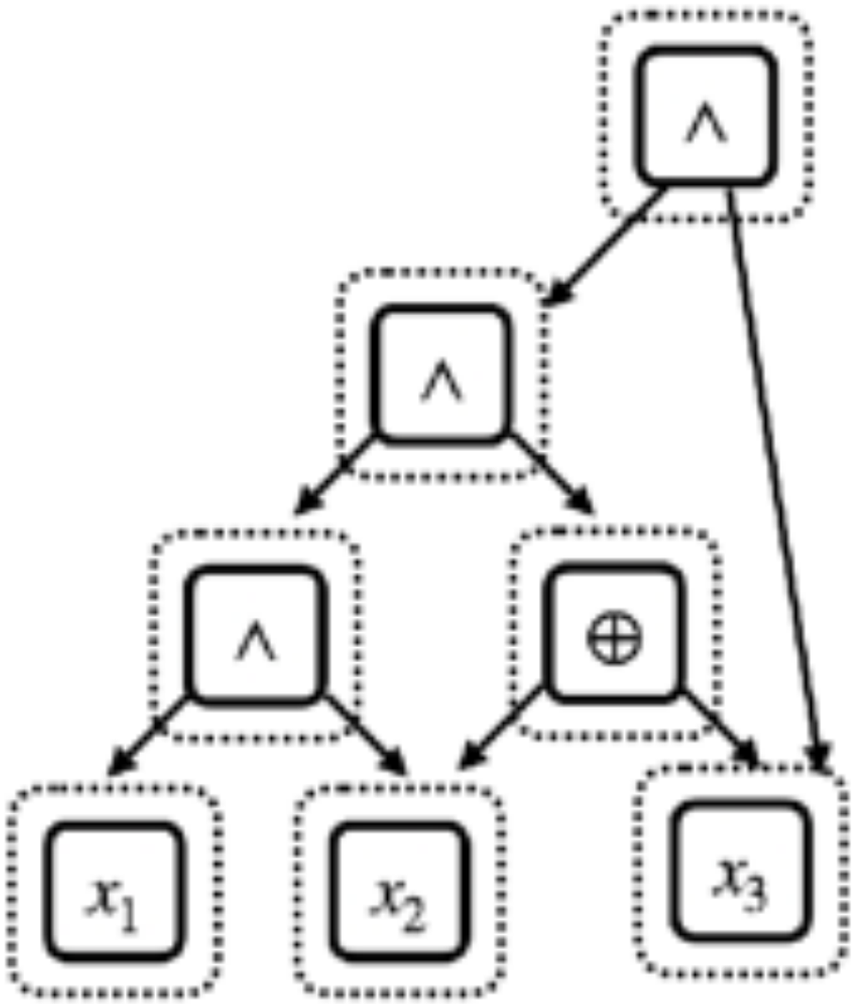
# 동일식 모두 모으기 (Equality Saturation)

- 규칙들을 모든 가능한 순서대로 적용해보고 모든 결과 저장
- 예: 원본 입력 회로 $((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$
- 쓸 수 있는 최적화 규칙들:

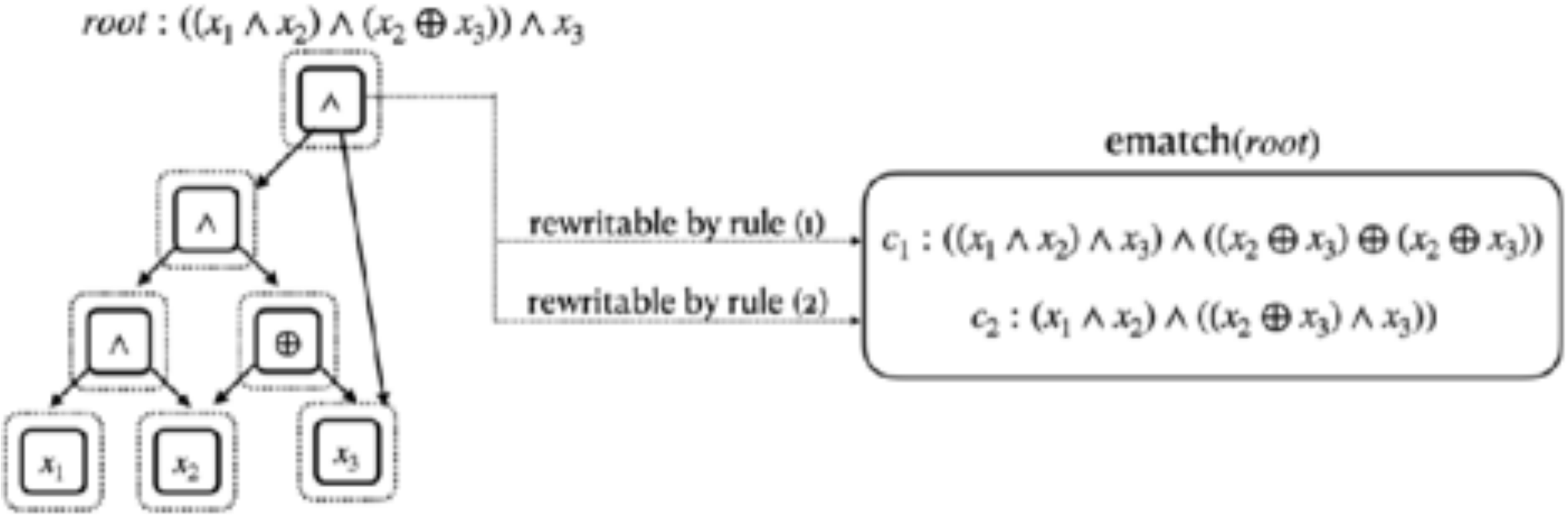
$$
\begin{array}{lrcl}
\text{rule (1)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & ((v_1 \wedge v_2) \wedge v_4) \wedge ((v_2 \oplus v_4) \oplus v_3) \\
\text{rule (2)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & (v_1 \wedge v_2) \wedge (v_3 \wedge v_4) \\
\text{rule (3)}: & (v_1 \oplus v_1) & \rightarrow & 0 \\
\text{rule (4)}: & (v_1 \wedge 0) & \rightarrow & 0
\end{array}
$$

# E-graph 를 이용한 완전탐색

$$root : ((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$$

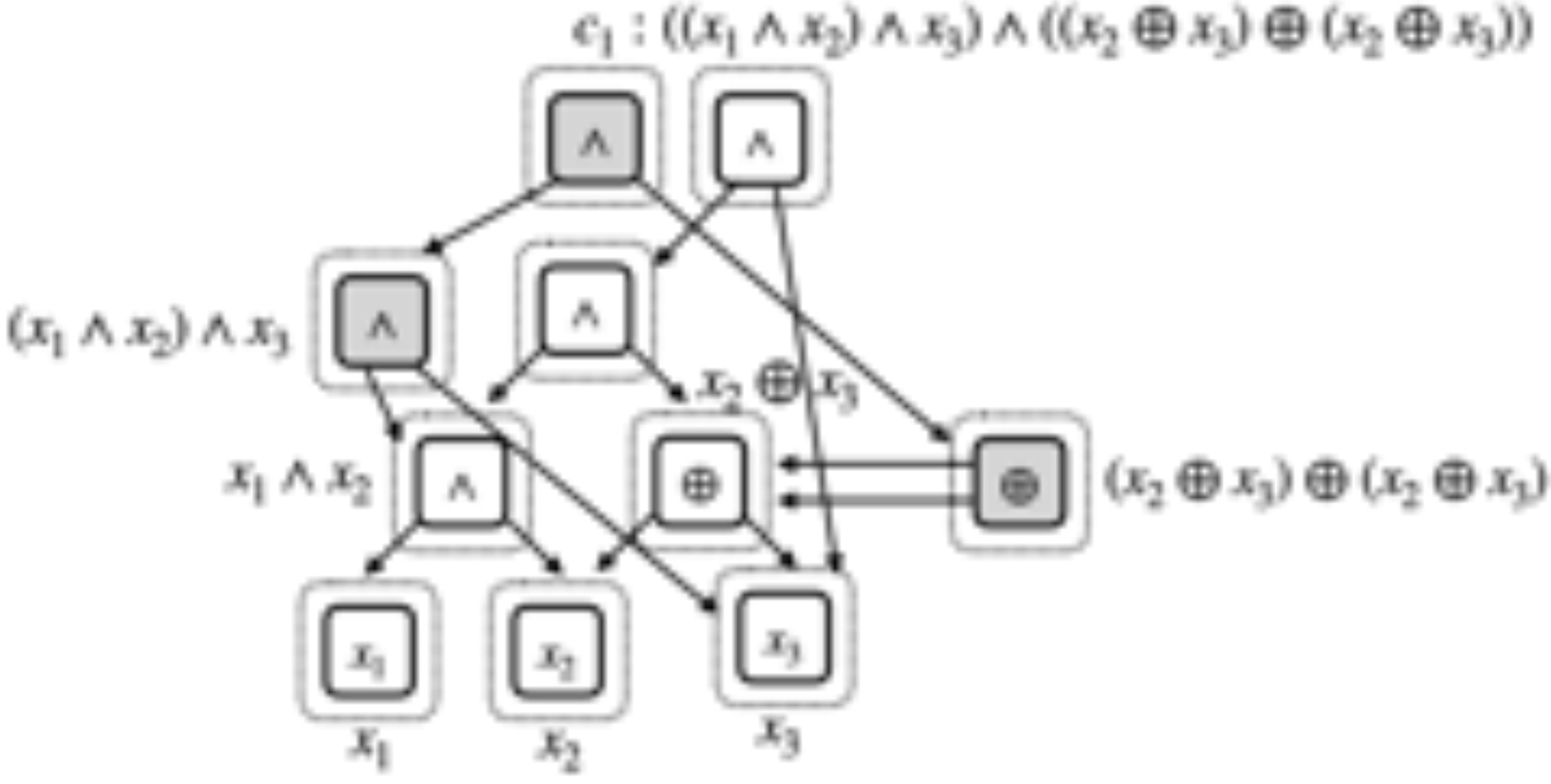


# E-graph 를 이용한 완전탐색



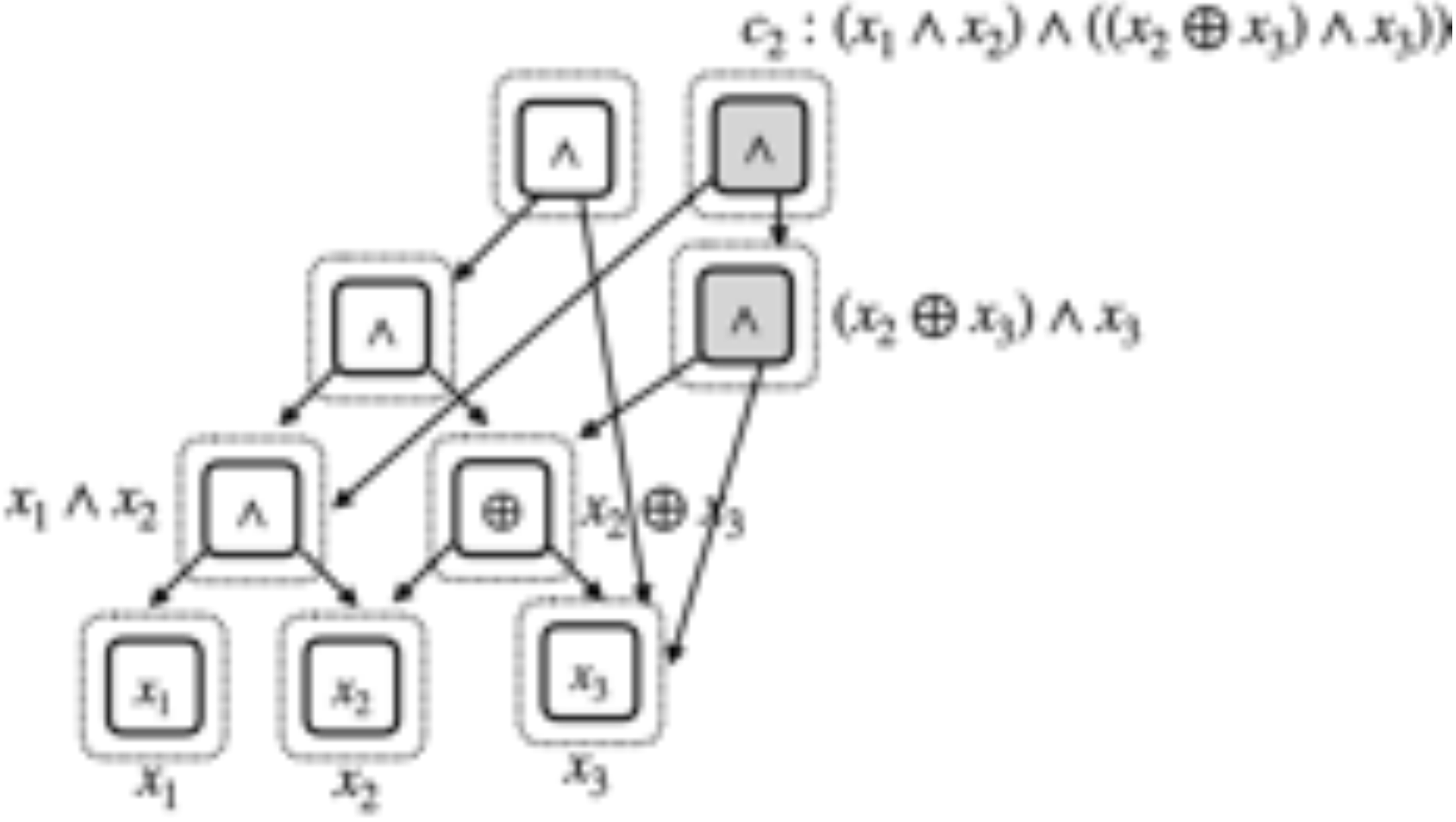
# E-graph 를 이용한 완전탐색

- 규칙 (I) 적용 (어두운색: 새로 추가된 노드)



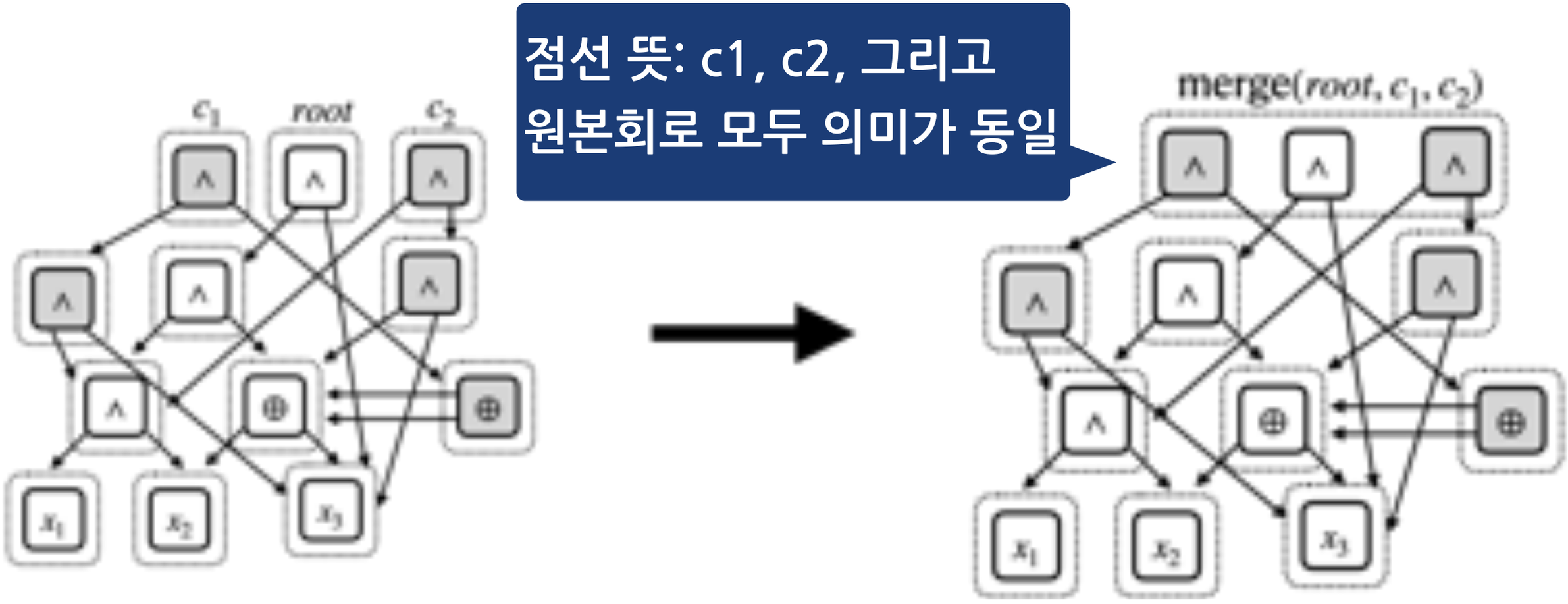
# E-graph 를 이용한 완전탐색

- 규칙 (2) 적용



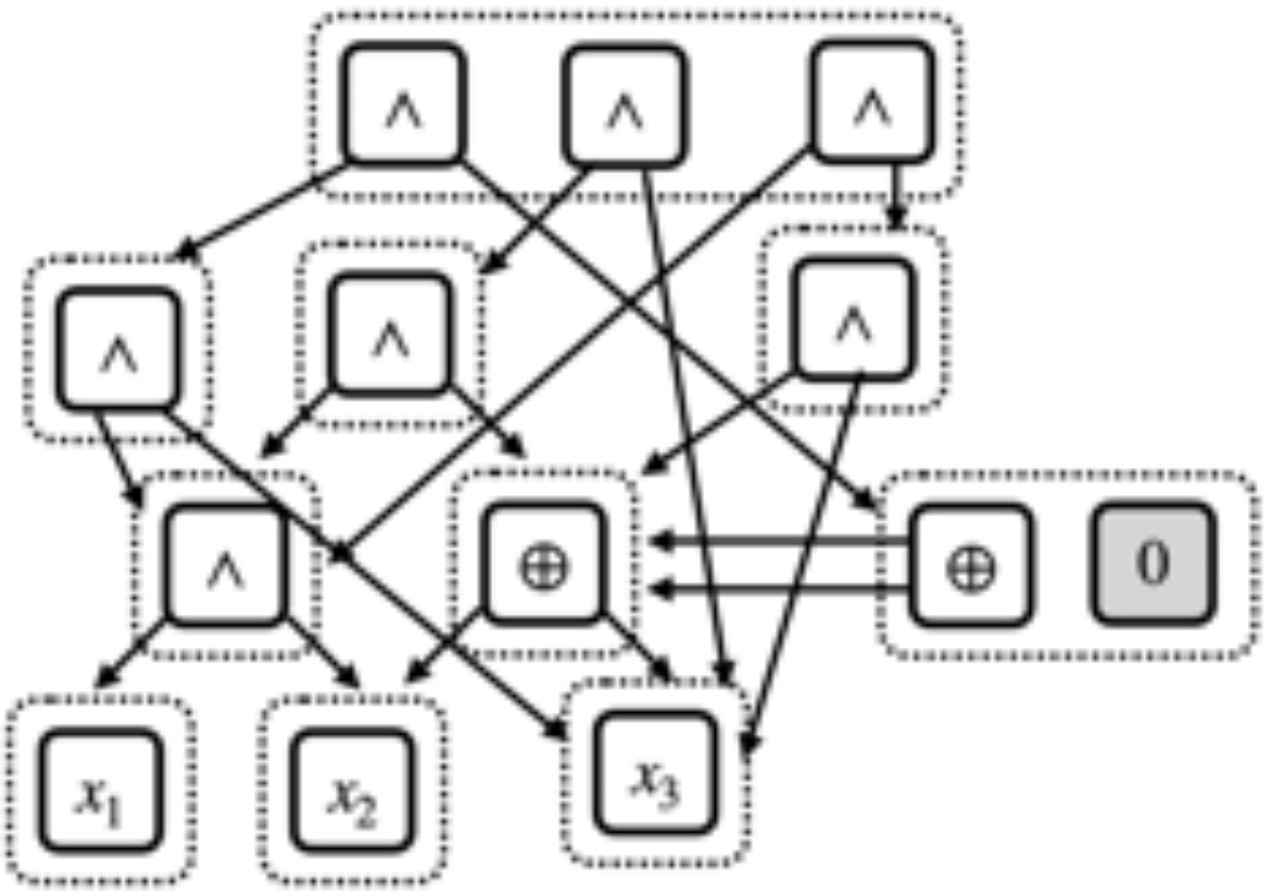
# E-graph 를 이용한 완전탐색

- 규칙 (1) 적용결과와 (2) 적용 결과 합치기



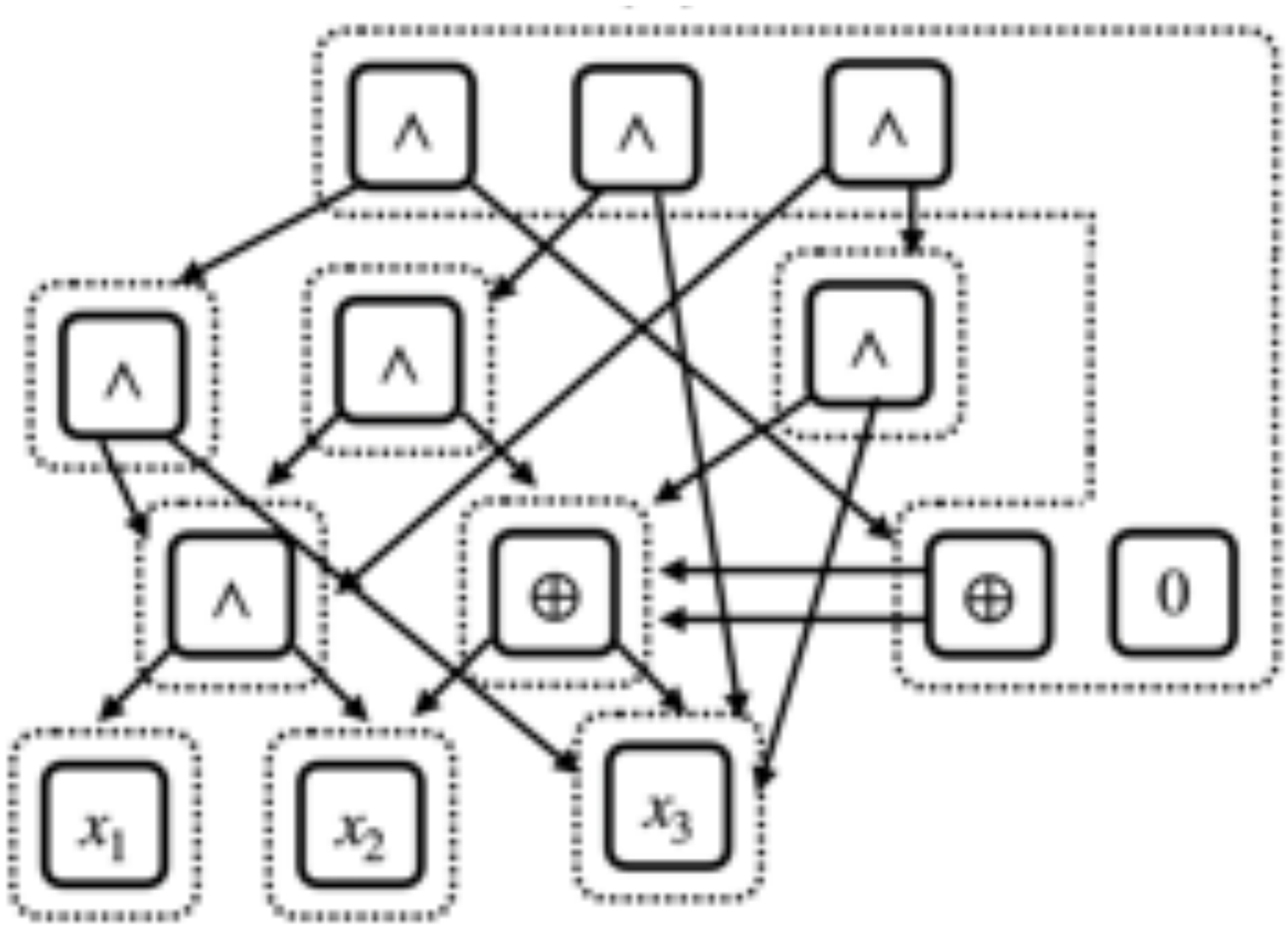
# E-graph 를 이용한 완전탐색

- 규칙 (3) $(v_1 \oplus v_1) \quad \rightarrow \quad 0$ 적용후, 결과 합치기



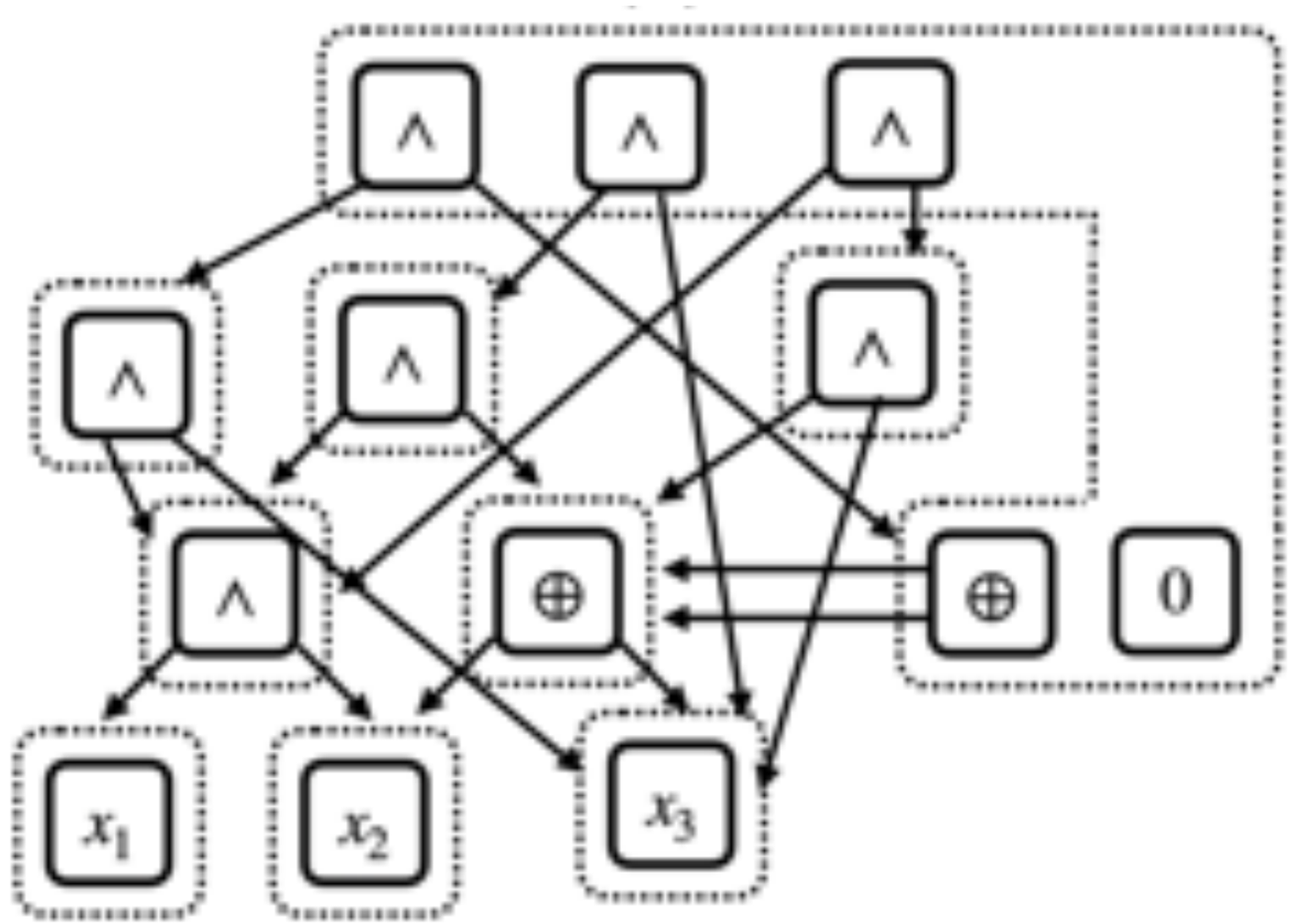
# E-graph 를 이용한 완전탐색

- 규칙 (4) $(v_1 \wedge 0) \quad \rightarrow \quad 0$ 적용 후 결과 합치기



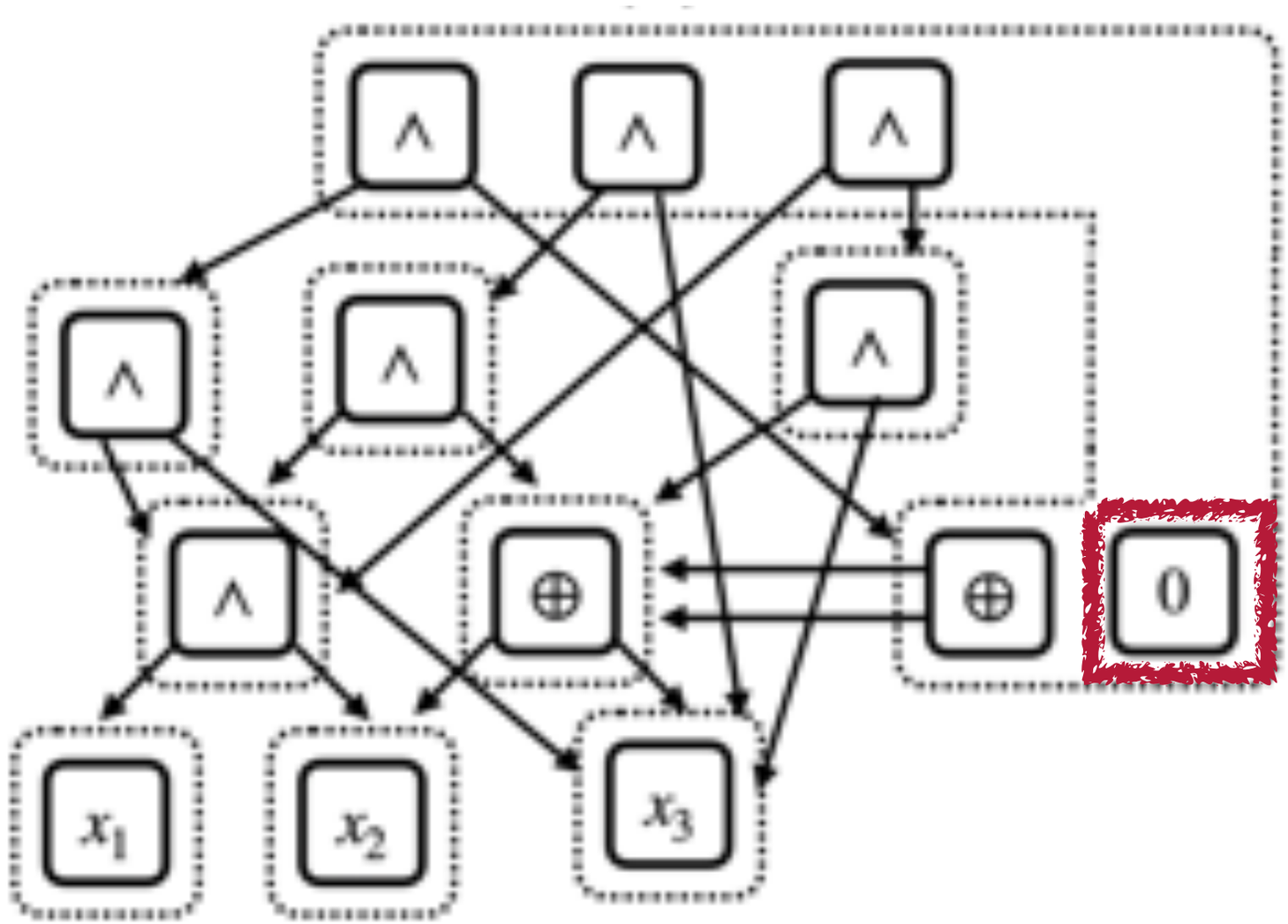
# E-graph 를 이용한 완전탐색

- 규칙적용이 더 이상 E-graph 변화 못시킴 => 끝!



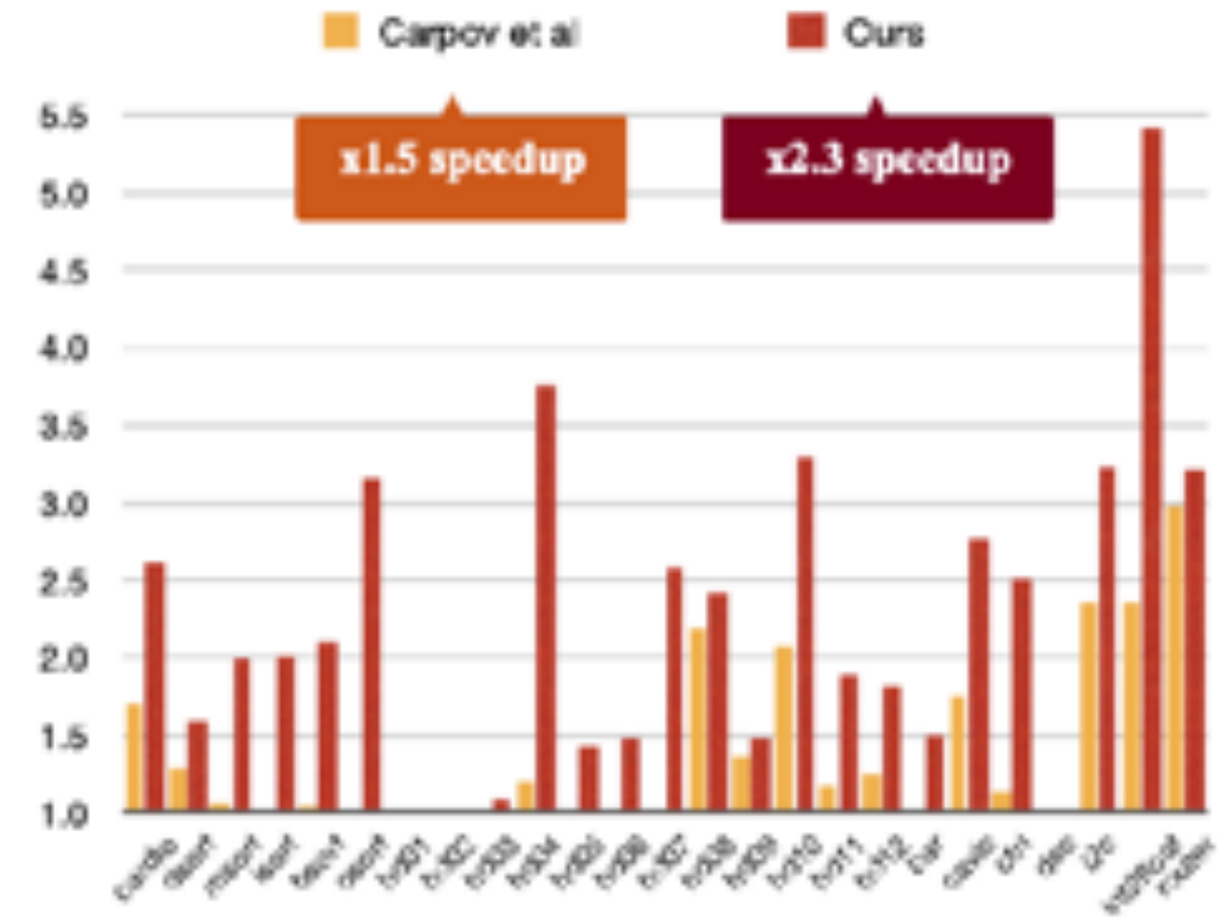
# E-graph 를 이용한 완전탐색

- 가장 곱셈깊이가 작은 회로 선택: **0**

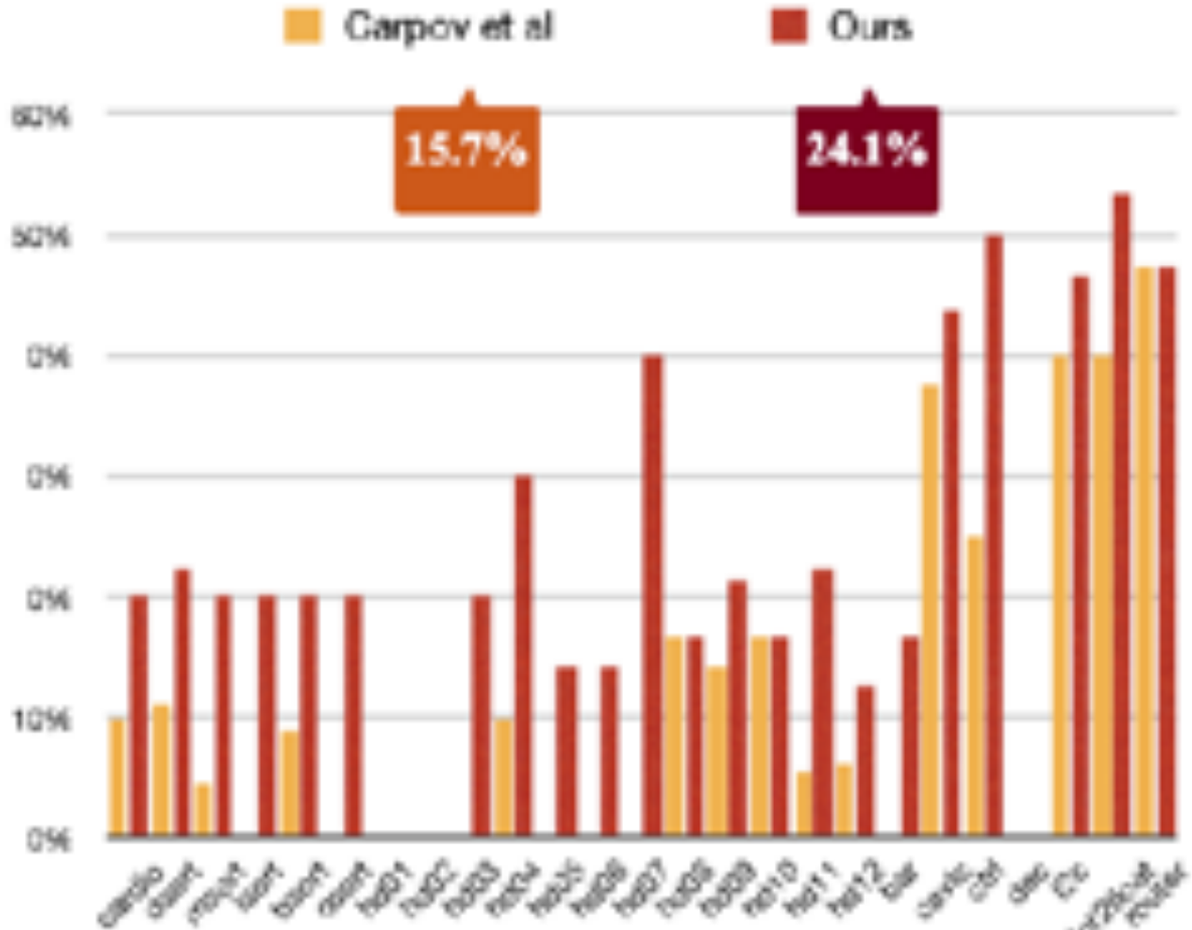


# 실험결과

수동 작성한 최적화 휴리스틱 기반




**Speedup**




**Depth Reduction**

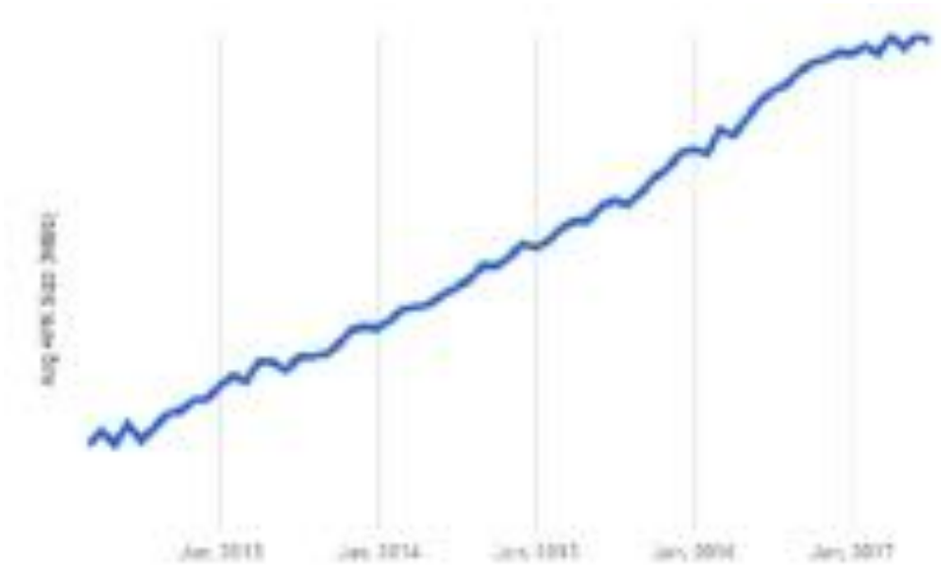
# 기존 연구결과(PLDI'20) 대비 개선효과

- 합성기를 EUSolver에서 더 나은 Duet 으로 교체
  - 학습한 최적화 규칙 186개 → 502개
- 동일식 모두 모으기
  - 최적화 효과: 총 25개 프로그램 중 19개 최적화 성공 (76%) → 22개 성공 (88%)
- 결론: 합성으로 최적화 규칙을 찾고 동일식 모두 모으기로 완전탐색시 최적화효과 극대화!

# Contents

- 프로그램 합성 소개
- 진행중인 합성 및 분석 연구들 소개
  - 합성기반 동형암호 프로그램 최적화
  - **안드로이드 인스턴트 앱 자동생성**
  - 재귀호출 프로그램 합성
  - 동형암호 기반 개인정보 유출없는 정적 분석

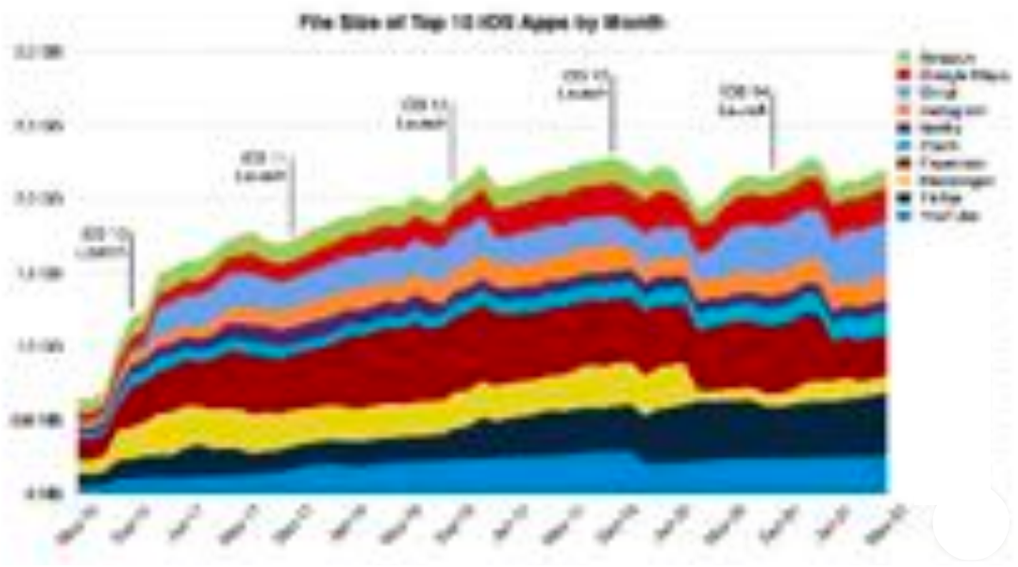
# 동기: 점점 증가하는 앱 크기



**Android**

2012년에서 2017년의 **5년** 간

**App 크기 5배 증가**



**iOS**

2016년에서 2021년의 **5년** 간

**App 크기 4배 증가**

출처 : Google IO18, SensorTower Inc.

# 대응책: 설치가 필요 없는 저용량 데모용 앱

App Clip

**용량 제한 15MB**


Google Play Instant

**용량 제한 10MB**


기존 App에 적용하기 위해서는

**용량 제한**에 맞춰서 **기능을 줄이고**

**코드와 자원의 제거**를

개발자가 **수동**으로 해 주어야 함

**➡ 고비용 작업 (안쓴다)**

# 우리의 제안: 자동 데모 앱 생성기

- **입력**:
  - 데모 App에 포함될 기능을 명세 — 사용 시나리오 (이벤트 시퀀스)
  - 제한 용량 (예: 15 MB)

- **출력**: 성공시 데모 App
  - 용량 제한 이하
  - 사용 시나리오를 재현 가능
  - 시나리오에 없는 이벤트에도 가능한 안죽고 잘 버팀

# Contents

- 프로그램 합성 소개
- 진행중인 합성 및 분석 연구들 소개
  - 합성기반 동형암호 프로그램 최적화
  - 안드로이드 인스턴트 앱 자동생성
  - **재귀호출 프로그램 합성**
  - 동형암호 기반 개인정보 유출없는 정적 분석

# 동기: 어려운 함수형 언어

- 한양대학교 ERICA 학부 프로그래밍언어론 수업
  - 프로그래밍 언어의 다양한 원리들 학습. 함수형 언어 OCaml 로 진행
- 수강 후기

수업 내용은 정말 알차서 좋았는데 과제가 너무 어려웠습니다

과제가 너무 어려웠습니다

More examples of OCaml coding

The homework is way too hard compared to the exam, please lower the difficulty of the homework

Actually teach us OCaml.

# OCaml 합성기

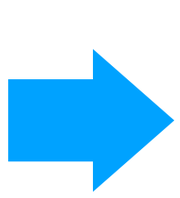
- 목표: 입출력 예제로부터 OCaml 프로그램 합성하도록.
  - 다양한 예제에 대해 합성된 코드로부터 OCaml 이해
    - OCaml 자체에 매몰되지 않고 핵심 콘텐츠에 집중
- 도전과제: 함수형 언어에서 흔한 재귀호출 — 합성하려는 함수가 합성하려는 함수 그 자체를 사용

# 합성에 사용하는 OCaml Subset

$$
\begin{array}{lcl}
P & ::= & \text{rec } f(x : \tau_1) : \tau_2 = e \\
e & ::= & x \\
& | & e_1\ e_2 \\
& | & . \\
& | & \kappa(e_1, \cdots, e_{a(\kappa)}) \\
& | & \kappa^{-i}(e) \\
& | & \text{match } e \text{ with } \overline{\kappa_i\ \_ \longrightarrow e_i}^{k}
\end{array}
$$

# 데모

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

synth list -> nat satisfying

[Nil] -> 0,
[Cons(0,Nil)] -> 1,
[Cons(0,Cons(0,Nil))] -> 2,
```



```
let rec (f : (list -> nat)) =
  fun (x:list) ->
    match x with
    Nil(_) ->
      O(Un_Nil(x))
    Cons(_) ->
      S((f (Un_Cons(x)).1))
```

# 향후 계획

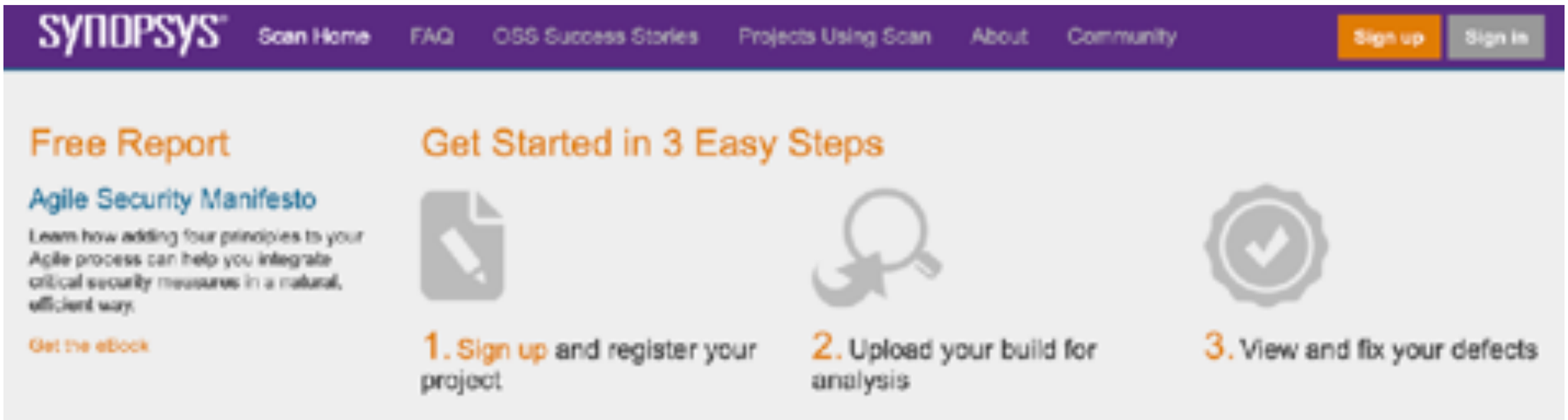
- 양방향 탐색 알고리즘 기반 프로그램 합성 기법을 재귀함수 합성에 적용

- Woosuk Lee, Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis, POPL 2021.

# Contents

- 프로그램 합성 소개
- 진행중인 합성 및 분석 연구들 소개
  - 합성기반 동형암호 프로그램 최적화
  - 안드로이드 인스턴트 앱 자동생성
  - 재귀호출 프로그램 합성
  - **동형암호 기반 개인정보 유출없는 정적 분석**

# 정적 분석 서비스 (Static analysis-as-a-service)





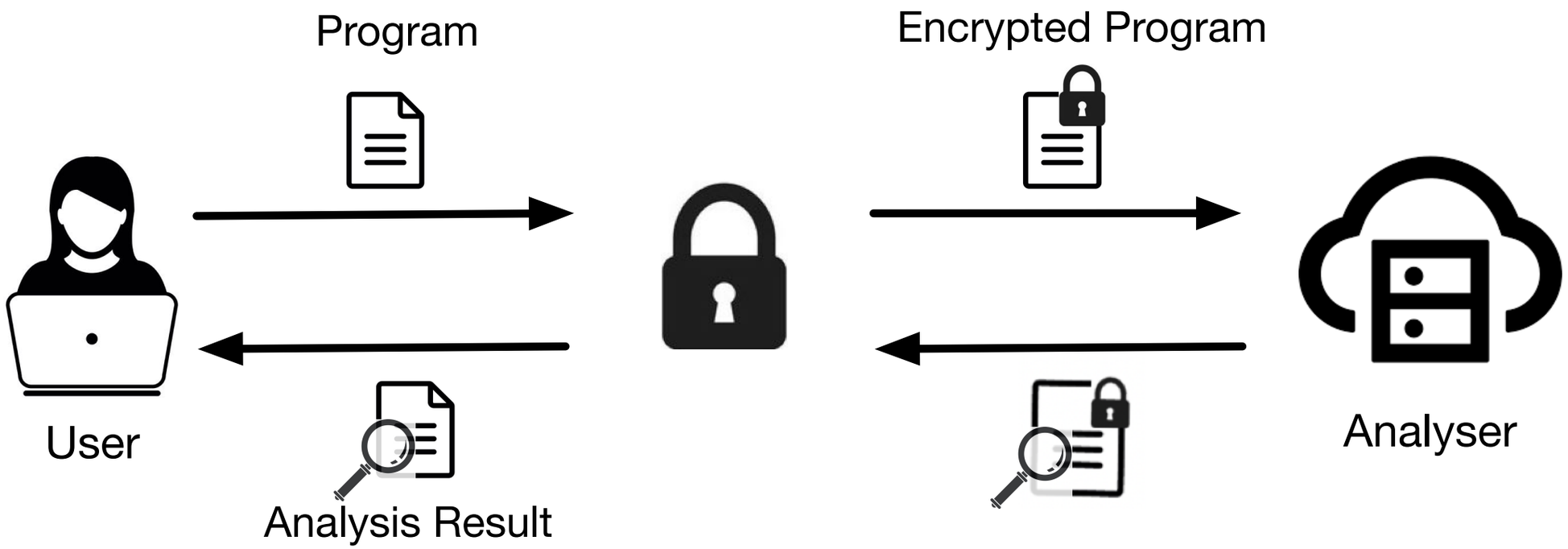
# 문제: 소스코드 지적재산권 보안

- 장점: 클라우드의 연산능력 사용, pay-per-line 등 유연하고 저렴하게 분석기 사용가능

- 단점: 분석 의뢰하는 대상코드는 서비스 제공자에 노출됨

- 그게 싫으면, 분석기 구매 후 사용 (On-premise)
  - 문제: 비쌈! (예: Coverity 10명 미만 1년 사용권 3360만원)
  - 왜 비싼가? ⇒ 분석기에 대한 지적재산권

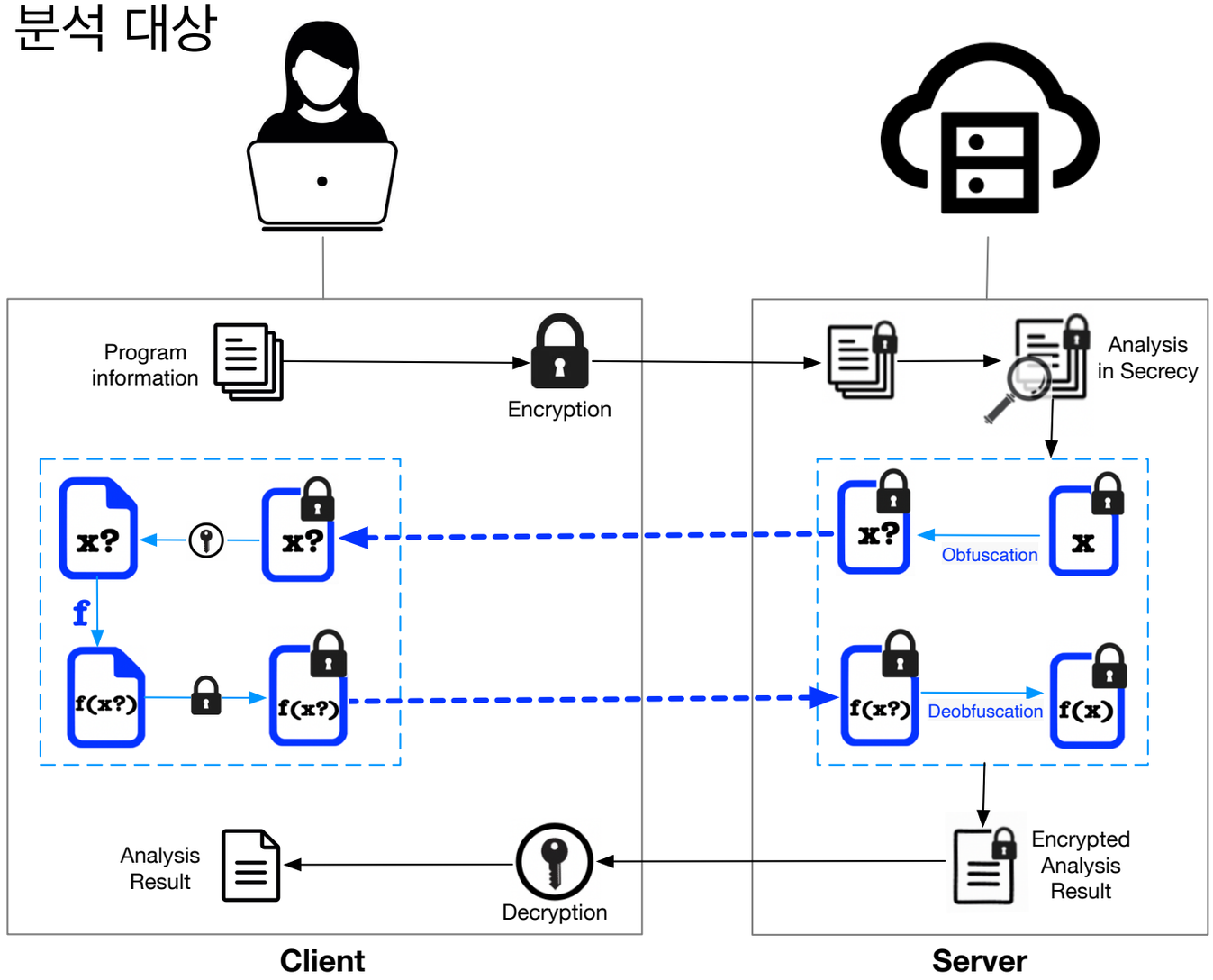
# 가능한 대안: 동형암호 사용




- 문제: 동형암호 연산 비용이 매우 비쌈 (300줄 분석에 이틀)

# 우리의 대안: 동형암호 + 다자간 계산 (Secure Multi-Party Computation)

- 동형암호 연산 중 서버에게 어려운 부분 클라이언트에 역으로 외주주기
  - 그러나 클라이언트에겐 매우 쉬운 문제 (역행렬 계산, 영행렬 체크)
- 상호작용 중 대상 프로그램 및 분석기에 대한 정보 노출 X
- Datalog 로 쓰인 정적 분석 대상



# 실험결과

- 소형 C 벤치마크 프로그램 대상 포인터 분석
- 동형암호에만 의존하던 예전 연구보다 월등한 성능

| Programs | | | SecureDL | | Previous Approach | |
|---|---|---|---|---|---|---|
| Name | N | LoC | Client | Server | Client | Server |
| bind-1 | 92 | 1300 | 14.2s | 2h 8m 22s | timeout | |
| bind-2 | 104 | 1634 | 16.1s | 1h 8m 53s | timeout | |
| bind-3 | 26 | 344 | 4.2s | 5m | timeout | |
| bind-4 | 55 | 555 | 8.7s | 47m 39s | timeout | |
| sm-1 | 56 | 884 | 8.5s | 50m 22s | timeout | |
| sm-2 | 39 | 570 | 6.1s | 36m 45s | timeout | |
| sm-3 | 52 | 592 | 8.2s | 1h 56m 15s | timeout | |
| sm-4 | 34 | 620 | 5.4s | 15m 29s | timeout | |
| sm-5 | 43 | 691 | 6.7s | 57m 31s | timeout | |
| sm-6 | 13 | 290 | 2s | 25s | 46s | 6h 21m |
| sm-7 | 96 | 1206 | 14.9s | 1h 33m 1s | timeout | |
| ftp-1 | 21 | 336 | 3.2s | 1m 9s | timeout | |
| ftp-2 | 28 | 996 | 4.3s | 1m 51s | timeout | |
| ftp-3 | 39 | 642 | 6.1s | 12m 22s | timeout | |

# 그 외 진행 중 연구들

- 요약해석 기반 프로그램 합성 가속화 (with 서울대)
- 프로그램 합성 기반 자동 프로그램 오류 수정

감사합니다!