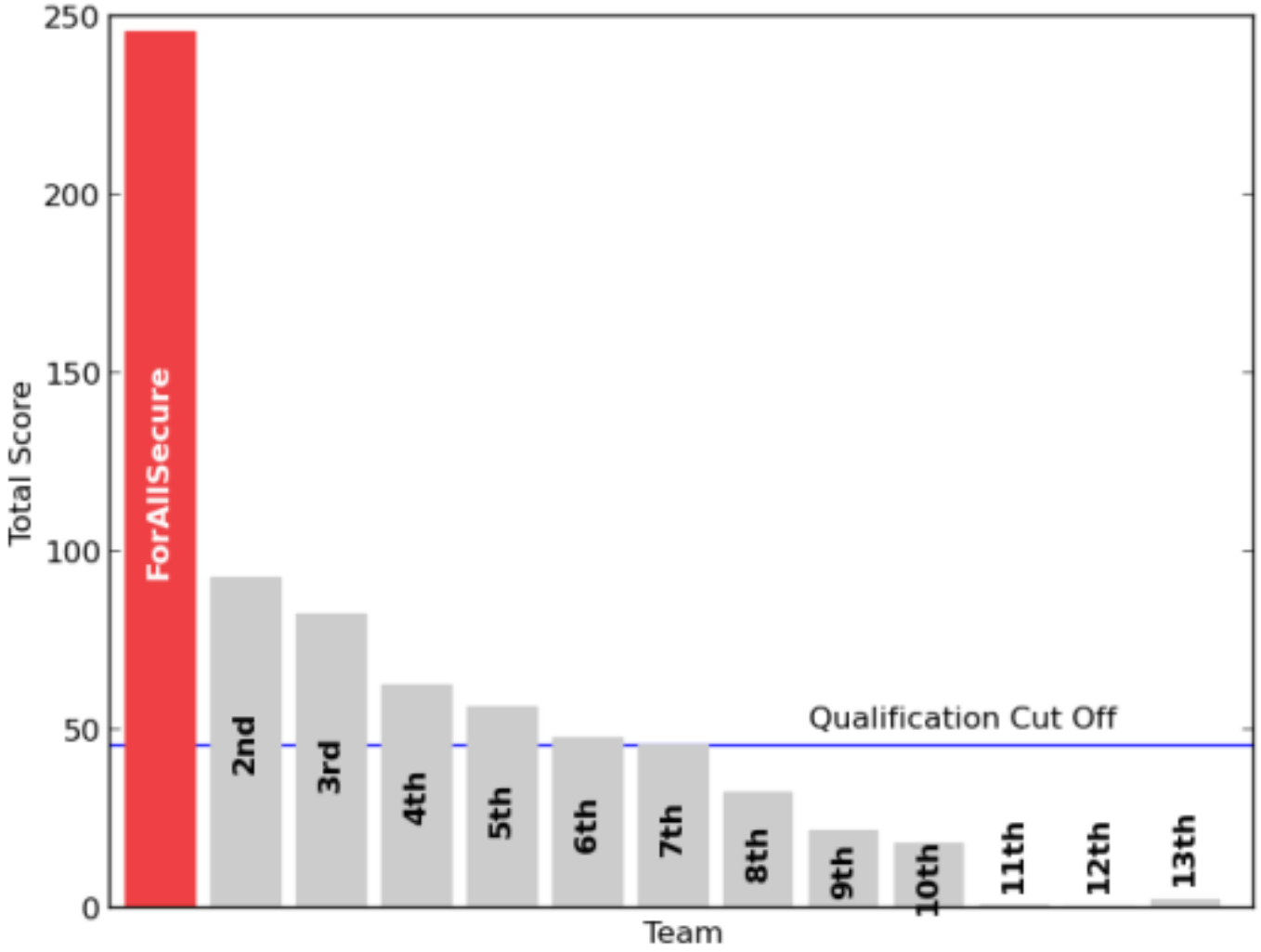
바이너리 분석을 통한 자동 익스플로잇 생성: 과거, 현재, 그리고 미래

차상길
카이스트

Cyber Grand Challenge (CGC)

Hacking competition between computers



Final scores for teams playing in the DARPA CQE

Figure taken from http://blog.forallsecure.com/2016/02/09/unleashing-mayhem/

해킹의 자동화

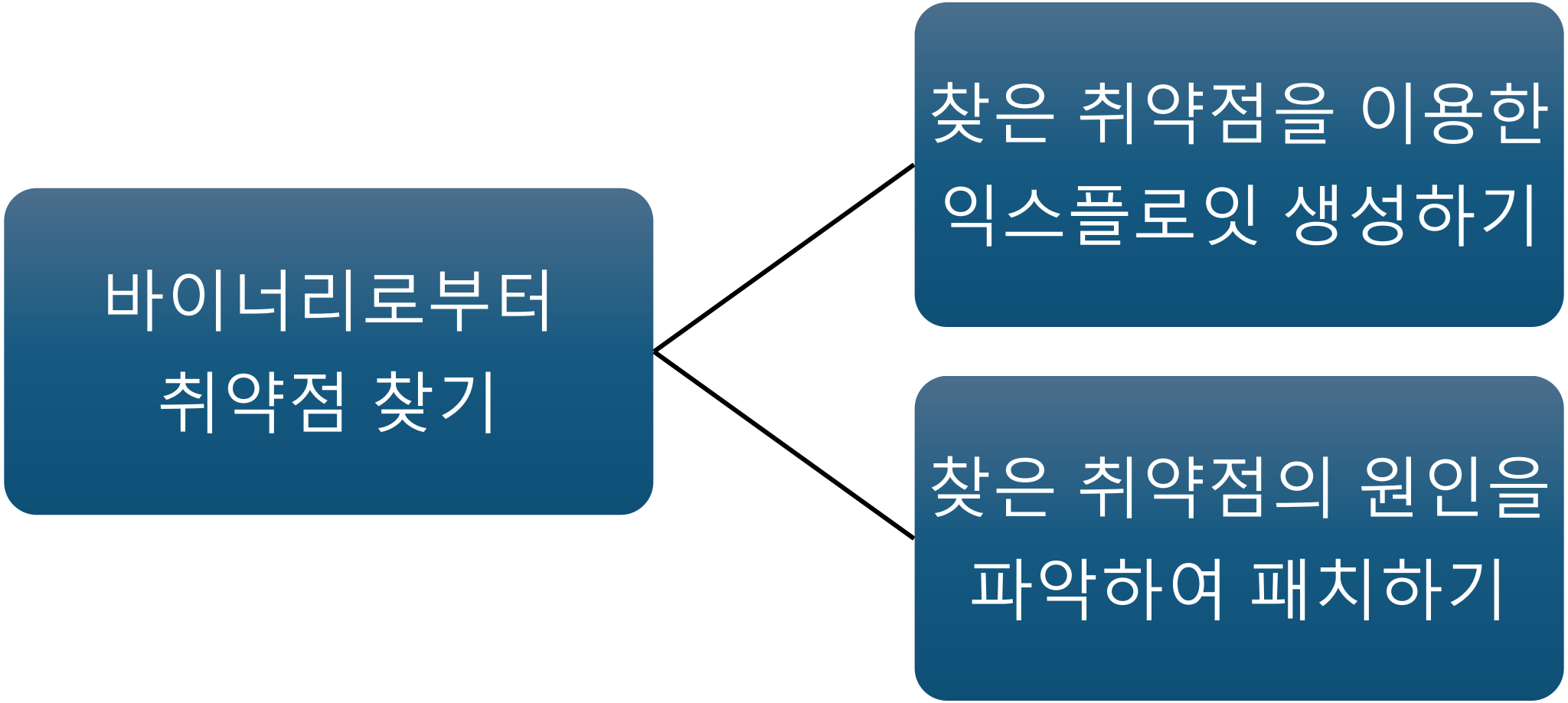
왜 해킹을 자동화 하는가?

1. 멋있어서
2. 역공학은 개개인의 능력에 의존적임 (대규모의 분석에 취약)
3. 버그가 너무 많고 그 중에 보안에 중요한 버그는 몇 안됨
4. 보안 취약점에 빠른 대응 가능

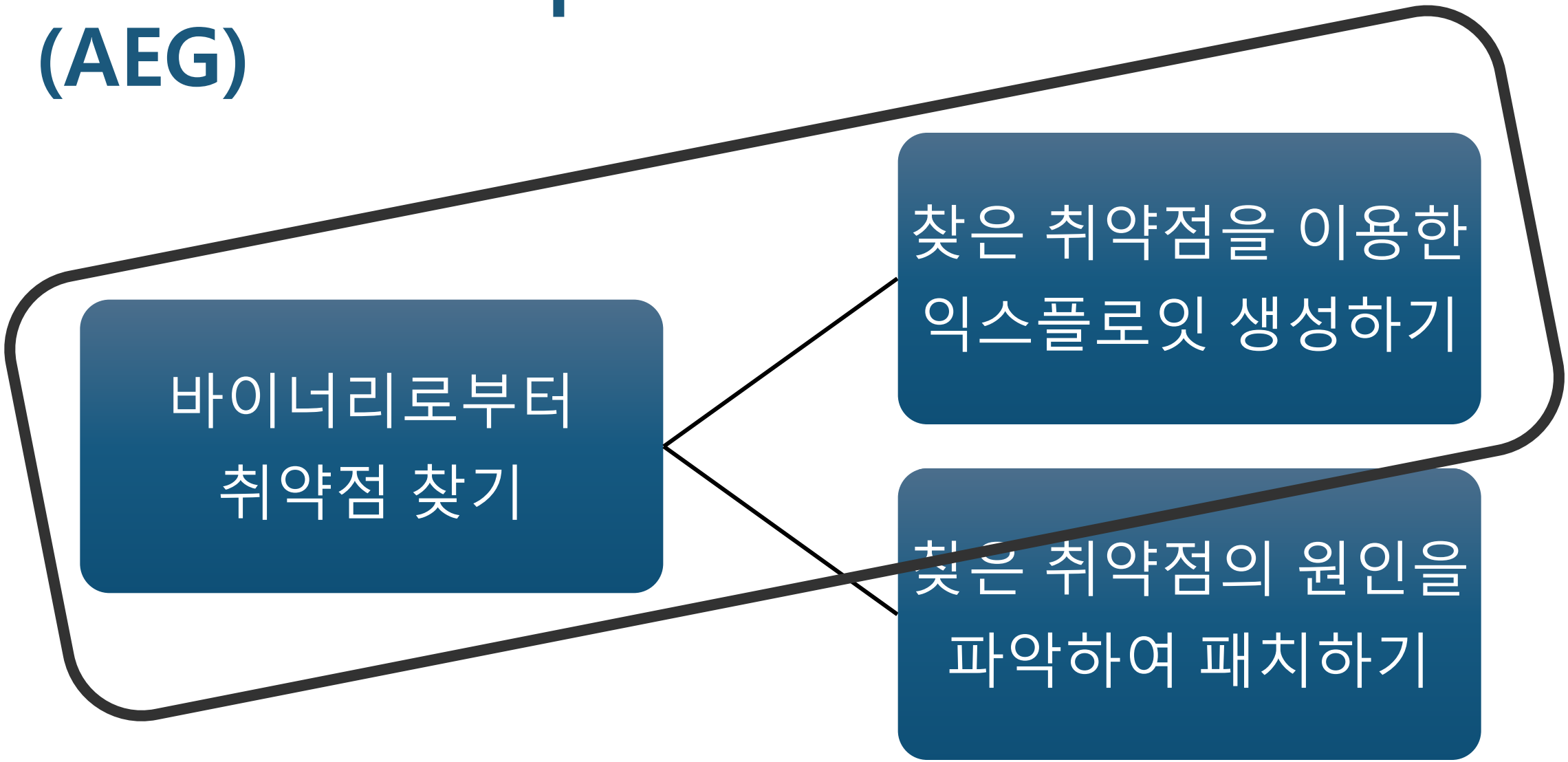
知彼知己 百戰不殆

- 손자병법

자동화된 해킹의 3요소



Automatic Exploit Generation (AEG)



AEG 의 시작

2005, Ganapathy et al.

Automatic Discovery of API-Level Exploits, ***ICSE 2005***

최초로 AEG 문제를 Verification 문제로 간주

Verification with a Twist

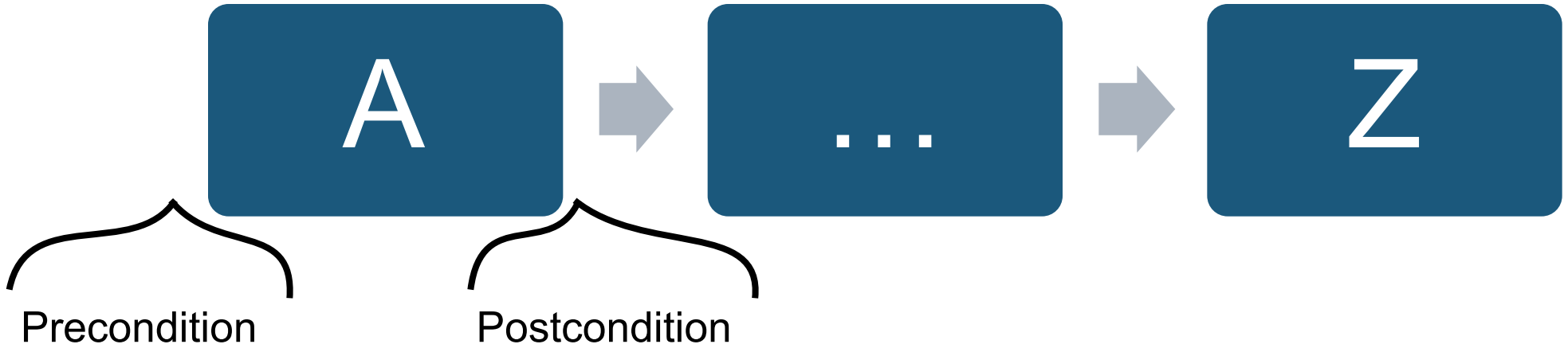
Unexploitability property

Automatic
Exploit
Generation

API-Level Exploits

(예제) 특정 파일에 대하여, 소유권 없이 Read/Write 권한을 갖게 되는 API sequence를 찾아라.

API model을 기반으로 하는 bounded model checking

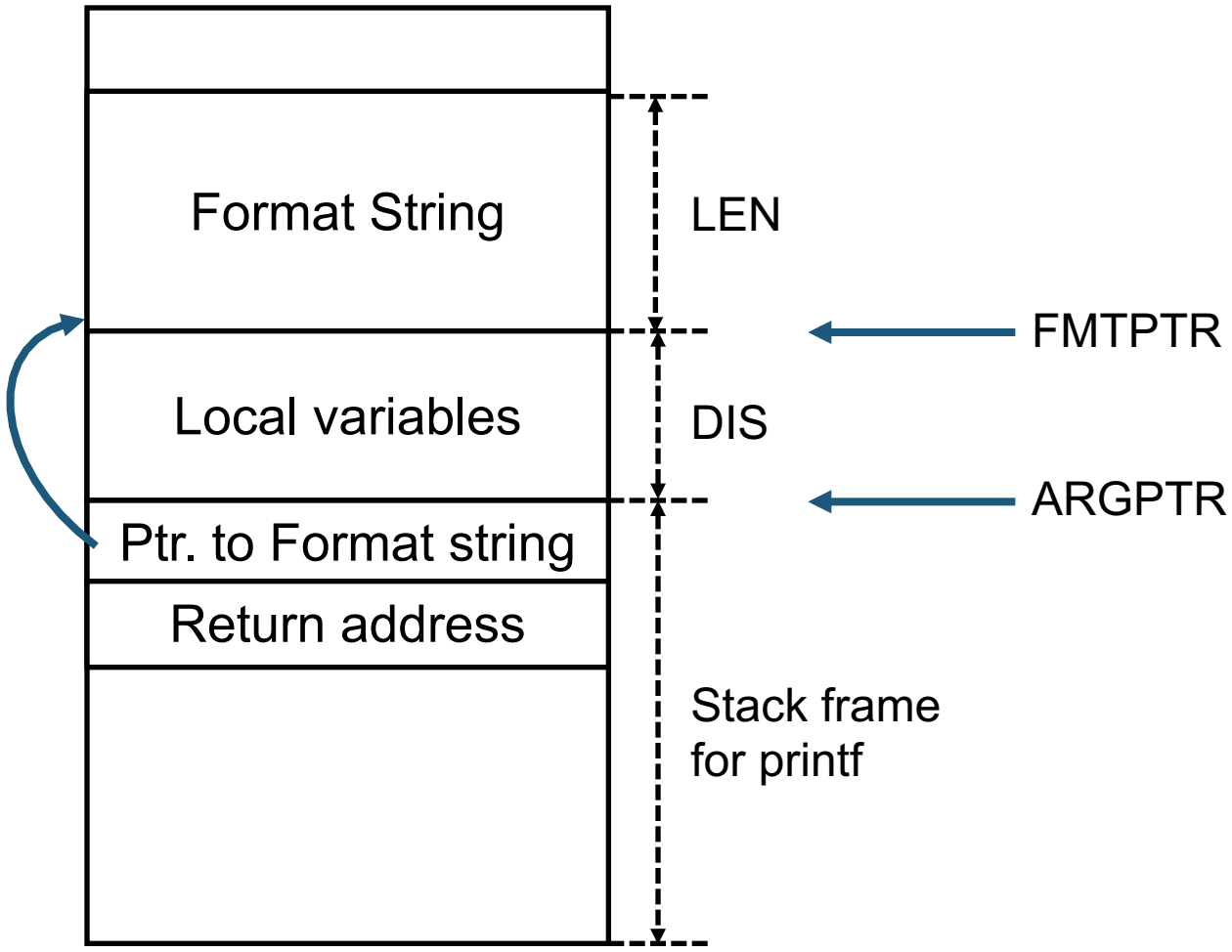


Format String Exploit Generation

- Format string bug를 찾았다는 가정하에 exploitability를 판단하고 실제 counter example (= exploit) 생성

- 포맷의 각 바이트가 하나의 명령으로 모델링 됨 (0-255 사이의 값에 대한 모델 생성)

Format String Exploit Generation



아래의 조건을 만족하면 exploitable:

$$
\begin{aligned}
&[\text{FMTPTR} < \text{DIS} + (\text{LEN} - 1) - 1] \\
\wedge\ &[\text{ARGPTR} > \text{DIS}] \\
\wedge\ &[\text{ARGPTR} < \text{DIS} + (\text{LEN} - 1) - 4] \\
\wedge\ &[*\text{FMTPTR} = \text{`\%'}] \\
\wedge\ &[*(\text{FMTPTR} + 1) = \text{`s'}] \\
\wedge\ &[*\text{ARGPTR} = a_1] \\
\wedge\ &[*(\text{ARGPTR} + 1) = a_2] \\
\wedge\ &[*(\text{ARGPTR} + 2) = a_3] \\
\wedge\ &[*(\text{ARGPTR} + 3) = a_4] \\
\wedge\ &[\text{MODE} = \text{printing}]
\end{aligned}
$$

한계점

- 최초의 자동화 시도였으나 주어진 API 모델 상에서만 동작
- 실제 Shell을 실행하는 exploit과는 거리가 있음
- 생성된 exploit이 실제로 동작하지 않을 수 있음:
  실제 실행 경로 상에서 허용되지 않는 입력 값이 존재할 수 있음

2008, Brumley et al.

Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications, ***Oakland 2008***

패치에서부터 자동으로 exploit을 생성하는 최초의 시도

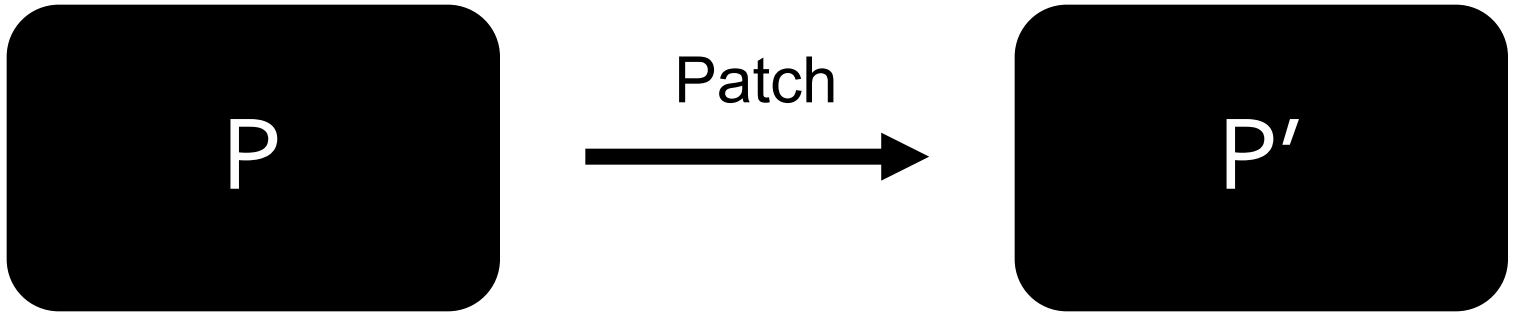
왜 패치인가?

- 윈도우 업데이트가 80%의 클라이언트에 설치되기까지는 최소 24시간이 걸림
- 하지만 일반적으로 웜이 퍼지는 데에는 1시간 이내가 소요
- 패치로부터 자동으로 몇 분 내에 익스플로잇 생성이 가능하다면?

Automatic Patch-based Exploit Generation (APEG)

- 주요 대상: 입력 값 검증 (validation) 버그

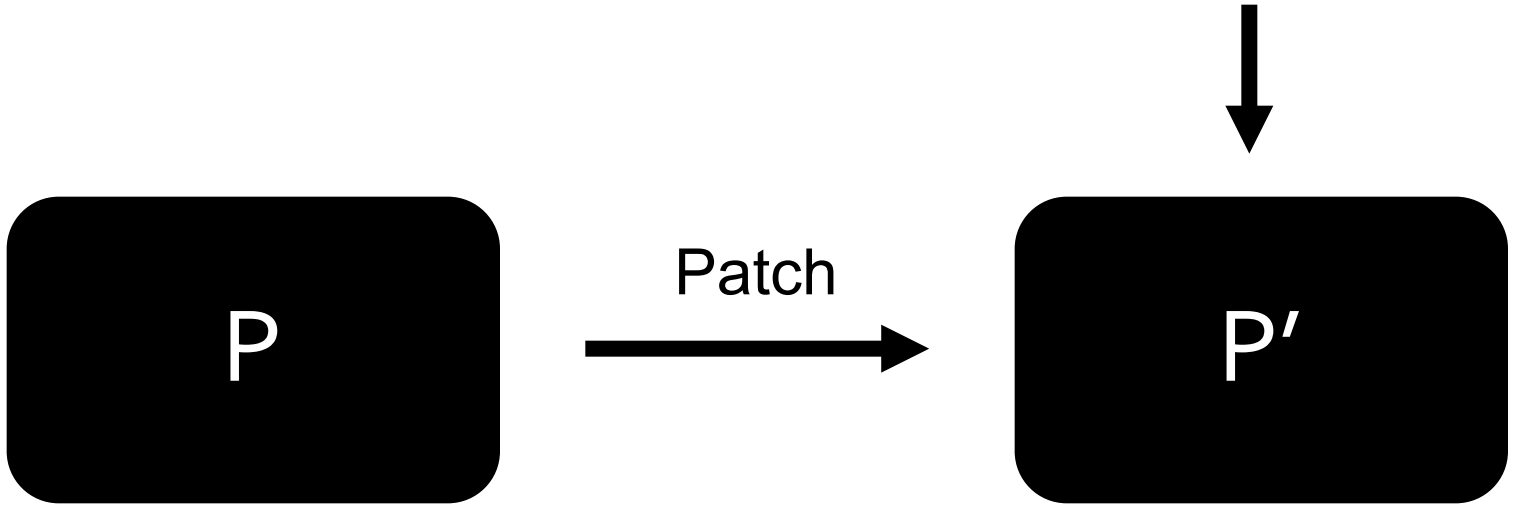
- 관찰: 입력 값 검증 버그의 경우, sanitization 체크를 넣어서 패치하는 경우가 대부분임



Automatic Patch-based Exploit Generation (APEG)

Weakest precondition

패치된 프로그램에서 sanitization 체크를 통과하지 못하는 입력값을 찾으면, 원래 프로그램에서의 익스플로잇일 가능성이 높음



한계점

- 입력 값 검증과 관련된 버그만 처리 가능
- 실제 Shell을 실행하는 exploit과는 거리가 있음

2009, Heelan et al.

Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities, MS Thesis

최초의 Control Flow Hijack 생성

- 주어진 프로그램 크래쉬에 대해서 정해진 알고리즘에 의해 exploit을 자동생성

- 문제점: 특정한 조건의 (eip가 handle 되거나 임의의 쓰기가 가능한) 크래쉬에 대해서만 exploit 생성 가능

2011, Avgerinos et al.

AEG: Automatic Exploit Generation, ***NDSS 2011***

최초로 버그를 찾고 익스플로잇 생성까지 자동화 (소스기반)

AEG 의 현재

AEG 관련 연구들

AEG: Automatic Exploit Generation, ***NDSS 2011***

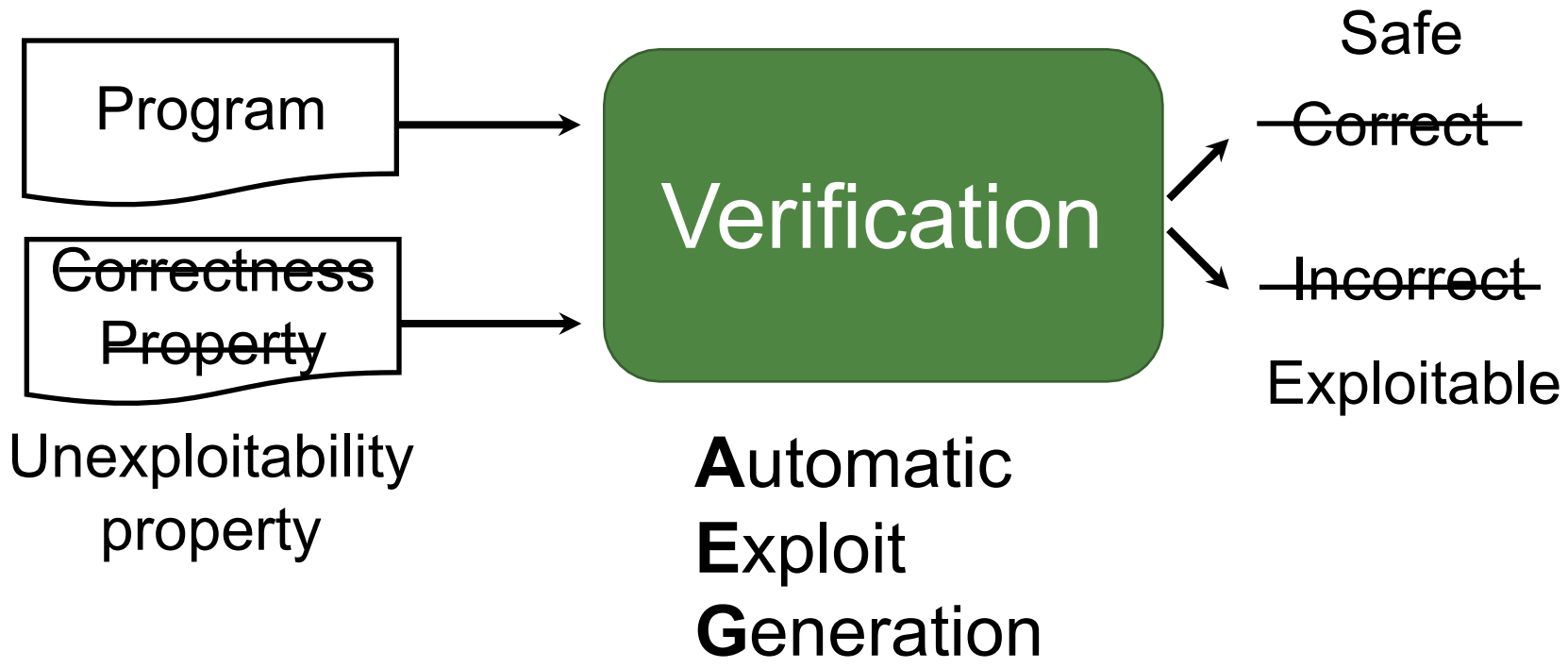
Unleashing Mayhem on Binary Code, ***Oakland 2012***

Autoamtic Exploit Generation, ***CACM 2014***

Automatic Generation of Data-Oriented Exploits, ***USENIX 2015***

Data-oriented Programming: On the Expressiveness of Non-control Data Attacks, ***Oakland 2016***

Verification with a Twist

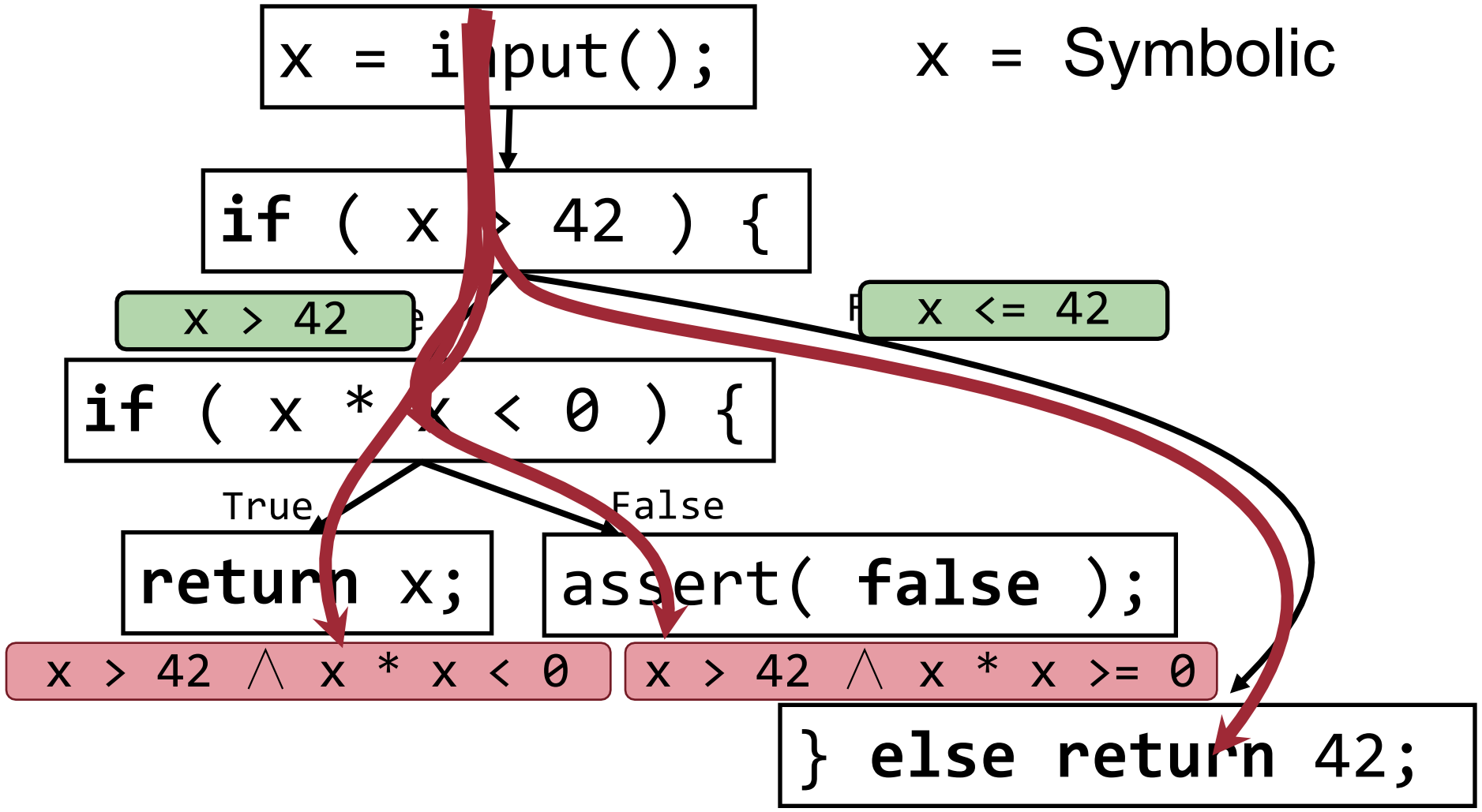


기호실행 Symbolic Execution*

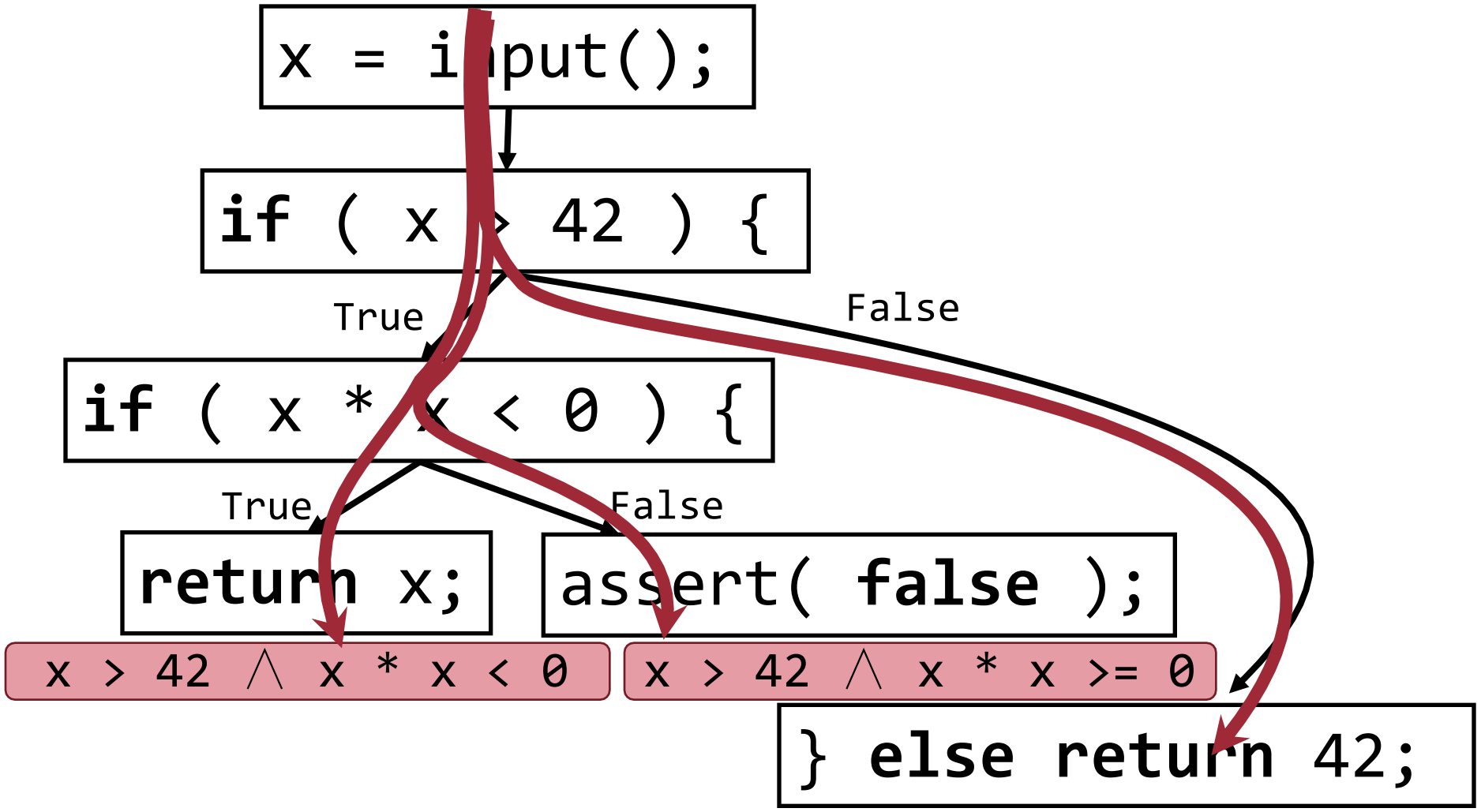
- Systematically explore execution paths
- For each execution path, construct a ***path formula*** that describes the input constraints to follow the path

* Robert S. Boyer, et al., ACM SIGPLAN Notices 1975
William E. Howden, *IEEE Transactions on Computers*, 1975
James C. King, CACM 1976

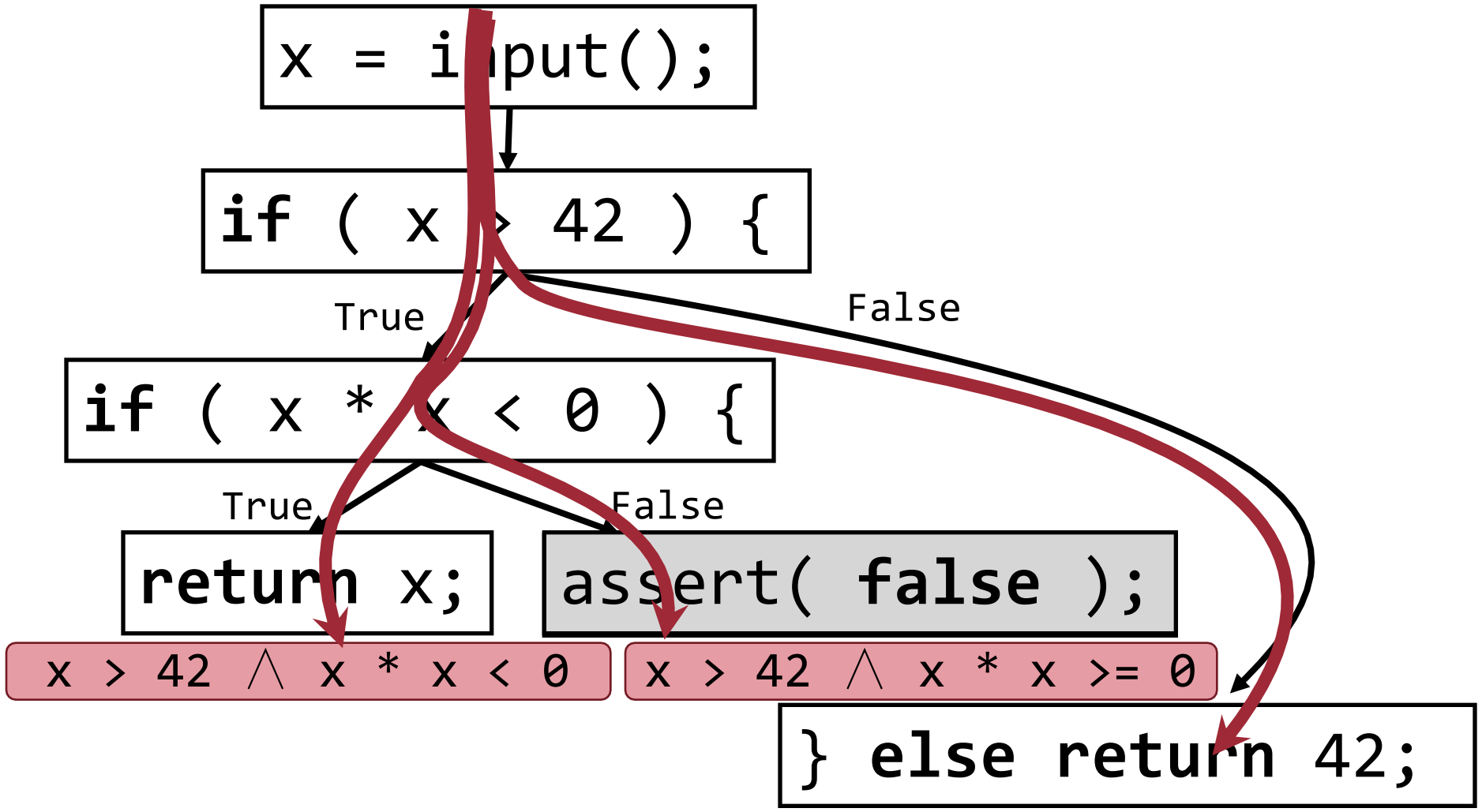
기호실행 Symbolic Execution



경로식 Path Formulas

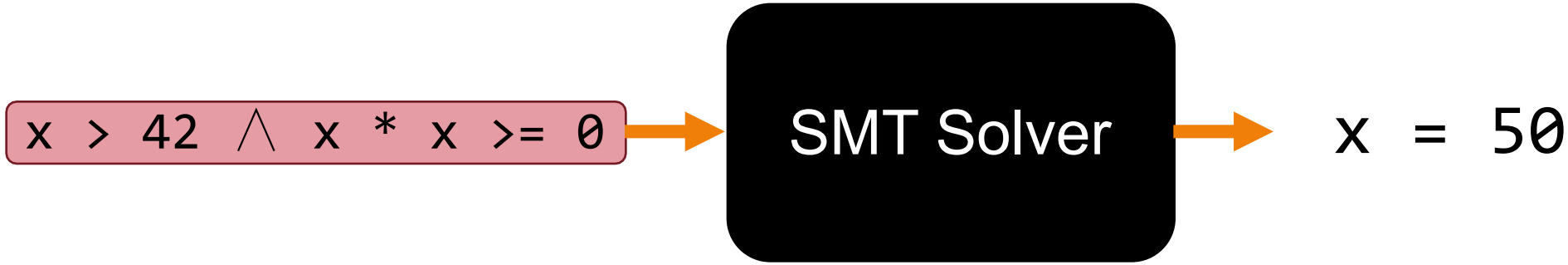


경로를 탐색하는 입력값 생성

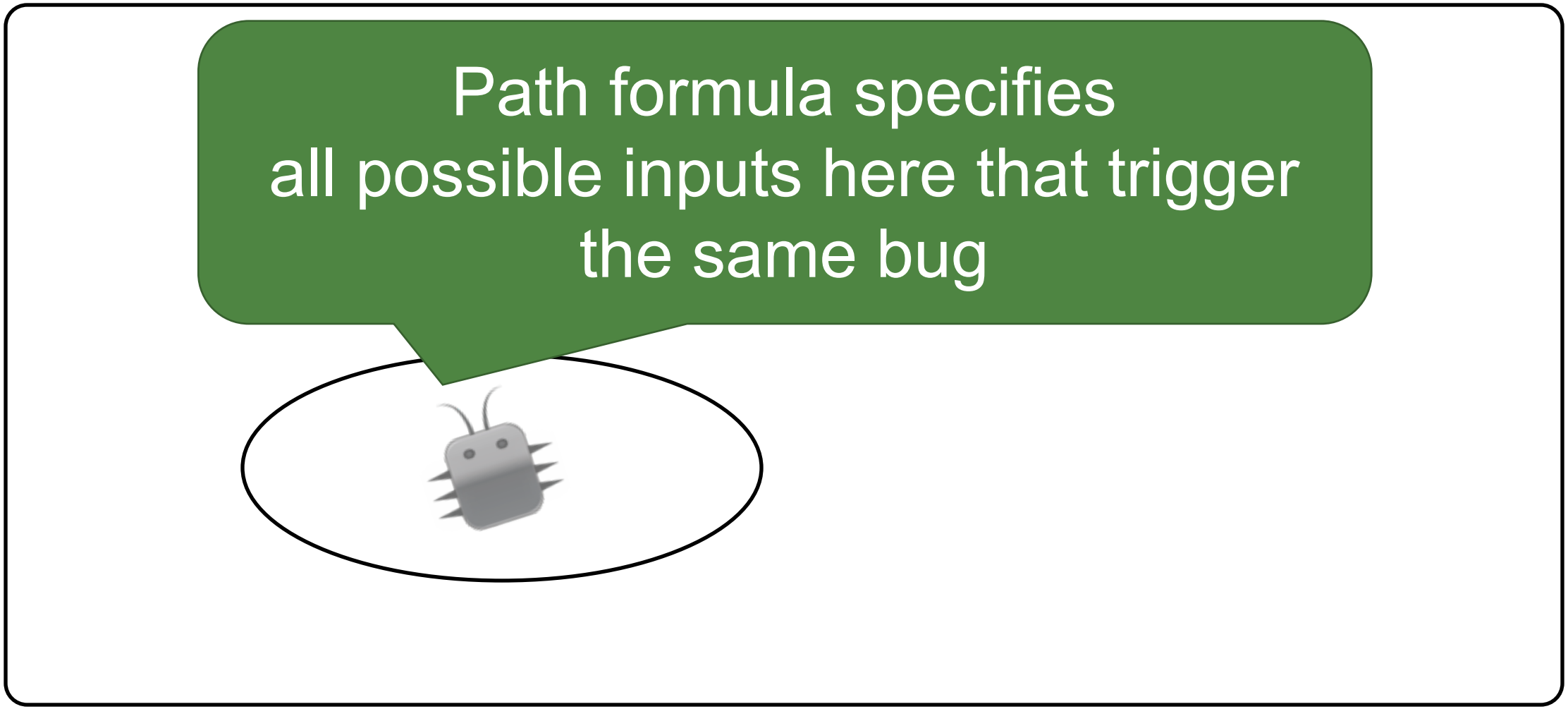


어떻게 경로식을 푸는가?

SMT (Satisfiability Modulo Theory) Solver를 활용

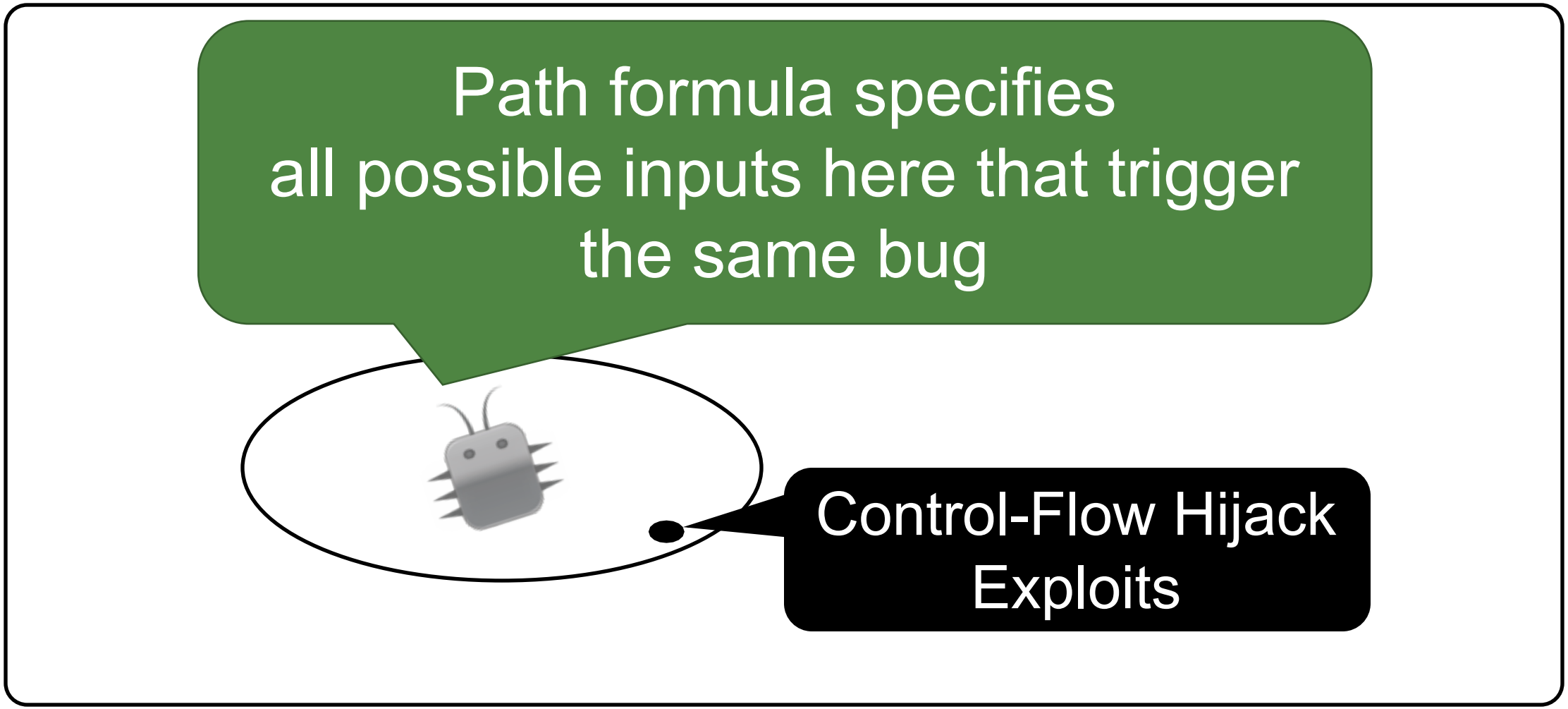


All Inputs



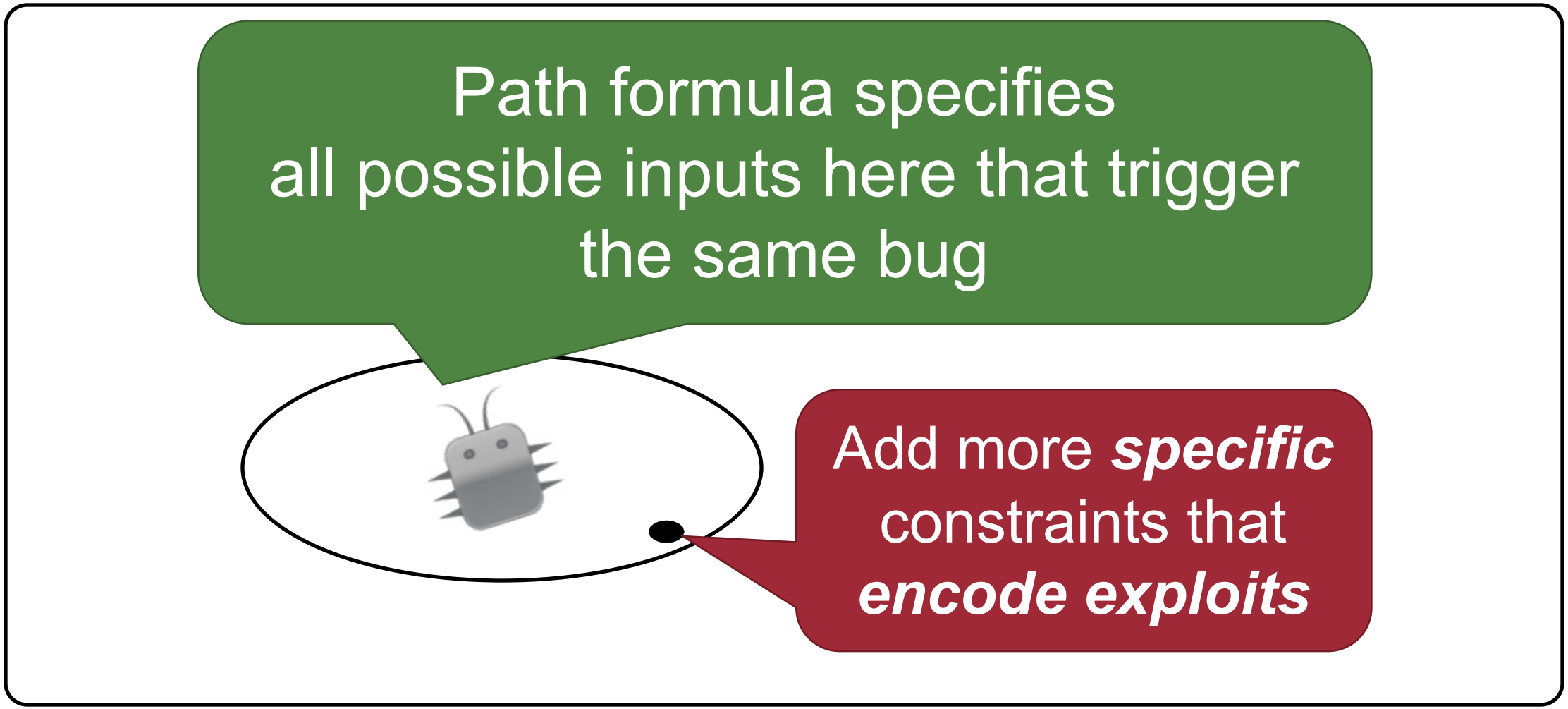
Program's Input Space

All Inputs



Exploit is just another input that triggers bugs

All Inputs



Exploit is just another input that triggers bugs

Exploit Constraints

Path formula derived from a symbolic execution

$\wedge$ (logical and)

Exploit Constraints

AEG 관련 연구들

AEG: Automatic Exploit Generation, ***NDSS 2011***

Unleashing Mayhem on Binary Code, ***Oakland 2012***

Autoamtic Exploit Generation, ***CACM 2014***

소스 기반

바이너리 기반

바이너리 분석의 문제점

Binary analysis is difficult because binary code does not have any ***program abstraction***

```
4C 8B 47 08          mov         r8,qword ptr [rdi+8]
BA 02 00 00 00       mov         edx,2
48 8B 4F 20          mov         rcx,qword ptr [rdi+20h]
45 0F B7 08          movzx       r9d,word ptr [r8]
E8 54 16 00 00       call        00000001400026BC
48 8B 74 24 38       mov         rsi,qword ptr [rsp+38h]
8B C3                mov         eax,ebx
48 8B 5C 24 30       mov         rbx,qword ptr [rsp+30h]
48 83 C4 20          add         rsp,20h
5F                   pop         rdi
C3                   ret
48 8B C4             mov         rax,rsp
48 89 58 08          mov         qword ptr [rax+8],rbx
48 89 68 10          mov         qword ptr [rax+10h],rbp
48 89 70 18          mov         qword ptr [rax+18h],rsi
48 89 78 20          mov         qword ptr [rax+20h],rdi
41 54                push        r12
41 56                push        r14
41 57                push        r15
48 83 EC 40          sub         rsp,40h
48 8B 9C 24 90 00    mov         rbx,qword ptr [rsp+0000000000000090h]
```

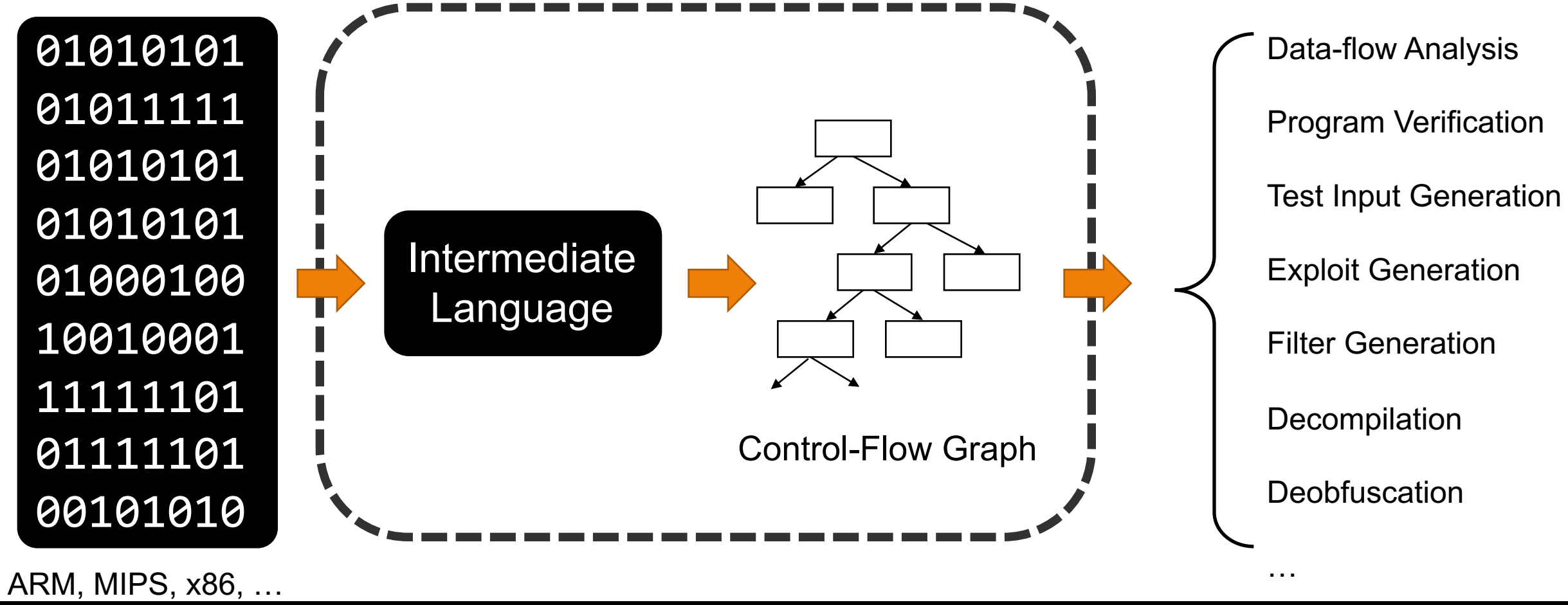
No Types,
No Variable Names,
No Functions,
etc.

AEG의 역설 …

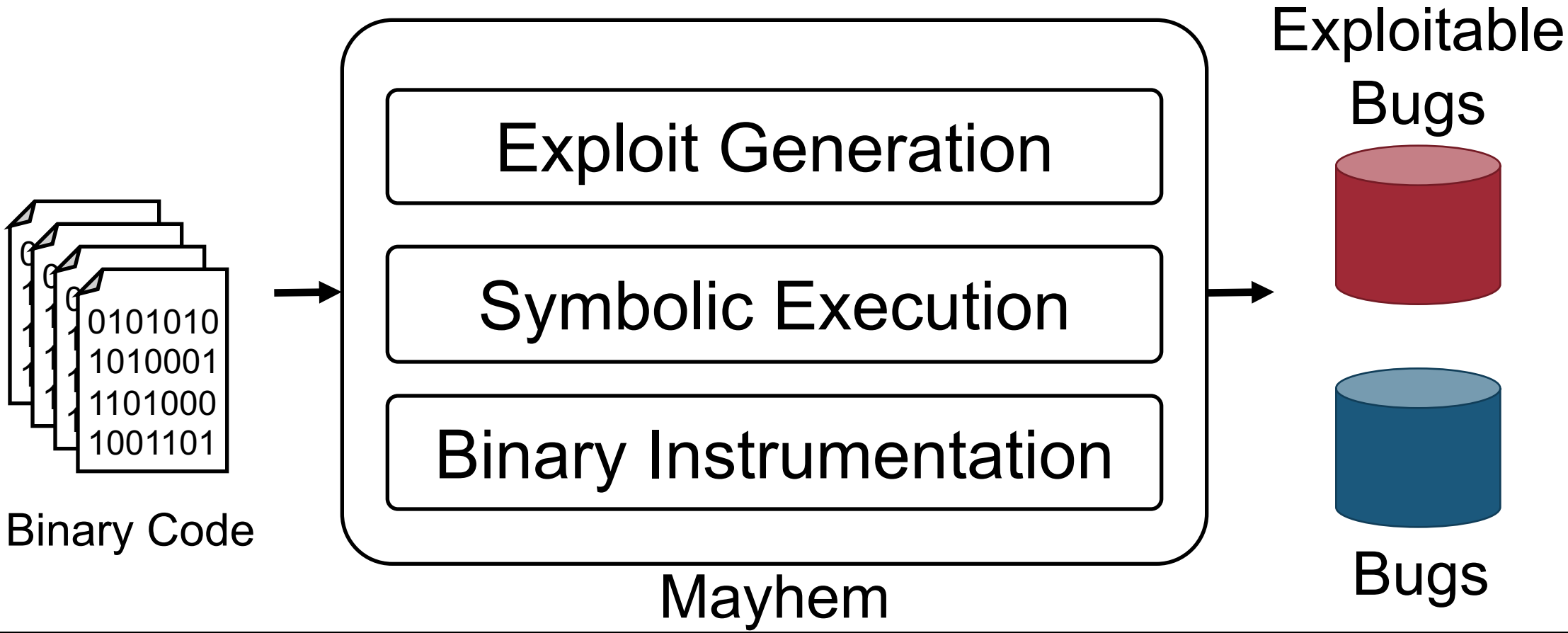
- 바이너리 분석은 어렵다

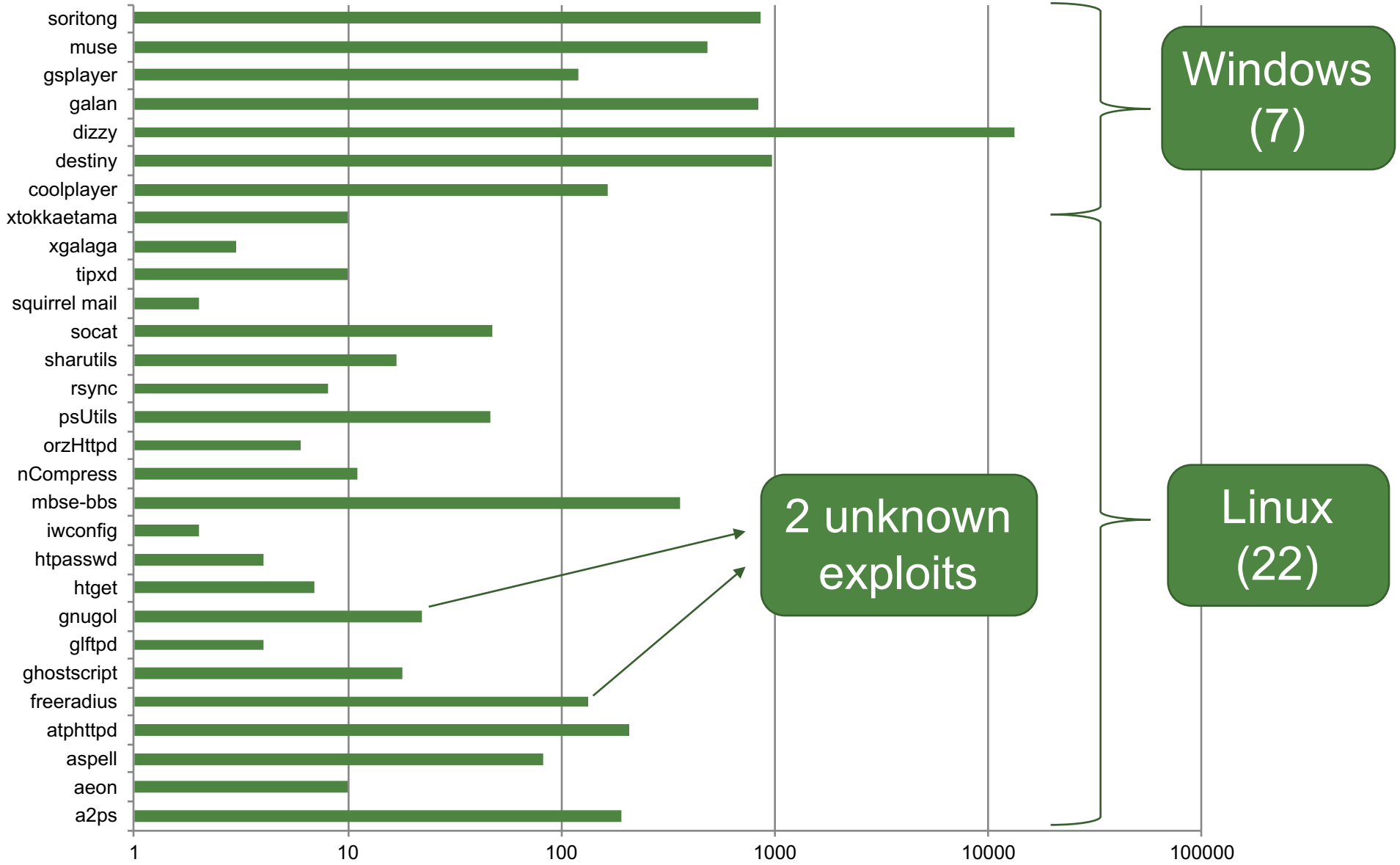
- 하지만 AEG에서는 소스코드 보다 바이너리 분석이 더 쉽다
  - 소스코드에서는 상세한 메모리 구조를 알 수 없음
  - 소스코드가 같더라도 컴파일러마다 전혀 다른 바이너리 코드를 생성할 수 있음
  - Exploit을 만드는데에는 메모리 구조에 대한 이해가 필수적임

자동화된 바이너리 분석 (역공학)



Mayhem: 최초의 바이너리 기반 AEG

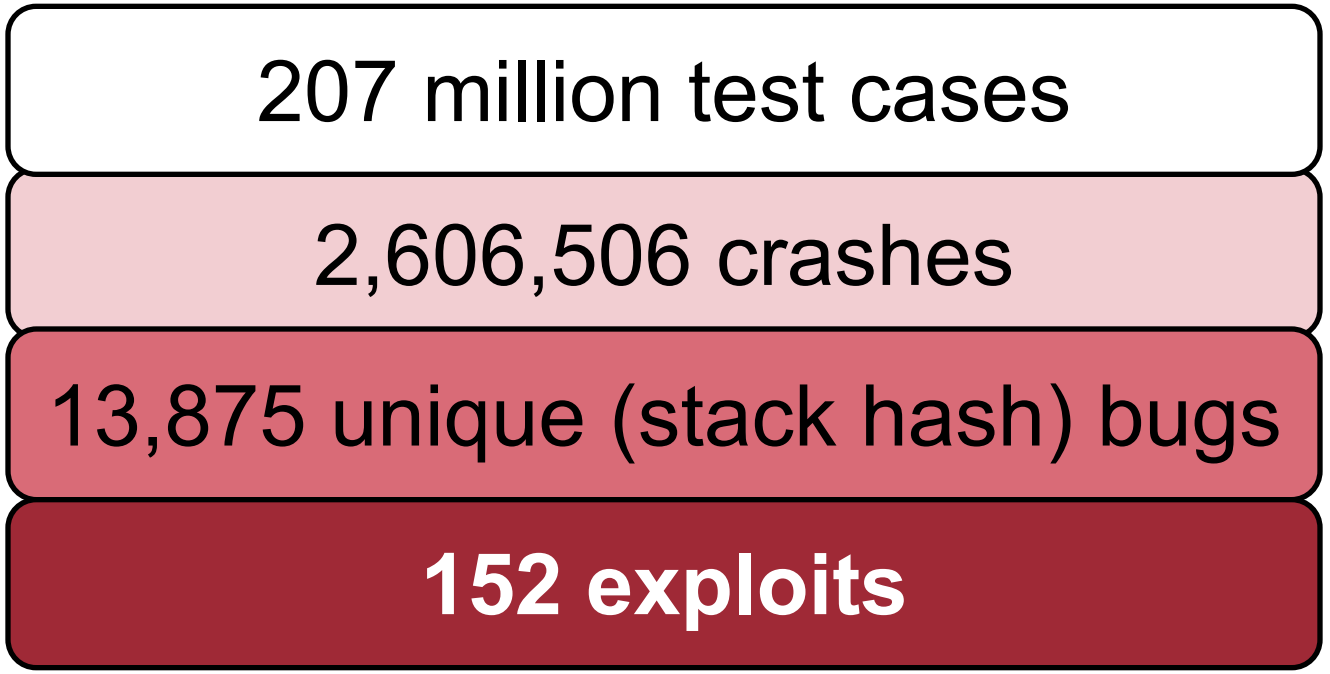




Exploit Generation Time (sec.)

지금까지의 결과

37,391 distinct binaries from Debian Linux
7.7 years CPU-time



Mayhem의 한계점

- We do not claim to find all exploitable bugs
- Given an exploitable bug, we do not guarantee we will always find an exploit

But Every Report is Actionable

- Lots of room for improving symbolic execution, chaining multiple vulnerabilities, etc.
- We do not consider defenses (such as DEP and ASLR)

또 다른 관련 연구들 (Data Exploits)

AEG: Automatic Exploit Generation, ***NDSS 2011***

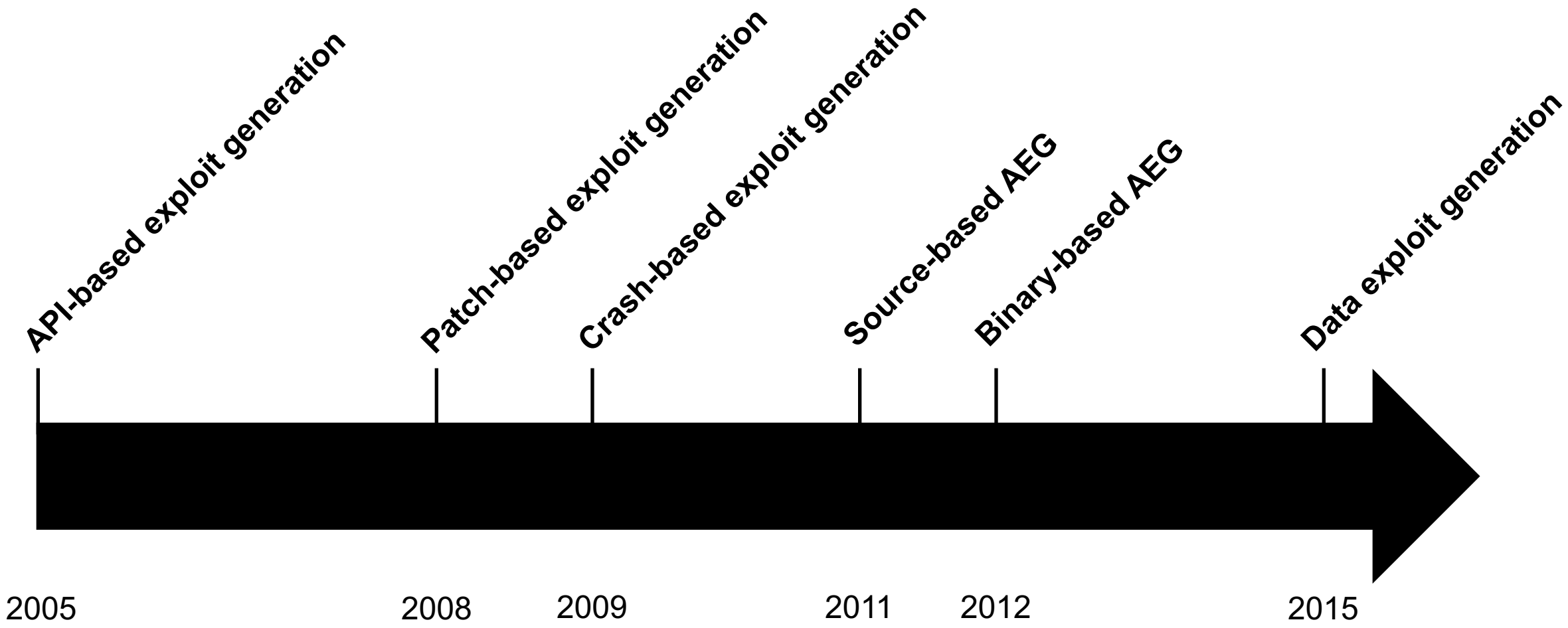
Unleashing Mayhem on Binary Code, ***Oakland 2012***

Autoamtic Exploit Generation, ***CACM 2014***

Automatic Generation of Data-Oriented Exploits, ***USENIX 2015***

Data-oriented Programming: On the Expressiveness of Non-control Data Attacks, ***Oakland 2016***

Timeline

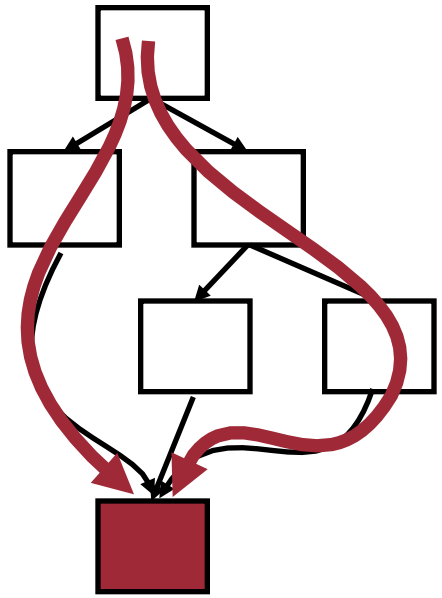

API-based exploit generation
Patch-based exploit generation
Crash-based exploit generation
Source-based AEG
Binary-based AEG
Data exploit generation
2005
2008
2009
2011
2012
2015

AEG 의 미래

AEG에서의 핵심 Challenge는?

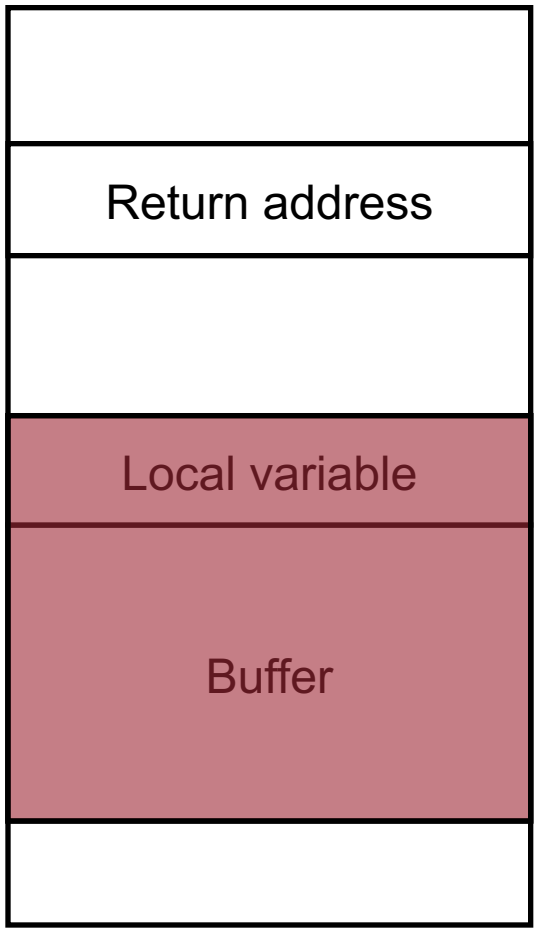
✓ 취약점을 찾는 것 vs. Exploit을 생성하는 것

AEG에서의 핵심은 ~~취약점을~~ 찾는 것이다



찾아낸 취약점을 유발하되,
특정 조건을 만족하면서 그 취약점에
도달할 수 있는 실행 경로를

특정 조건을 만족하는 실행경로?



버퍼오버플로우 발생

버퍼오버플로우 발생

동일한 취약점
서로 다른 경로

EIP를 컨트롤하기
= EIP를 컨트롤할 수 있는 실행경로 찾기

취약점 탐지기술의 발전

- White-box (기호실행) 과 퍼징의 접목
  - Driller, ***NDSS 2016***

- 동적분석과 정적분석과의 접목

- 단순 취약점이 아닌 취약점을 도달하는 여러 경로에 대한 탐색 기술 발전

바이너리 분석의 발전

- 세계적 관심분야
  - Singapore NUS: $6.1 Million Since 2014
  - Microsoft is commercializing their binary analysis tool

- 좀 더 정확히 program abstraction을 복원하는 기술 필요

Exploit Hardening과의 접목

방어 체계를 무력화하는 방법

Q: Exploit Hardening Made Easy, ***USENIX Security 2011***

다양한 공격기술 개발

- Use-after-free나 type confusion 버그 등에 대한 자동화된 공격 기술 개발
- Memory leak 등을 활용한 취약점 공격 기술 개발
- 1개 이상의 취약점을 접목한 공격 기술 개발

자동화된 패치 기술과의 접목

- 취약점의 탐지, 검증에 이어 자동화된 수정 (패치) 까지 연결되는 기술 개발

- 핵심은 버그의 근본 원인을 자동으로 파악하는 것
  - Root cause analysis

결론

- AEG는 2005년 부터 시작된 신생 분야이다
- AEG를 위해서 수많은 프로그램 분석 기법이 사용되어 왔다.
  - Bounded model checking
  - Symbolic execution
  - Weakest precondition
- AEG에서 바이너리 분석은 필수적이다.
- 단순히 취약점을 찾는 것보다 특정 조건을 유발하는 실행경로를 찾는 것이 더 중요하다
- 앞으로 더 다양한 방향으로의 발전이 예상된다.

Question?