AI-driven Security for Automotive

2025.08

자동차융합대학

GENERAL MOTORS
GM TECHNICAL CENTER KOREA

KMU 국민대학교
KOOKMIN UNIVERSITY

CONTENTS

01

Overview

■ Why we have to use the Automated Vulnerability Detection System?

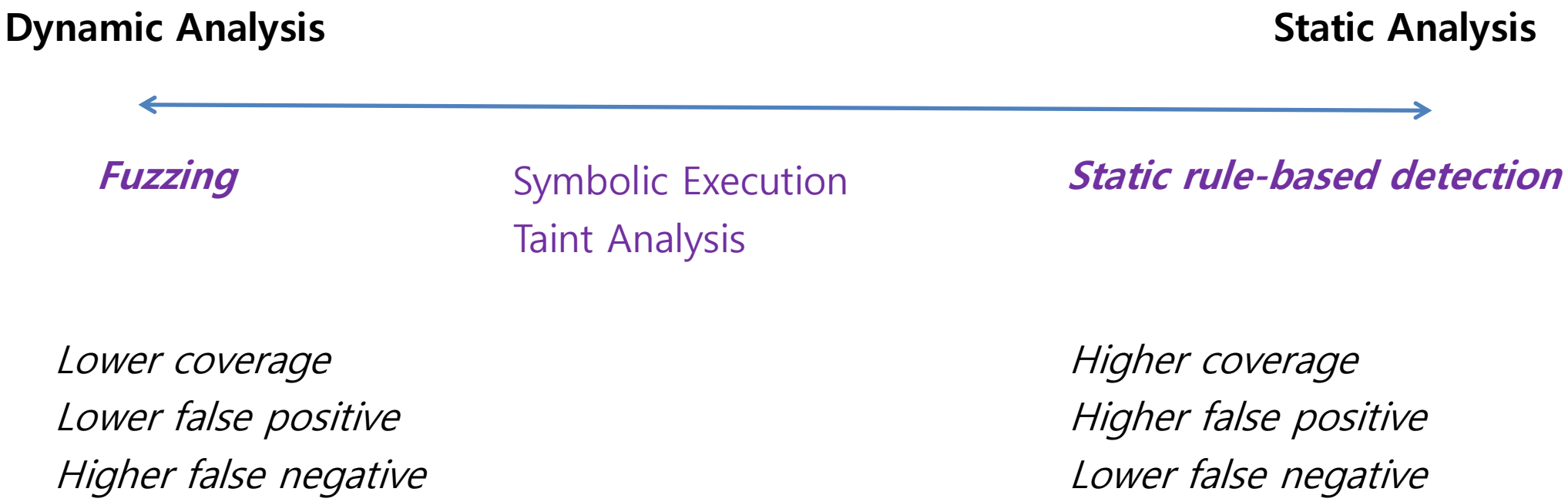
Ref: https://www.getadvanced.net/technology-blog/article/vulnerability-scanning-vs-penetration-testing-whats-the-difference

■ Why we have to use the Automated Vulnerability Detection System?

	VULNERABILITY ASSESSMENT	PENETRATION TEST
Organizational Security Program Maturity Level	Low to Medium. Usually requested by organizations that already know they have issues, and need help getting started.	High. The organization believes their defenses to be strong, and wants to test that assertion.
Goal	Attain a prioritized list of vulnerabilities in the environment so that remediation can occur.	Determine whether a mature security posture can withstand an intrusion attempt from an advanced attacker with a specific goal.
Focus	Breadth over depth.	Depth over breadth.

Ref: https://www.getadvanced.net/technology-blog/article/vulnerability-scanning-vs-penetration-testing-whats-the-difference

Techniques for vulnerability detection



Related Work

■ Dynamic Analysis based Approaches

- Fuzzing
- Symbolic Execution
- Concolic Execution (Concrete + Symbolic)
- Taint Analysis (Include DBI)
- *AI driven Fuzzing*

■ Static Analysis based Approaches

- Code Similarity based Approaches → VUDDY (IoTCube), VulPecker
- Pattern based Approaches
  - Rule-based → Clang, Fortify, Coverity, etc.
  - Traditional Machine Learning-based
  - *Deep Learning-based → VulDeePecker, SySeVR, AutoVAS…*

02

Static Approach

Code Similarity based Vulnerability Detection

■ VUDDY (IEEE S&P 2017)

- 함수 해시 비교로 복붙된 취약 코드를 1,000배 빠르고 정확하게 탐색

Level 0: No abstraction.

```
1 void avg (float arr[], int len) {
2   static float sum = 0;
3   unsigned int i;
4   for (i = 0; i < len; i++);
5     sum += arr[i];
6   printf("%f %d",sum/len,validate(sum));
7 }
```

Level 1: Formal parameter abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   static float sum = 0;
3   unsigned int i;
4   for (i = 0; i < FPARAM; i++)
5     sum += FPARAM[i];
6   printf("%f %d",sum/FPARAM,validate(sum);
7 }
```

Level 2: Local variable name abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   static float LVAR = 0;
3   unsigned int LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR++)
5     LVAR += FPARAM[LVAR];
6   printf("%f %d",LVAR/FPARAM,validate(LVAR));
7 }
```

Level 3: Data type abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   DTYPE LVAR = 0;
3   unsigned DTYPE LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR++)
5     LVAR += FPARAM[LVAR];
6   printf("%f %d",LVAR/FPARAM,validate(LVAR));
7 }
```

Level 4: Function call abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   DTYPE LVAR = 0;
3   unsigned DTYPE LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR)
5     LVAR += FPARAM[LVAR];
6   FUNCCALL("%f %d",LVAR/FPARAM,FUNCCALL(LVAR));
7 }
```

Fig. 2: Level-by-level application of abstraction schemes on a sample function.

Original:	`int sum (int a, int b) { return a + b; }`
Preprocessed:	`returnfparam+fparam;`
Length:	20
Hash value:	c94d99100e084297ddbf383830f655d1
Fingerprint:	{20, c94d99100e084297ddbf383830f655d1}
Original:	`void increment () { int num = 80; num++; /* no return val */ }`
Preprocessed:	`dtypelvar=80;lvar++;`
Length:	20
Hash value:	d6e77882a5c55c67f45f5fd84e1d616b
Fingerprint:	{20, d6e77882a5c55c67f45f5fd84e1d616b}
Original:	`void printer (char* src) { printf("%s", src); }`
Preprocessed:	`funccall("%s",fparam);`
Length:	23
Hash value:	9a45e4a15c928699afe867e97fe839d0
Fingerprint:	{23, 9a45e4a15c928699afe867e97fe839d0}

Fig. 3: Example functions and corresponding fingerprints.

Challenge	기존 코드 스니펫 비교 방식은 오탐 과다
Solution	**함수 단위로 코드 추상화 + 정규화 후 해시화** → 함수 하나 = 해시 하나 → 기존 취약 함수의 해시와 비교하여 탐지
Impact	**1,000배 빠르고, 높은 정확도!** 많은 프로젝트에서 **복붙된 취약 함수 자동 탐지**

Code Similarity based Vulnerability Detection

■ CENTRIS (ICSE 2021)

- 수정된 오픈소스 구성요소를 90% 이상의 정확도로 식별

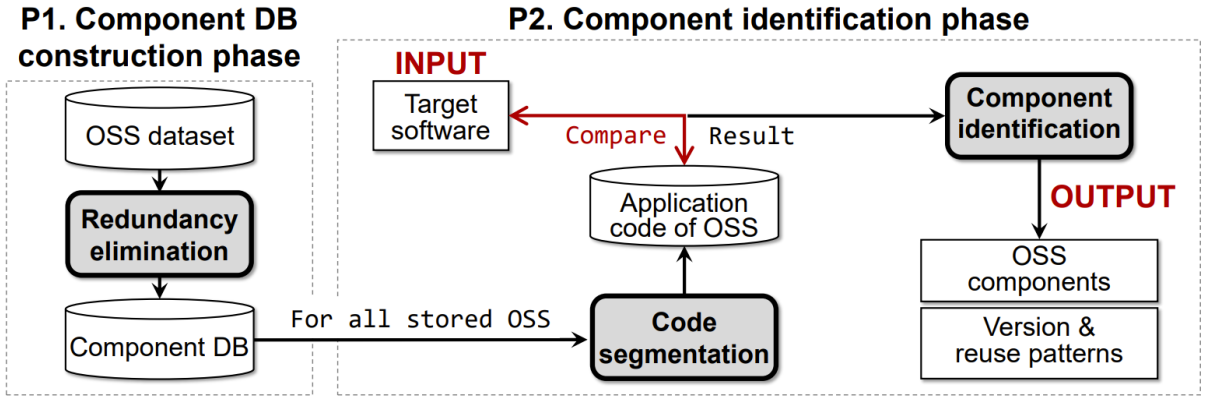


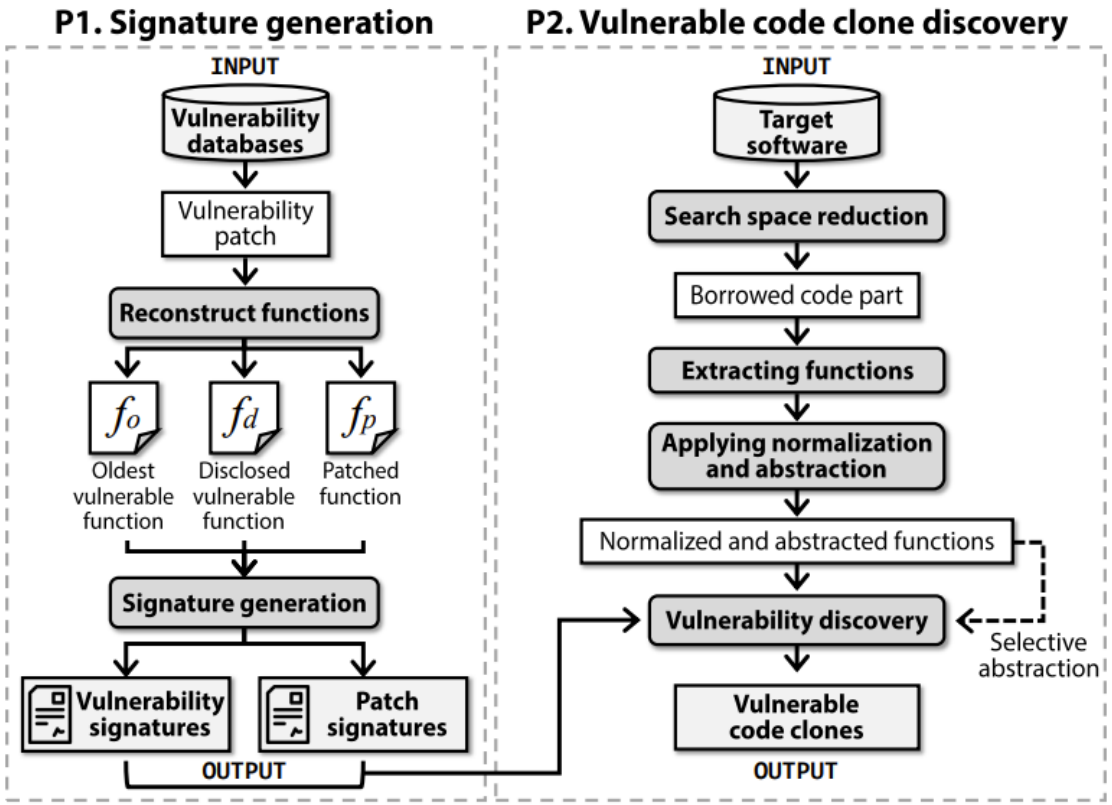
Fig. 2: High-level overview of the workflow of CENTRIS.

Challenge	기존 방식은 함수 전체를 비교하기 때문에 정확도가 낮음 (~50%)
Solution	전체 함수에서 시그니처를 생성하는 대신, ✔ **중복 제거 (Redundancy Elimination)** ✔ **코드 세분화 (Code Segmentation)** 통해 **불필요 정보 줄이고, 핵심 패턴만 추출**
Impact	기존 대비 **9배 향상된 정확도 (90% 이상)** 불완전하거나 **수정된 OSS까지 식별** 가능!

Code Similarity based Vulnerability Detection

■ MOVERY (USENIX Security 2022)

- 수정된 OSS 코드에서도 취약점을 정확하게 탐지



Challenge	오픈소스를 **수정하거나 일부 변경**했을 경우, 기존 취약점 스캐너(CVE 기반)는 **탐지 실패 가능성** 높음
Solution	✔ 취약 함수 **수정 전후 코드 라인 분석**, 시그니처 생성 ✔ 수정된 코드 기반에서도 동일한 취약점 여부 탐지
Impact	기존 CVE 기반 스캐닝보다 **높은 정확도** **자체 커스터마이징한 OSS 환경도 안정적으로 탐지 가능**

Figure 2: High-level overview of the workflow of MOVERY.

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- OSS 재사용으로 인해 전파된 취약코드를 탐지하는 기술
- OSS의 코드는 수정과 함께 전파되는 경우가 많음
  - 취약 코드의 모양(구문) 역시, 알려진 취약 코드와 매우 다른 경우가 존재
- 예: Lua 취약점 케이스 (CVE-2014-5461; DoS 취약점)
  - Lua 5.1 부터 Lua 5.2.3 까지 존재했던 취약점

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- OSS 재사용으로 인해 전파된 취약코드를 탐지하는 기술
- OSS의 코드는 수정과 함께 전파되는 경우가 많음
  - 취약 코드의 모양(구문) 역시, 알려진 취약 코드와 매우 다른 경우가 존재
- 예: Lua 취약점 케이스 (CVE-2014-5461; DoS 취약점)
  - Lua 5.1 부터 Lua 5.2.3 까지 존재했던 취약점

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- 예: Redis에서 발견된 Lua 취약점
  - 같은 취약점을 포함하고 있지만 취약 함수들의 구문이 아주 다름

```
int luaD_precall (lua_State *L, StkId func, int nresults) {
  lua_CFunction f;
  CallInfo *ci;
  int n;  /* number of arguments (Lua) or returns (C) */
  ptrdiff_t funcr = savestack(L, func);
  switch (ttype(func)) {
    …
    case LUA_TLCL: {  /* Lua function: prepare its call */
      StkId base;
      Proto *p = clLvalue(func)->p;
    - luaD_checkstack(L, p->maxstacksize);
    - func = restorestack(L, funcr);
      n = cast_int(L->top - func) - 1;
    + luaD_checkstack(L, p->maxstacksize);
```

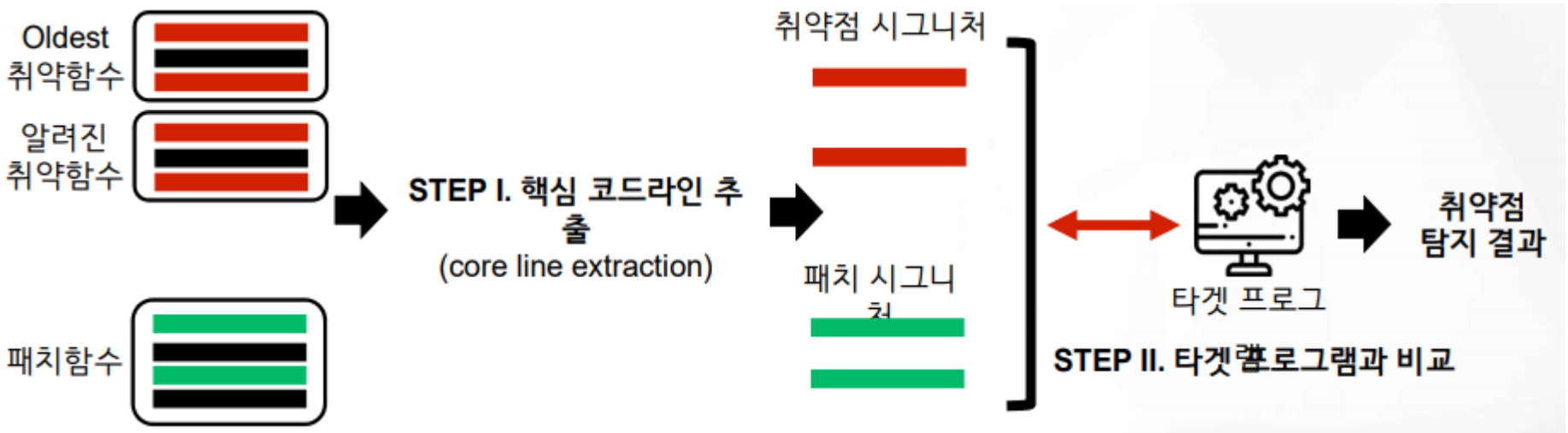
CVE-2014-5461의 알려진 취약함수

```
int luaD_precall (lua_State *L, StkId func, int nresults) {
  LClosure *cl;
  ptrdiff_t funcr;
  if (!ttisfunction(func)) /* `func' is not a function? */
    func = tryfuncTM(L, func);  /* check the `function' tag method */
  funcr = savestack(L, func);
  cl = &clvalue(func)->l;
  L->ci->savedpc = L->savedpc;
  if (!cl->isC) {  /* Lua function? prepare its call */
    CallInfo *ci;
    StkId st, base;
    Proto *p = cl->p;
  - luaD_checkstack(L, p->maxstacksize);
  + luaD_checkstack(L, p->maxstacksize + p->numparams);
```

Redis에서 발견된 동일 취약점의 취약 함수
(Lua 5.1 재사용)

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- MOVERY의 핵심 아이디어
  - 가장 오래된 취약함수/알려진 취약함수를 모두 고려
  - 취약/패치 함수 내의 핵심 코드라인만을 고려하여 전파된 취약점 탐지



■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- 핵심 코드라인
  - 필수 코드라인 (Essential Code Lines)
  - 의존 코드라인 (Dependent Code Lines)
  - 제어 흐름 코드라인 (Control-flow Code Lines)

Oldest 취약함수

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = (parity) ? hstartcol : (numrows - hstartcol);
11
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

알려진 취약함수 및 패치

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9   - hstartcol = (numrows + 1 - parity) > 1;
10  - // ORIGINAL (WRONG): m = (parity) ?
                            hstartcol : (numrows - hstartcol);
11  - m = numrows - hstartcol;
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```


■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- 필수 코드라인 (Essential Code Lines)
  - 패치에서 삭제된 코드 라인 중 가장 오래된 취약함수와 알려진 취약함수에 모두 존재하는 코드라인
  - 실제 취약점 발현과 밀접한 관련이 있는 코드 라인들

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = (parity) ? hstartcol : (numrows - hstartcol);

11
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Oldest Vulnerable Function

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9   - hstartcol = (numrows + 1 - parity) > 1;
10  - // ORIGINAL (WRONG): m = (parity) ?
                            hstartcol : (numrows - hstartcol);
11  - m = numrows - hstartcol;
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Disclosed Vulnerable Function

Code Similarity based Vulnerability Detection

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- 의존 코드라인 (Dependent Code Lines)
  - 필수 코드라인과 control/data 의존성이 존재하는 코드라인
  - 취약점이 전파된 후에도 여전히 발현될 가능성이 있는지를 확인하기 위함

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = (parity) ? hstartcol : (numrows - hstartcol);

11
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Oldest Vulnerable Function

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    // ORIGINAL (WRONG): m = (parity) ?
                              hstartcol : (numrows - hstartcol);
11    m = numrows - hstartcol;
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Disclosed Vulnerable Function

Code Similarity based Vulnerability Detection

■ **MOVERY: 오픈소스 소프트웨어 취약코드 탐지**

- 조건 제어 코드라인 (Control-flow Code Lines)
  - 필수 코드라인 까지 거쳐가는 모든 조건제어문들
  - 같은 조건으로 취약점이 트리거 될 수 있는지를 확인하기 위함

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = (parity) ? hstartcol : (numrows - hstartcol);

11
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Oldest Vulnerable Function

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5        abort();
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    // ORIGINAL (WRONG): m = (parity) ?
                              hstartcol : (numrows - hstartcol);
11    m = numrows - hstartcol;
12    n = m;
13    dstptr = buf;
14    srcptr = &a[(1 - parity) * stride]
15    ...
```

Disclosed Vulnerable Function

■ MOVERY: 오픈소스 소프트웨어 취약코드 탐지

- 핵심 코드라인들만 모아서 Vulnerable Signature 생성
  - 같은 방식으로 Patch Signature 생성
  - 다만, 패치에서 추가된 코드라인을 고려

- 타겟 프로그램의 함수(f)가 아래 세가지 조건을 모두 만족하면 전파된 취약 함수로 판단
  - Vulnerable Signature의 모든 코드라인을 포함
  - Patch Signature의 어떠한 코드라인도 포함하지 않음
  - Oldest 취약 함수 혹은 알려진 취약 함수의 구문 유사도가 50% 이상

Code Similarity based Vulnerability Detection

■ CNEPS (ICSE 2024)

- 복잡한 OSS 의존성을 정밀하게 mapping → 보다 효율적인 위협 대응 가능

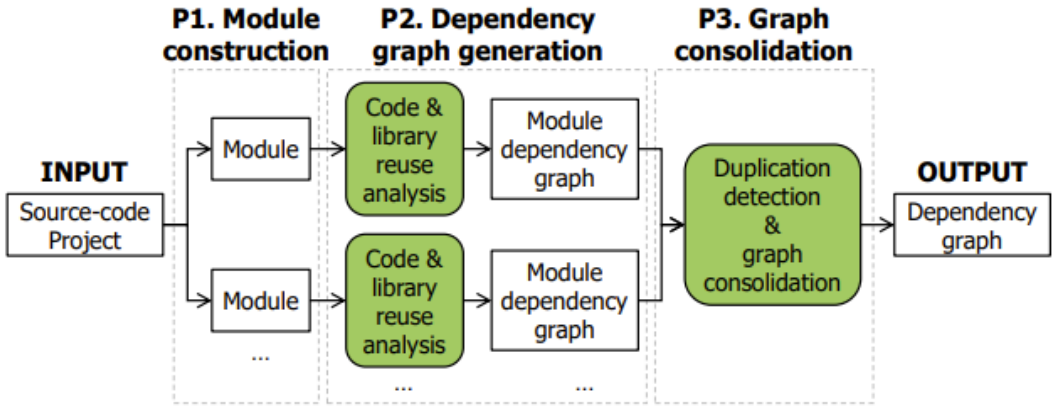


Figure 3: High-level workflow of CNEPS.

Challenge	OSS 의존성은 **계층적이고 복잡**해서 **정확한 리스크 분석이 어려움**
Solution	✔ 구성요소 간 **의존성 관계 맵핑** **(Dependency Mapping)** ✔ 이를 통해 위험도 평가(Risk Assessment), **업데이트 영향도 관리**
Impact	SBOM의 의존성 탐지 정확도 **75% 향상**

Deep learning-based Vulnerability Detection

■ Motivation

- Fuzzing, Symbolic Execution 등의 Automated Vulnerability Detection System은 매우 중요한 Research 영역이지만, **기존 기법들은 높은 수준의 Human Expertise가 요구되며, Coverage 관점에서도 Vulnerability를 Miss하는 경우가 많음. (High False Negative)**

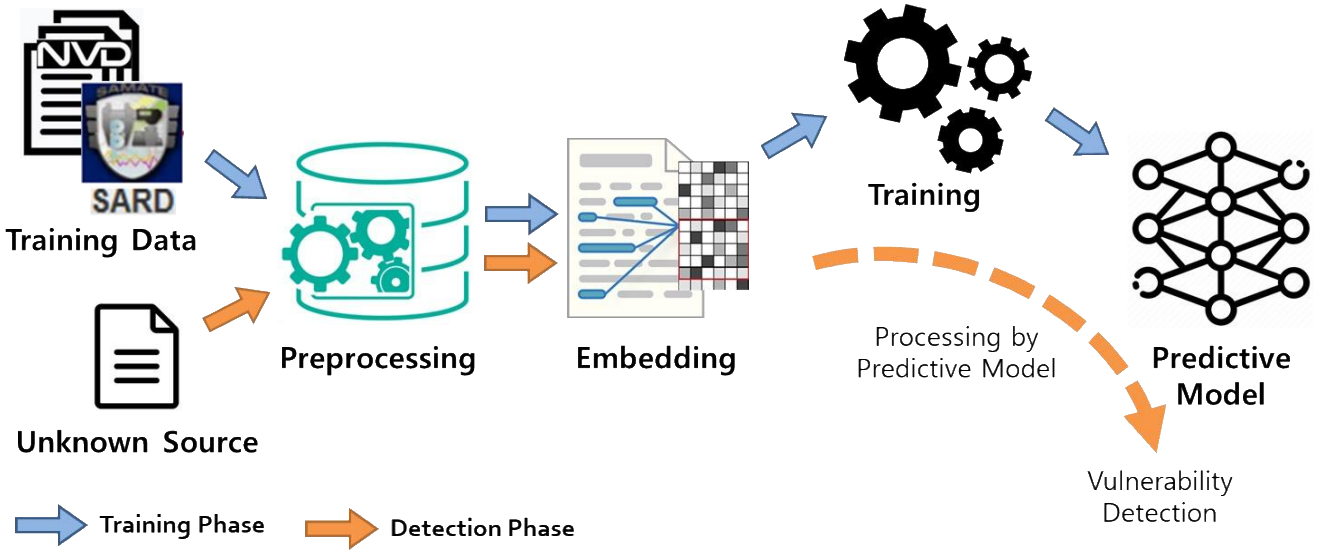
- 기존 Static Analysis 기법들은 **Static Rule을 기반으로 하거나, Code Similarity를 기반으로 하여, 다양한 종류의 Vulnerability에 대한 탐지에 한계점**을 가지고 있음
  - 따라서 다양한 Vulnerability를 Human Expertise 없이 자동으로 탐지할 수 있는 Deep Learning-based Vulnerability Detection 연구가 진행되고 있으나, Source Code를 Embedding Vector화 하는 방법에 대한 연구와 다양한 Neural Network Model의 효율성에 대한 비교 연구가 부족

- 따라서 본 연구에서는 **NVD 및 SARD의 다양한 Project를 Dataset으로 활용**하여 **Source Code를 효과적으로 Embedding Vector화 할 수 있는 방법**을 연구하고, 각 Project 별로 Neural Network Model의 효율성에 대한 비교 분석을 통해 Project에 따라서 어떻게 **Deep Learning-based Vulnerability Detection 기법을 활용하기 위한 Principle을 제공**할 수 있도록 한다.

Deep learning-based Vulnerability Detection

■ Overview

- Dataset
  - NVD based on CVE / SARD based on CWE
- Source code Representation:
  - Program slicing / Tokenization (w/ Symbolization) / Word Embedding
- Neural Network
  - RNN (LSTM, GRU)



Deep learning-based Vulnerability Detection

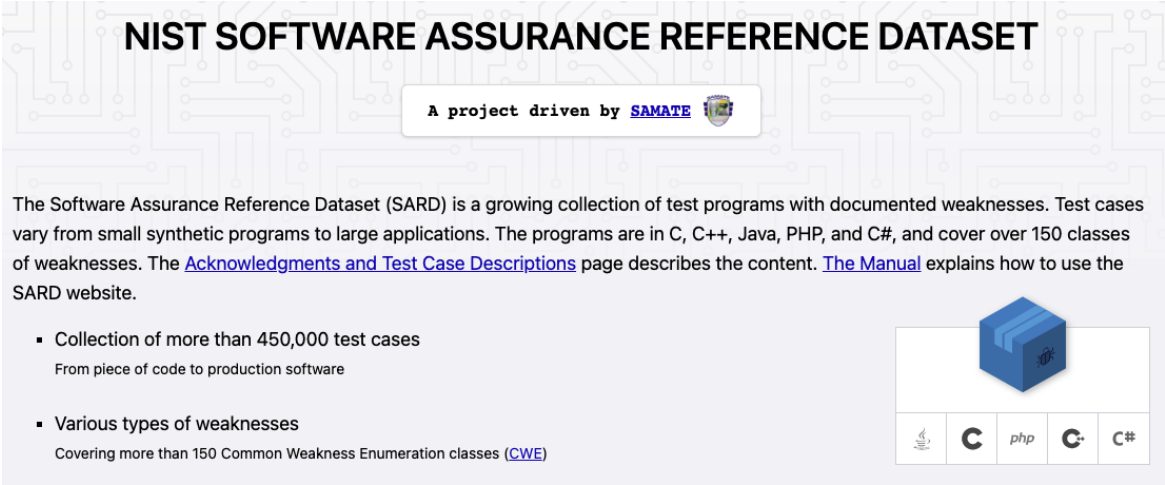
■ Challenges – dataset

- NVD based on CVE
  - The NVD dataset is composed of a source code in the wild, but this is not able to compile.
  - Linux Kernel을 포함한 19개의 Open Source Project의 CVE Patch 정보를 기반으로 총 368개의 Program 확보 (2,167 files) from NVD
    - ✓ Open-Source Project
      - Linux kernel, Firefox, Thunderbird, Seamonkey, firefox_esr, Thunderbird_esr, Wireshark, FFmpeg, Apache Http Server, Xen, OpenSSL, QEMU, Libav, Asterisk, Cups, Freetype, gnuTLS, libvirt, and VLS media player.
  - [Z. Li et al: comparative study of deep learning-based vulnerability detection system]에서 공개한 Program Source 기반으로 확보
    - ✓ https://github.com/VulDeePecker/Comparative_Study
    - ✓ 추가로 vulnerable code website 참고
      - https://www.vulncode-db.com/

Deep learning-based Vulnerability Detection

■ Challenges – dataset (cont'd)

- SARD based on CWE
  - A collection of test cases in the C/C++ language. It contains examples organized under 118 different CWEs.
  - The SARD dataset is composed of synthesized source code; on the other hand, this is a compilable code.
  - SARD dataset을 이용하여 127,891개 file 확보
  - Download
    - ✓ https://samate.nist.gov/SARD/test-suites/112



Deep learning-based Vulnerability Detection

■ Example code of the NVD Dataset

```
static int CVE_2011_3002_VULN_GrowAtomTable(AtomTable *atable, int size)
{
    int *newmap, *newrev;

    if (atable->size < size) {
        if (atable->amap) {
            newmap = realloc(atable->amap, sizeof(int)*size);
            newrev = realloc(atable->arev, sizeof(int)*size);
        } else {
            newmap = malloc(sizeof(int)*size);
            newrev = malloc(sizeof(int)*size);
            atable->size = 0;
        }
        if (!newmap || !newrev) {
            /* failed to grow -- error */
            if (newmap)
                atable->amap = newmap;
            if (newrev)
                atable->arev = newrev;
            return -1;
        }
        memset(&newmap[atable->size], 0, (siz
        memset(&newrev[atable->size], 0, (siz
        atable->amap = newmap;
        atable->arev = newrev;
        atable->size = size;
    }
    return 0;
} // CVE_2011_3002_VULN_GrowAtomTable
```

Diff

```
static int CVE_2011_3002_PATCHED_GrowAtomTable(AtomTable *atable, int size)
{
    int *newmap, *newrev;

    if (atable->size < size) {
        if (atable->amap) {
            newmap = moz_xrealloc(atable->amap, sizeof(int)*size);
            newrev = moz_xrealloc(atable->arev, sizeof(int)*size);
        } else {
            newmap = moz_xmalloc(sizeof(int)*size);
            newrev = moz_xmalloc(sizeof(int)*size);
            atable->size = 0;
        }
        if (!newmap || !newrev) {
            /* failed to grow -- error */
            if (newmap)
                atable->amap = newmap;
            if (newrev)
                                                    newrev;


                                                    size], 0, (size - atable->size) * sizeof(int));
                                                    size], 0, (size - atable->size) * sizeof(int));
                                                    AtomTable
```

Code Gadget

```
55131 CVE-2011-3002/CVE_2011_3002_VULN_GrowAtomTable.c memset 23
static int CVE_2011_3002_VULN_GrowAtomTable(AtomTable *atable, int size) 1
int * newmap , * newrev ; 3
if ( atable -> size < size )  5
if ( atable -> amap )  6
newmap = realloc ( atable -> amap , sizeof ( int ) * size ); 7
newrev = realloc ( atable -> arev , sizeof ( int ) * size ); 8
newmap = malloc ( sizeof ( int ) * size ); 10
newrev = malloc ( sizeof ( int ) * size ); 11
atable -> size = 0; 12
if ( ! newmap || ! newrev )  14
memset ( & newrev [ atable -> size ] , 0 , ( size - atable -> size ) * sizeof ( int ) ); 23
1
```

Deep learning-based Vulnerability Detection

KMU 국민대학교 KOOKMIN UNIVERSITY

■ Example code of the SARD Dataset

```
static int badSource(int data)
{
    {
        char inputBuffer[CHAR_ARRAY_SIZE] = "";
        /* POTENTIAL FLAW: Read data from the console using fgets() */
        if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)
        {
            /* Convert to int */
            data = atoi(inputBuffer);
        }
        else
        {
            printLine("fgets() failed.");
        }
    }
    return data;
}
```

```
34033 62589/CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_42.c fgets 29
static int badSource(int data) 24
char inputBuffer [ CHAR_ARRAY_SIZE ] = "" ; 27
if ( fgets ( inputBuffer , CHAR_ARRAY_SIZE , stdin ) != NULL )  29
data = atoi ( inputBuffer ); 32
return data ; 39
1
```

Code Gadget

```
static void goodG2B()
{
    int data;
    /* Initialize data */
    data = -1;
    data = goodG2BSource(data);
    {
        int i;
        int * buffer = (int *)malloc(10 * sizeof(int));
        /* initialize buffer */
        for (i = 0; i < 10; i++)
        {
            buffer[i] = 0;
        }
        /* POTENTIAL FLAW: Attempt to write to an index of the array that is above the upper bound
         * This code does check to see if the array index is negative */
```

```
34034 62589/CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_42.c fgets 115
static int goodB2GSource(int data) 110
char inputBuffer [ CHAR_ARRAY_SIZE ] = "" ; 113
if ( fgets ( inputBuffer , CHAR_ARRAY_SIZE , stdin ) != NULL )  115
data = atoi ( inputBuffer ); 118
return data ; 125
0
```

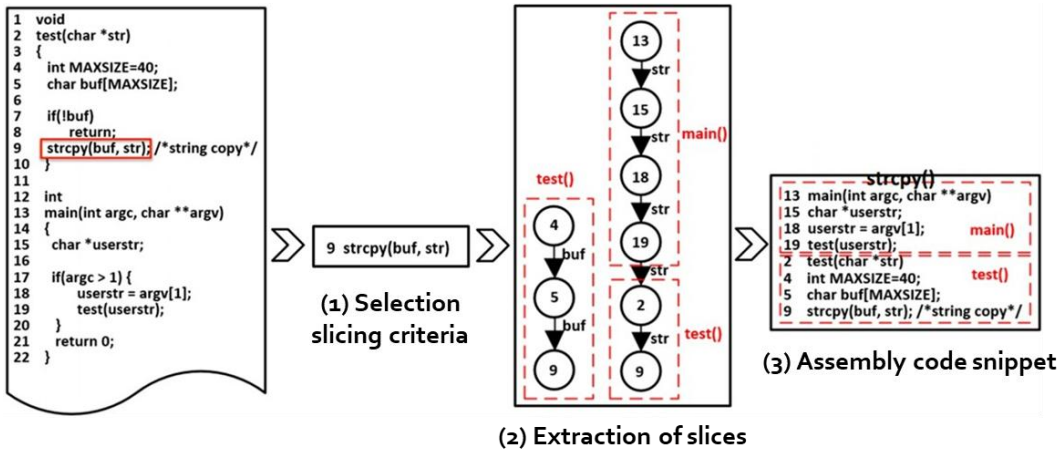
Code Gadget

Deep learning-based Vulnerability Detection

■ Challenges – noise data

- Selection of slicing criteria
  - The first criteria is a list of application programming interface (API) functions where many vulnerability occurrences have bee n reported.
  - The second criteria is a modified code statement that can be verified using the diff between the vulnerability code and the p atched code.
- Extraction of program slices
  - Using Forward and backward slicing with control dependency and data dependency information (*SDG-**IFDS type).

*SDG: Software Dependency Graph
**IFDS: Intraprocedural Finite Distributive Subset

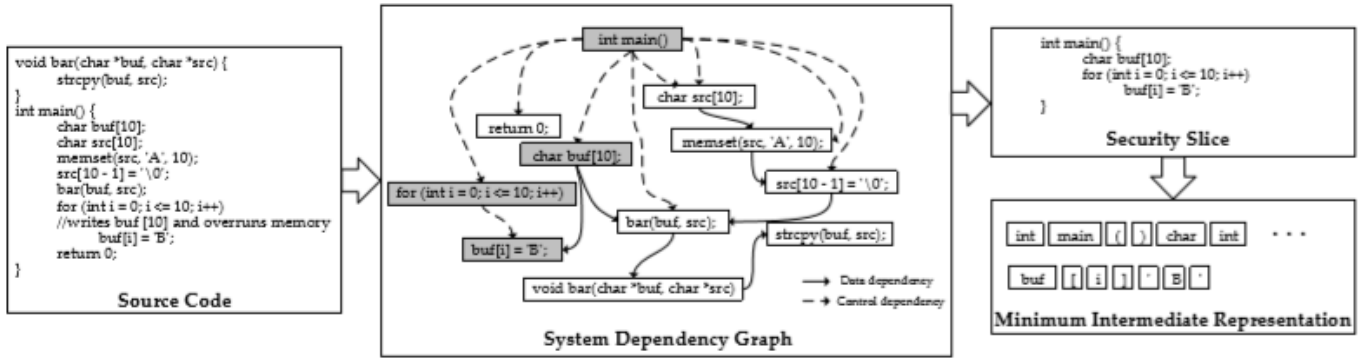


■ Challenges – noise data (cont'd)

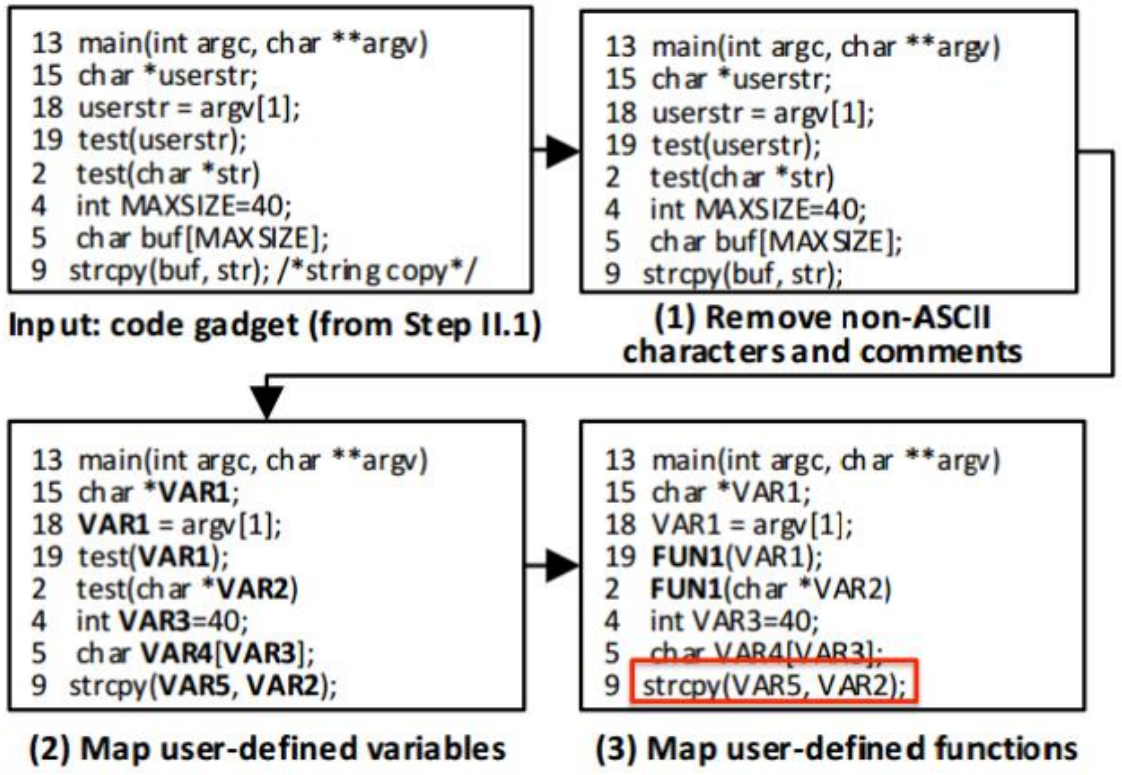
- Slicing Tools
  - LLVM-Slicing
    - ✓ Symbolic program slicing with LLVM
    - ✓ This tool processed program slicing of the SARD dataset.
    - ✓ https://github.com/zhangyz/llvm-slicing
  - Joern
    - ✓ Joern is a platform for analyzing source code, bytecode, and binary executables. It generates code property graphs (CPGs), a graph represent ation of code for cross-language code analysis.
    - ✓ This is not a slicing tool, but the NVD dataset is not compilable source code, so we used this tool as a parsing tool.
    - ✓ https://joern.io/

■ Challenges – Tokenization

- Source code의 token 단위는 문법에 따라 명확하게 구분할 수 있음
- Variable name, User-defined function name ?
  - Name 자체의 의미는 무시하고 VAR#1, FUNC#1 형태로 symbolization을 진행
  - Considering points
    - ✓ Variable Type 정보는 중요하므로 Type에 따라 VAR_STR#1, VAR_INT#1 등으로 type 정보를 포함하여 symbolization 진행
    - ✓ *Function의 return value와 parameter 정보가 중요하므로 해당 정보를 포함하여 symbolization 진행*
    - ✓ *동일한 variable인 경우에도 어떤 함수에서 사용하는지에 따라 vulnerable 여부에 영향을 줄 수 있음*
    - ✓ *Variable Scope의 정보를 포함하여 symbolization 진행*

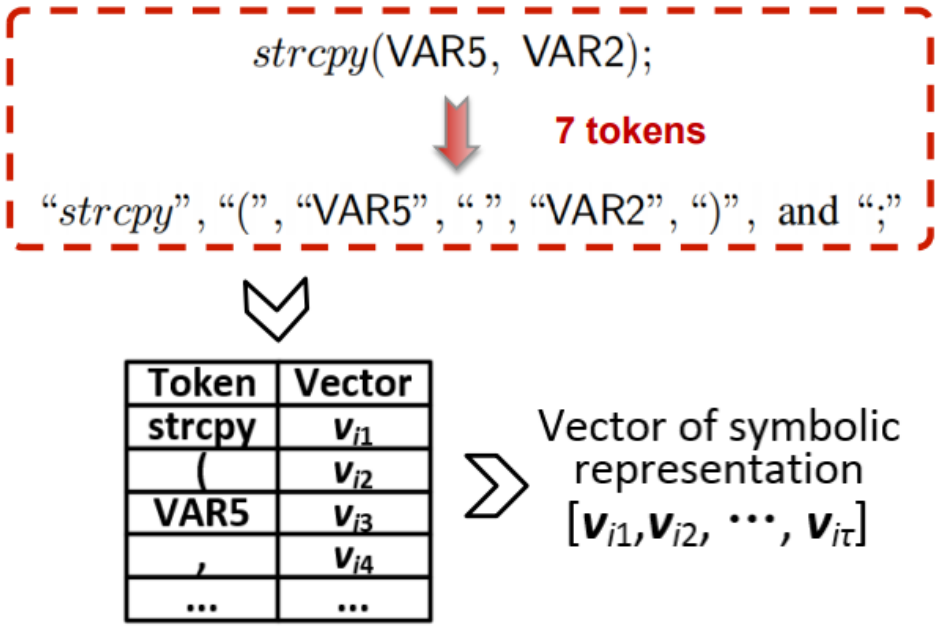


■ Example of Symbolic Representation



Deep learning-based Vulnerability Detection

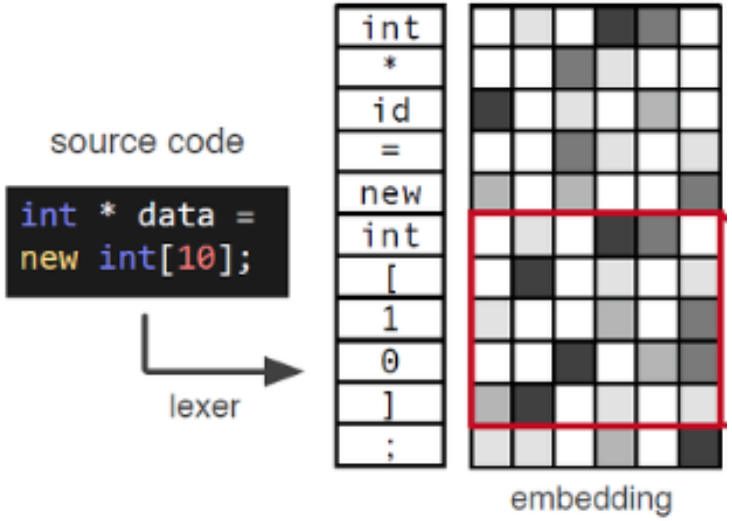
■ **Encode the symbolic representations into vectors**



■ Challenges – code embedding

- Static word-embedding method
  - Applying *Word2Vec, Sent2Vec, Doc2Vec, FastText, Glove and Code2Vec*
  - 각 token의 meaning 정보를 포함하여 embedding vector로 representation 가능
  - Considering point
    - *Context 정보를 포함해야 할까? (contextual word-embedding: BERT, GPT …)*

- Splitting and Padding
  - To make fixed length of input data
  - *Splitting and Padding according to slicing type and slice criteria.*



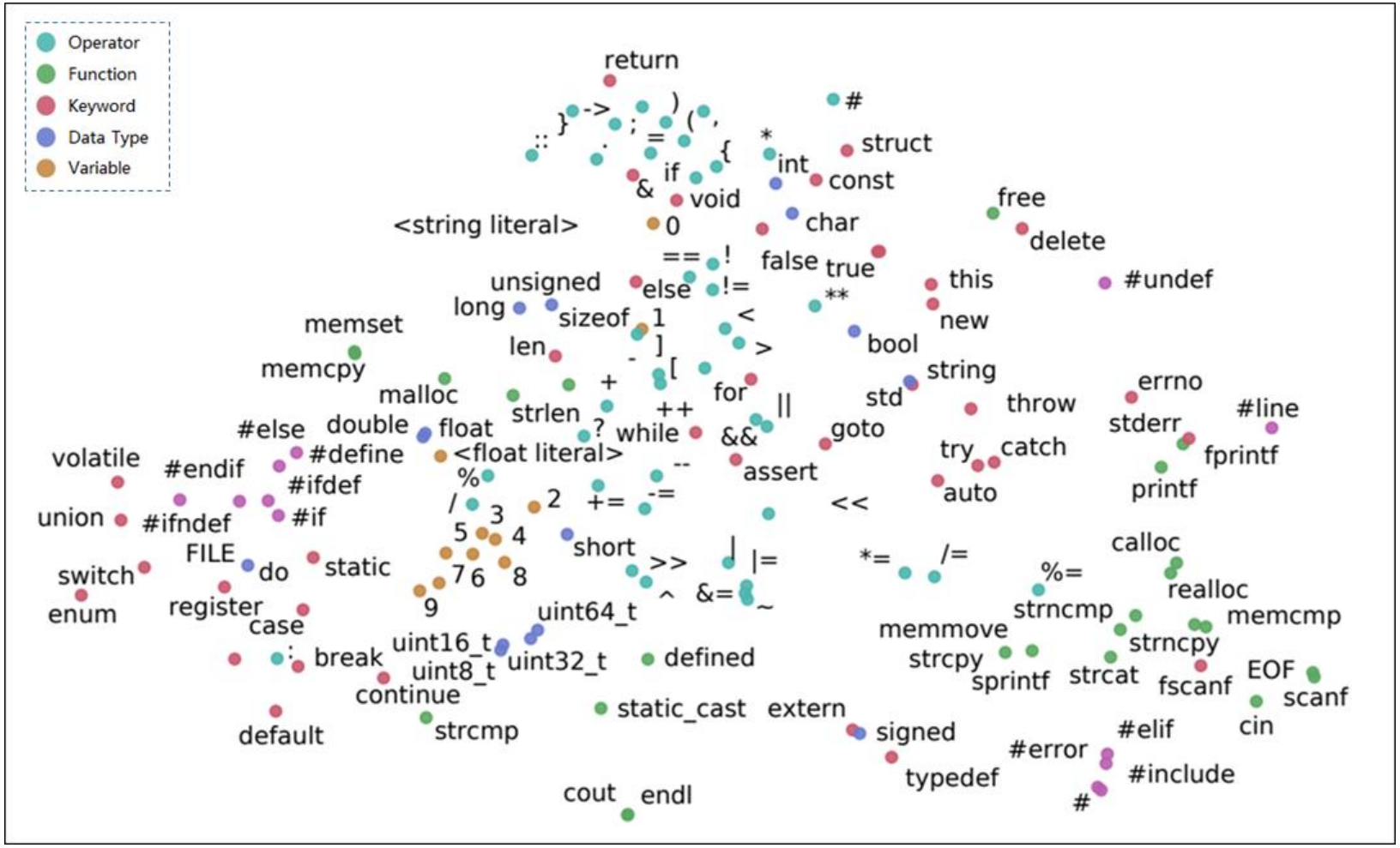
Deep learning-based Vulnerability Detection

■ Word2Vec

- Using gensim package
- Parameters
  - Vector Size: 100
  - Window Size: 5
  - Min count: 5
  - CBOW

■ Sent2Vec & Doc2Vec

- Using gensim package
- Parameters
  - Vector Size: 20
  - Window Size: 5
  - Min count: 5
  - Distributed Memory



■ Challenges – optimal model

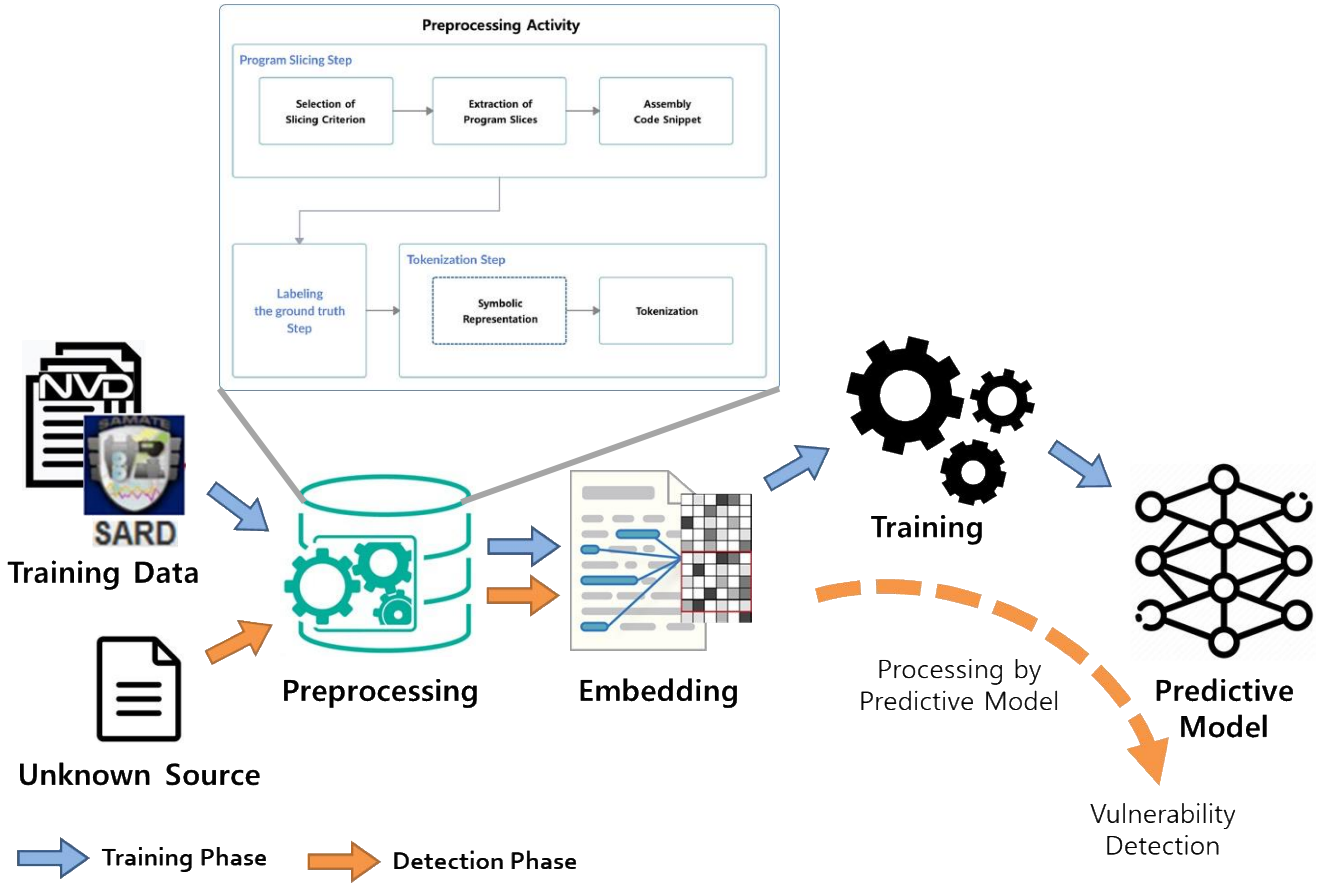
- CNN (Convolutional Neural Network)
  - Source Code를 이미지로 representation 후 CNN에 적용
    - ✓ Information Loss를 최소화하며 Source Code를 이미지로 변경하기 위한 방법이 필요
    - ✓ Sequence 정보를 학습 정보에 활용하기 어려움

- GNN (Graph Neural Network)
  - Program Slicing 결과를 control flow graph (node, edge)로 representation 후 GNN에 적용
    - ✓ 별도의 변경 없이 Program Slicing 결과를 그대로 사용할 수 있음
    - ✓ Vulnerable code와 Patched code의 graph가 동일하게 표현될 수 있음. 즉, data dependency 차이로 인한 정보 활용이 어려움

- **RNN (Recurrent Neural Network)**
  - Program Slicing 결과를 sequence data로 embedding representation 후 RNN (LSTM, GRU 등)에 적용
    - ✓ Source Code를 sequence data 형태의 embedding vector로 변환을 해야 함
    - ✓ Sequence data 형태로 표현되므로 control flow와 data flow 정보 모두 활용 가능 함

Deep learning-based Vulnerability Detection

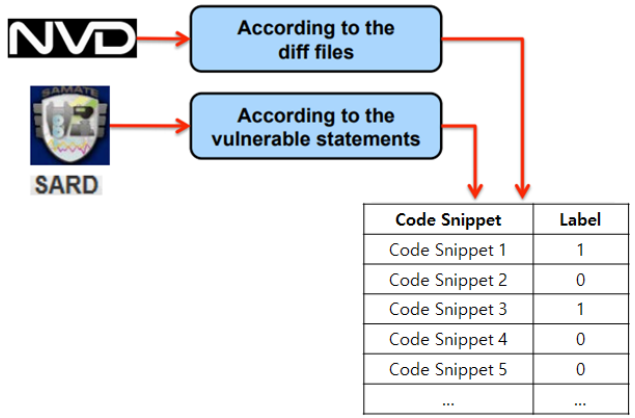
■ Remind Overview



Deep learning-based Vulnerability Detection

■ Environment

- CPU: Intel i7-9700K operating at 3.6GHz
- GPU: NVIDIA GeForce GTX 2070
- Keras with TensorFlow



■ Dataset

- Training Data (80%) : Test Data (20%)
- 5-folding Cross Validation

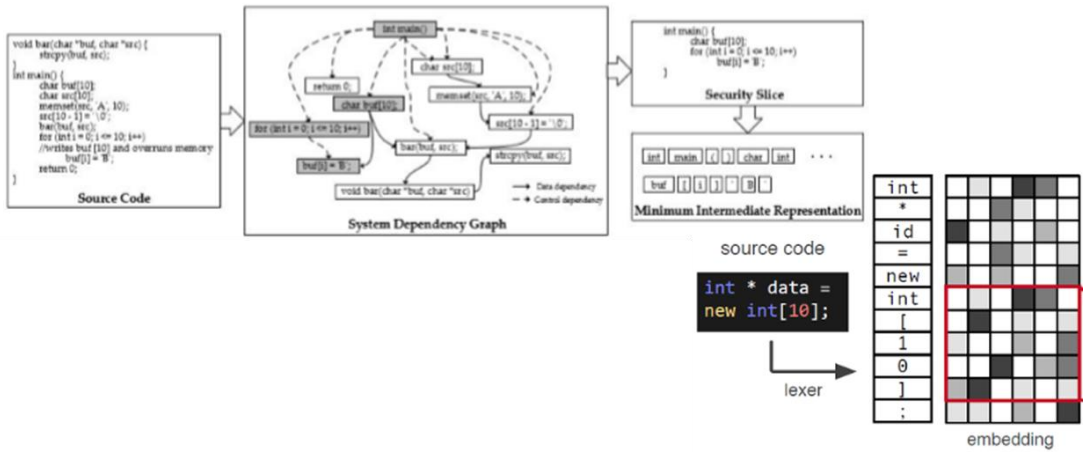
Dataset	#Target Files	#Code Snippets	#Vulnerable	#Benign
SARD	10,655 files	76,667 (79,583)	30,097 (31,252)	46,570 (48,331)
NVD	2,033 files	65,432 (66,164)	14,194 (14,349)	51,238 (51,815)
Total	12,688 files	142,099 (145,747)	44,291 (45,601)	97,808 (100,146)

※ SGD-IFDS based slicing 기준 / Inter (Intra)

Deep learning-based Vulnerability Detection

■ **RQ: What is the optimal encoding method for source code?**

- Source Code를 Encoding하기 위해 Program Slicing, Word Embedding, Symbolization, Padding & Split 등의 기법을 사용하였으며, 어떤 조합이 최적의 성능을 나타내는지 확인하기 위한 실험을 수행하였음
- 각각의 항목들은 독립변수로 취급하여 실험을 설계하였고, 실험 결과를 기반으로 각 항목 간의 관계를 추론하였음
- Default Set
  - Program Slicing: SGD-IFDS (Inter)
  - Word Embedding: Word2Vec
  - Symbolization: Simple
  - Padding & Split: Post
  - Snippet Size: 80



Deep learning-based Vulnerability Detection

■ **RQ: What is the optimal encoding method for source code?**

- **Program Slicing**
  - Scope
    - ✓ Inter-Procedure / Intra-Procedure
  - Type
    - ✓ Weiser: Data dependency
    - ✓ SGD-IFDS: Data dependency + Control dependency

- 범위가 늘어나면 Precision 향상
- 정보가 많아지면 Recall 향상

- **Insight**
  - 유의미한 정보(Data dependency + Control dependency, Inter-Procedure)를 많이 포함하는 Program Slicing 기법을 사용한 경우가 더 좋은 성능을 내고 있음.

Slicing	FPR	FNR	Precision	Recall	F1-Score
Intra + Weiser	5.33%	8.87%	88.11%	91.13%	89.59%
Intra + SGD-IFDS	5.10%	7.19%	88.56%	92.81%	90.63%
Inter + Weiser	4.20%	8.56%	90.68%	91.44%	91.06%
Inter + SGD-IFDS	*3.73%*	*6.61%*	*91.69%*	*93.39%*	*92.53%*

※ Embedding (word2vec), Symbolization (Type-aware), Padding & Split (Combine), Snippet size (80), LSTM

Deep learning-based Vulnerability Detection

■ RQ: What is the optimal encoding method for source code?

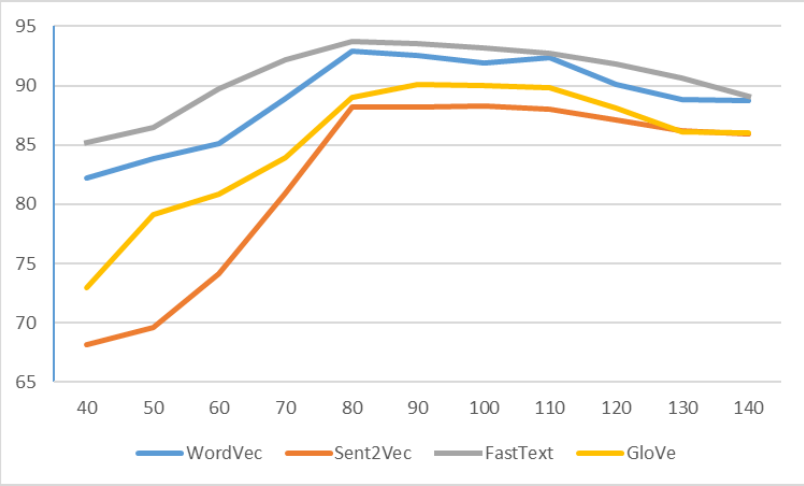
- **Word Embedding**
  - Word2Vec: CBOW, skip-gram
  - Sent2Vec: Extended word2vec (CBOW) for Large Corpus
  - FastText: Extended word2vec (including sub-word)
  - GloVe: Extended word2vec (including co-occurrence)

Word Embedding	FPR	FNR	Precision	Recall	F1-Score
Word2Vec	3.73%	6.61%	91.69%	93.39%	92.53%
Sent2Vec	5.23%	10.84%	88.40%	89.16%	88.78%
FastText	*3.14%*	*5.31%*	*93.02%*	*94.69%*	*93.85%*
GloVe	4.92%	7.61%	88.95%	92.39%	90.64%

※ Slicing (Inter + GDG-IFDS), Symbolization (simple), Post-padding, Snippet size (optimal), LSTM

■ RQ: What is the optimal encoding method for source code?

- **Insight**
  - FastText의 경우 Sub-word에 대한 학습을 포함하여 Corpus내에 없는 Token이 존재하지 않는 경우에도 None 처리가 되는 것이 없기 때문에 제한된 Token으로 구성된 Corpus를 가지고 학습하는 특성에서 가장 좋은 성능을 내는 것으로 추측할 수 있음.
  - 최적의 Snippet Size는 Word Embedding Algorithm에 크게 영향을 받지 않음.



Deep learning-based Vulnerability Detection

■ **RQ: What is the optimal encoding method for source code?**

- **Symbolization**
  - None: None Symbolization
  - Simple: VAR#, FUNC#
  - Type-aware: VAR_CHAR#, VAR_INT#, VAR_BOOL#, VAR_STRUCT#, FUNC#
- **Insight**
  - Symbolization 유무의 효과는 Embedding시 좀더 유의미한 결과를 나타낼 수 있도록 도와주기 때문에 좋은 성능을 나타내는 것으로 판단되며, Type-aware와 Simple은 미세한 성능 차이기는 하지만, Dataset의 크기가 커지면 성능 향상 효과가 더욱 크게 나타날 것으로 판단됨.

Symbolization	FPR	FNR	Precision	Recall	F1-Score
None	6.21%	10.04%	86.02%	89.96%	87.95%
Simple	3.73%	6.61%	91.69%	93.39%	92.53%
Type-aware	*3.44%*	*6.09%*	*92.35%*	*93.91%*	*93.13%*

※ **Embedding (word2vec), Slicing (Inter + GDG-IFDS), Post-padding, Snippet size (80), LSTM**

■ RQ: What is the optimal encoding method for source code?

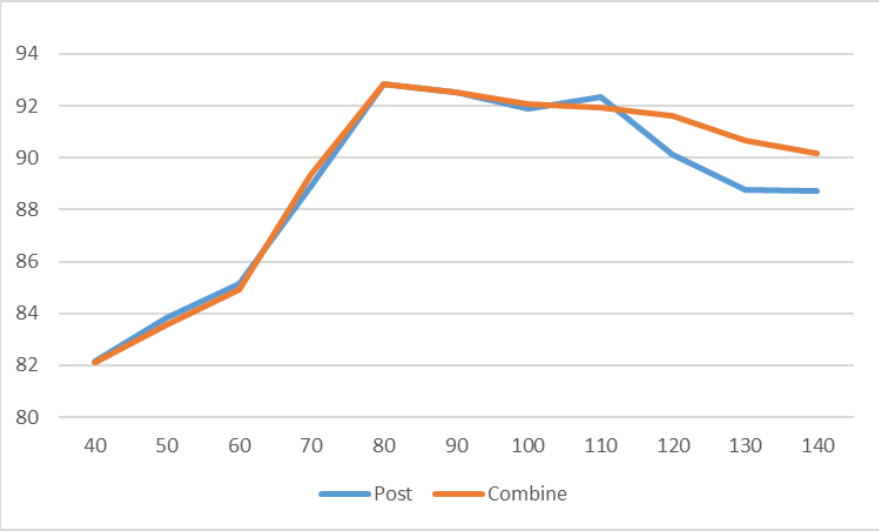
- **Padding & Split**
  - Post: Post Padding & Split
  - Combine:
    - ✓ forward slicing으로만 구성된 Snippet의 경우, post-padding (<size), post-split (>size)
    - ✓ backward slicing으로만 구성된 Snippet의 경우, pre-padding (<size), pre-split (>size)
    - ✓ backward, forward slicing Snippet이 모두 포함된 경우 slicing criterion이 가장 많이 위치하도록 slicing하고, padding은 post 방식을 적용

Padding & Split	FPR	FNR	TPR	Precision	F1-Score
Post	3.73%	6.61%	91.69%	93.39%	92.53%
Combine	3.75%	6.59%	91.66%	93.41%	92.53%

※ Embedding (word2vec), Symbolization (simple), Slicing (Inter + GDG-IFDS), Snippet size (80), LSTM

■ RQ: What is the optimal encoding method for source code?

- **Insight**
  - Optimal Snippet Size에서는 Padding & Split 방식에 의한 성능 차이가 크지 않았고, Snippet Size가 큰 경우에는 상대적으로 Combine 방식의 효과가 크게 나타남을 확인할 수 있음. → padding & split에 의한 효과는 optimal snippet size에 의해 가려지는 것으로 추정됨.



■ Discussion

- Dealing with several language
  - 본 연구는 C언어를 Target Language로 하고 있음

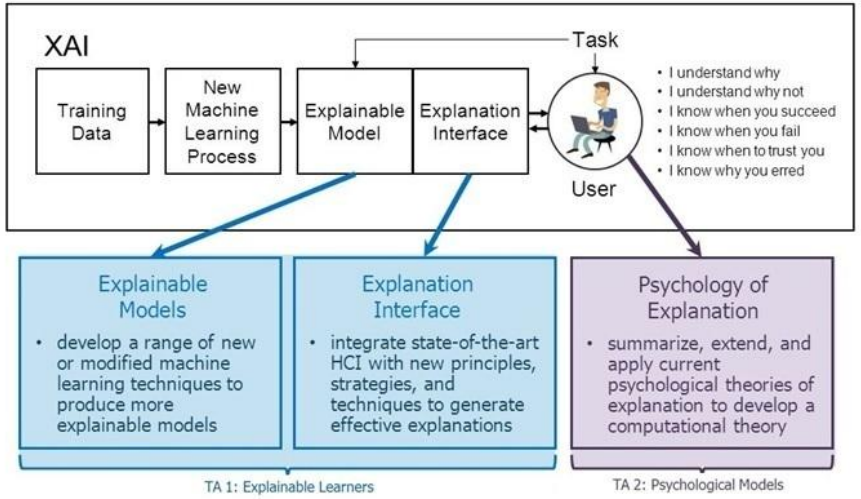
  - *Target Language를 확장하려면…*
    - ✓ Dataset 확보 ← 가장 중요함!
    - ✓ Program Slicing 기법 변경 ← 문법을 고려해야 함!
    - ✓ 나머지는 동일하게 적용 가능

  - *Binary Image를 대상으로 하려면…*
    - ✓ Binary Image를 Disassemble 하여 Asm2Vec 적용
      - 그러나, 여전히 Obfuscation에 취약하다는 문제점이 있음
      - S. H. H. Ding, B. C. M. Fung and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," *2019 IEEE Symposium on Security and Privacy.*

  - 즉, Dataset을 확보하고, embedding vector로 representation 가능하다면 several language 적용 가능!

■ Discussion (cont'd)

- Finding defect origin with XAI
  - Fine-grain vs. Coarse-grain
    - ✓ Fine grain 분석의 경우 detection 후 defect origin을 찾기 쉬움. 그러나 분석 성능이 떨어지는 문제가 있음
    - ✓ 결국 높은 성능을 달성하기 위해서는 Coarse-grain 분석이 필요한데, 이 경우 detection 후 defect origin을 찾기 어려운 문제가 있음
  - 따라서 XAI 기술을 적용하여 detection 후 defect origin이 어디인지 찾아주거나 vulnerability 분석을 위한 정보를 제공하는 추가 연구가 필요함



출처: https://www.darpa.mil/program/explainable-artificial-intelligence

Deep learning-based Vulnerability Detection

■ Key Takeaways

- Gathering Dataset
  - 과제 목표에 맞는 dataset 확보가 가장 중요함

- Eliminating Noise Data
  - Real world source code의 경우 전체 코드 대상으로 하는 경우 compile 가능하지만, 상대적으로 많은 noise 정보가 포함될 가능성 있음 → program slicing 기법의 고도화가 필요함
  - 일부 코드만 대상으로 하는 경우 (본, 연구의 NVD dataset의 경우와 동일), 상대적으로 program slicing을 적용하기 어려움.

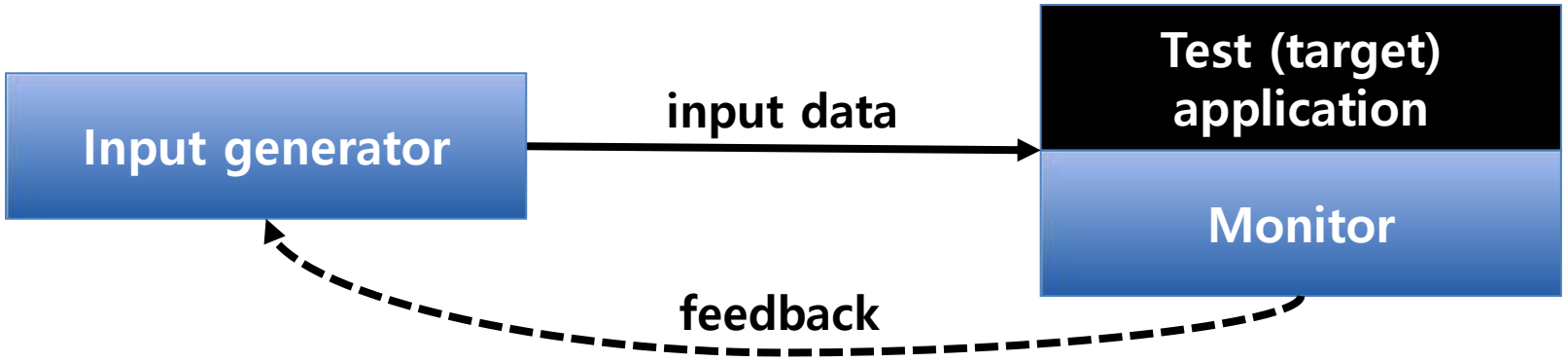
- Using Advanced Embedding Method & DL Model
  - Source Code의 Context 정보가 중요하다면, BERT, GPT 등 attention 계열의 NLP model 적용 가능

03

Dynamic Approach

Fuzzing Overview – definition

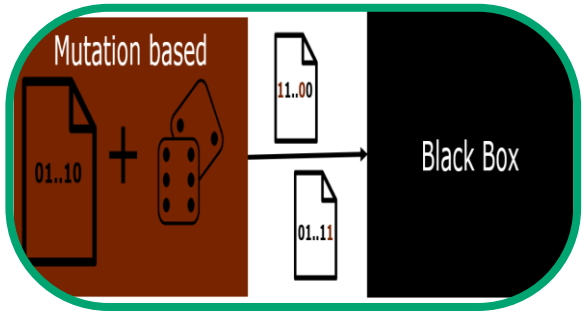
- Fuzzy + Testing ➔ Fuzzing
- *Automatically generate* test cases
- Many *slightly anomalous test cases* are input into a target
- Application is *monitored* for errors.
- Inputs are generally either *file based* (pdf, png, wav, etc.) or *network based* (http, SNMP, etc.)



Fuzzing Overview – Evolution Timeline

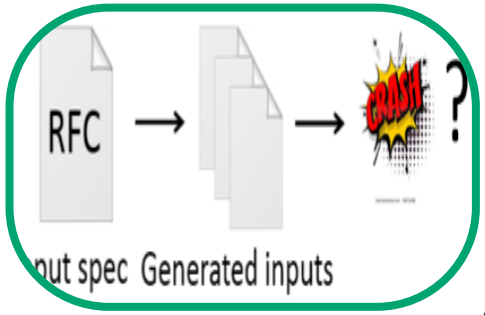
Dumb Fuzzing
Random input generation without format awareness

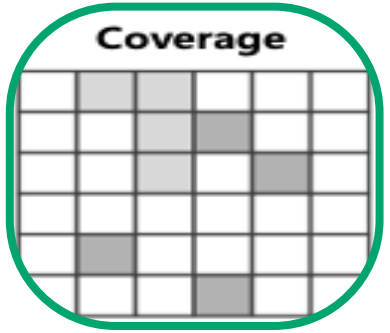
Barton Miller et al. '89



Mutation based Fuzzing
Modify existing inputs

Generation based Fuzzing
Generate from protocol/format

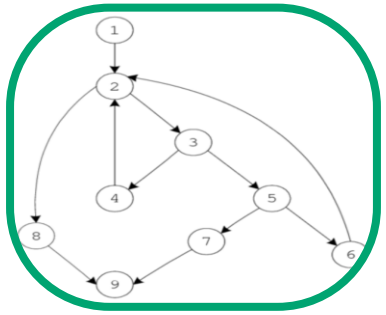




Coverage guided Fuzzing
Feedback from code coverage

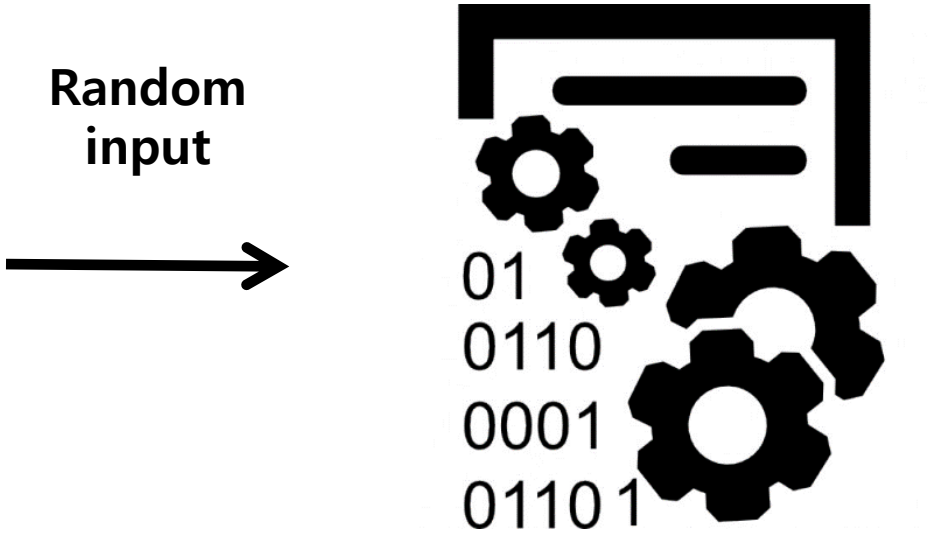
American Fuzzy Lop (AFL)

Dataflow guided Fuzzing
Leverages internal dataflow to refine fuzzing



Fuzzing Overview – Blackbox Fuzzing

Random input



Test Program

Barton Miller et al. ‘89

Fuzzing Overview – Blackbox Fuzzing

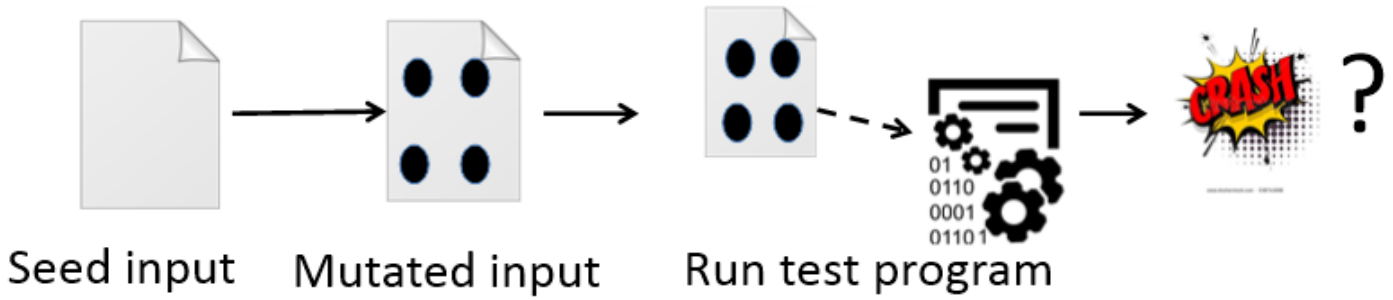
- **Given a program *simply feed random inputs* and see *whether it exhibits incorrect behavior* (e.g., crashes)**

- **Advantage: *easy, low programmer cost***

- **Disadvantage: inefficient**
  - Inputs often require structures; random inputs *are likely to be malformed*
  - Inputs that trigger an incorrect behavior is a *very small fraction*, probably of *getting lucky is very low.*

Fuzzing Overview – Mutation-based Fuzzing

- Take a ***well-formed input, randomly perturb*** (flipping bit, etc.)
- ***Little or no the structure knowledge*** of the inputs is assumed.
- Anomalies are ***added to existing valid inputs***
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Fuzzing Overview – Mutation-based Fuzzing

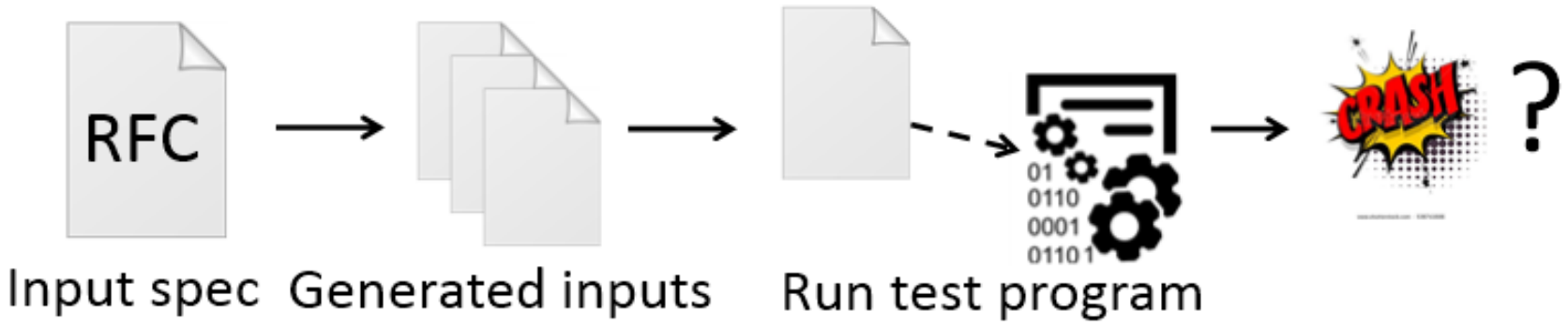
- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus (seed)
- May fail for protocols with checksums, those which depend on challenges

Fuzzing Overview – Generation-based Fuzzing

- **Test cases are generated from some description of *the input format*: RFC, documentation, etc.**
  - Using specified protocols/file format information.

- **Anomalies are added to *each possible spot* in the inputs**

- **Knowledge of protocol should give *better results than random fuzzing***



Fuzzing Overview – Mutation-based vs. Generation-based

■ Mutation-based fuzzer

- Pros: easy to set up and automate, little to no knowledge of input format required
- Cons: limited by initial corpus, may fall for protocols with checksums and other hard checks

■ Generation-based fuzzer

- Pros: completeness, can deal with complex dependencies (e.g., checksum)
- Cons: writing generators is hard, performance depends on the quality of the spec.

Fuzzing Overview – How much fuzzing is enough?

- Mutation-based fuzzers may generate an *infinite number of test cases*. When has the fuzzer run long enough?

- Generation-based fuzzers may generate a *finite number of test cases*. What happens when they are all run, and no bugs are found?

■ Properties of Security Assessment (test)

- Assessment resources (test engineer, time, etc.) are finite.
- The security assessment does not guarantee that the target project is perfectly secure. Instead, it ensures the target project is secure in specific scenarios (test case).

Need to improve test efficiency
(speed, coverage, etc.)

Fuzzing Overview – Code coverage

- **Some of the answers to these questions lie in code coverage.**

- **Code coverage is a metric that can be used to determine *how much code has been executed*.**

- **Coverage data can be obtained using a variety of profiling tools.**
  - E.g., gcov, lcov

■ **Line/block coverage**

- measures how many lines of source code have been executed.

■ **For the code on the right, how many test cases (values of pair (a, b)) Needed for full (100%) line coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


Fuzzing Overview – Code Coverage (Branch Coverage)

■ **Branch coverage**

- Measures how many branches in code have been taken (conditional jmps)

■ **For the code on the right, how may test cases needed for full branch coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


■ **Path coverage**

- Measures how many paths have been taken

■ **For the code on the right, how many test cases needed for full path coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


Fuzzing Overview – Benefits of code coverage

■ **Can answer the following questions**

- How good is an initial file?
- *Am I getting stuck somewhere?*
- How good is fuzzer X versus fuzzer Y?
- Am I getting benefits by running multiple fuzzers?

Fuzzing Overview – Problems of code coverage

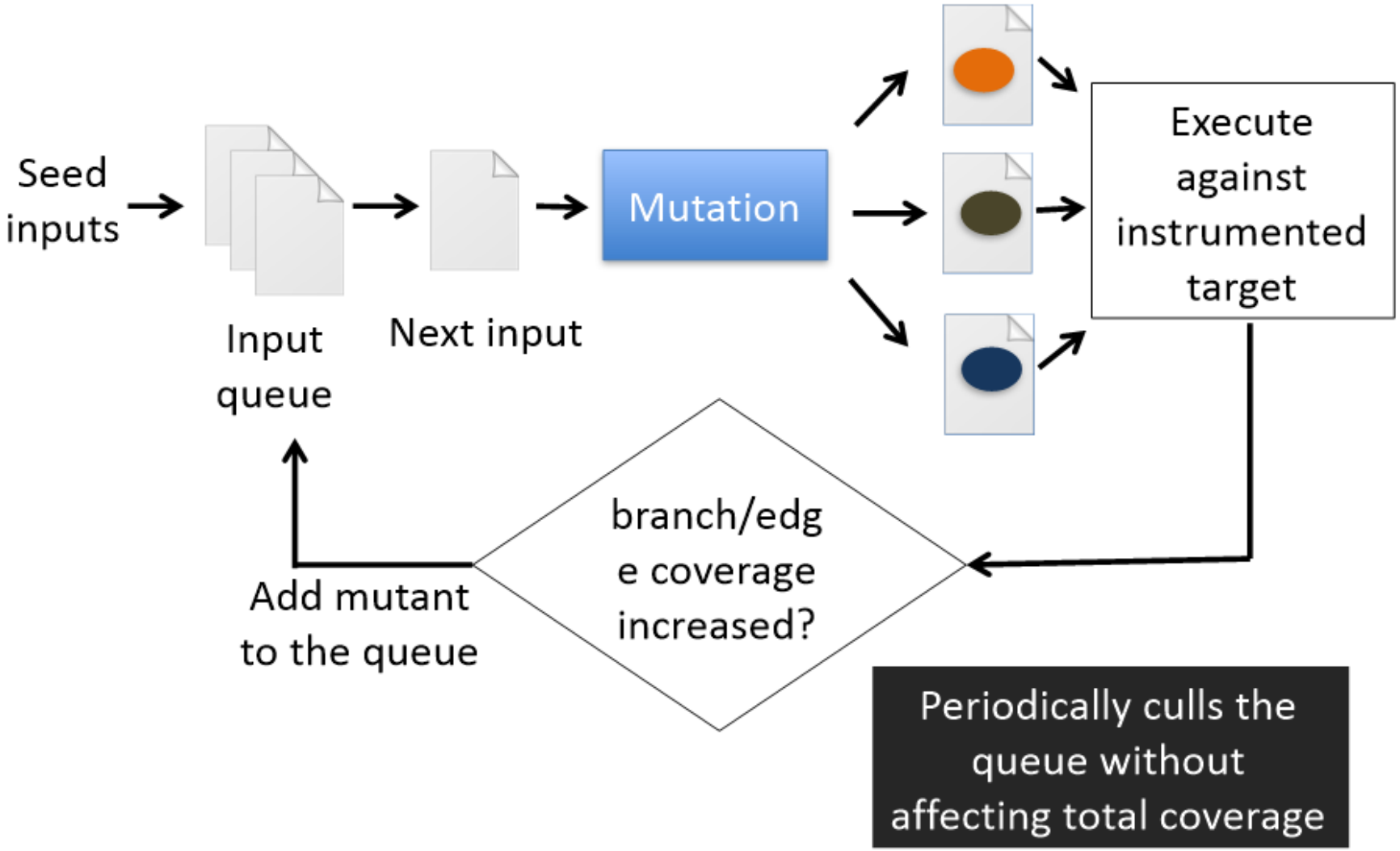
- Does full line coverage guarantee find the bug?
- Does full branch coverage guarantee find the bug?
- Does full path coverage guarantee find the bug?

```
void mySafeCopy(char *dst, uint32 dstLen, char* src, uint32 srcLen)
{
        if (dstLen <= 0 || srcLen <= 0) return;
        if (dst && src) strcpy(dst, src);
}
```

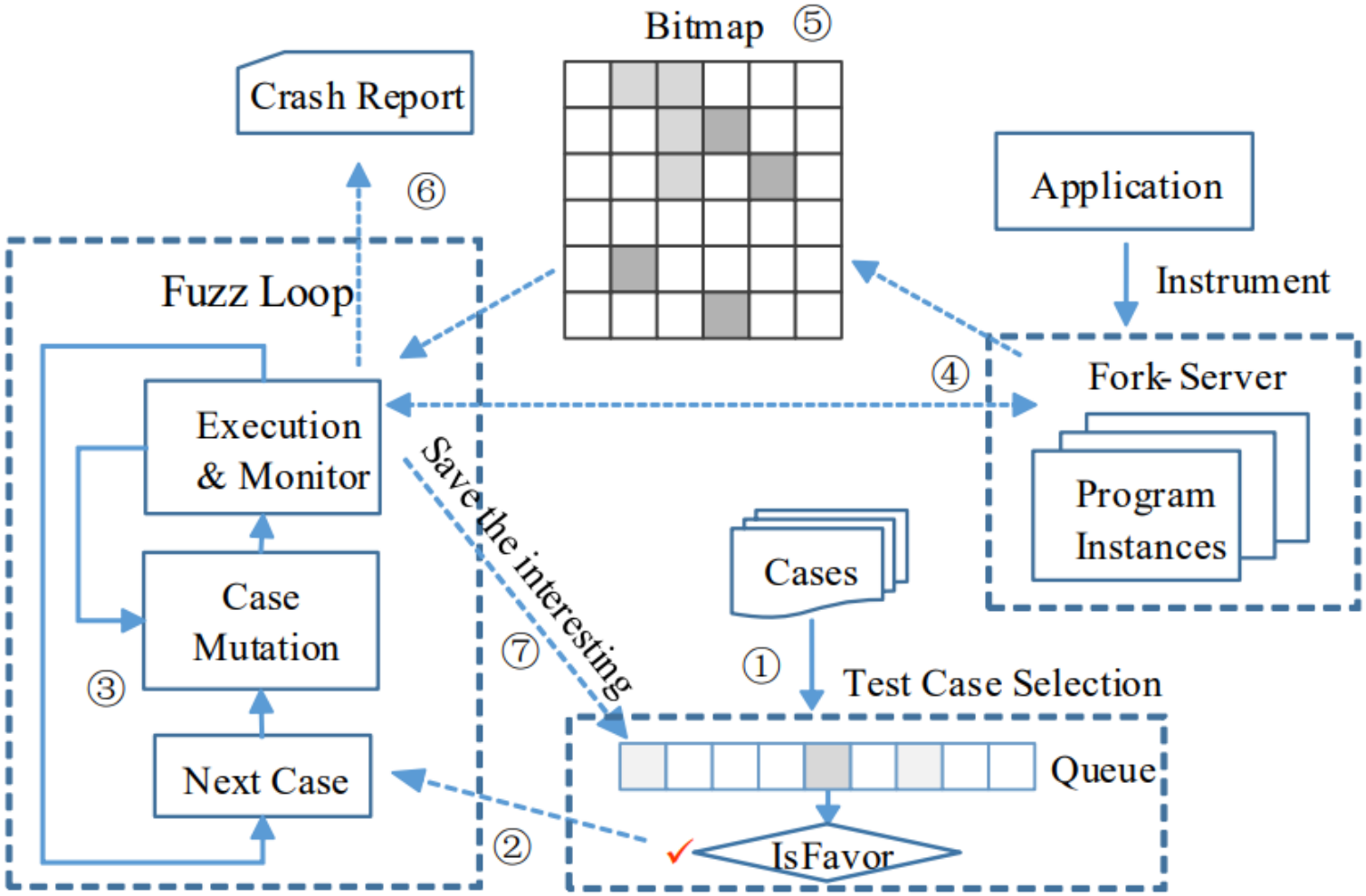
■ Special type of mutation-based fuzzing

- Run mutated inputs on instrumented program and measure code coverage
- Search for *mutants that result in coverage increase*
- Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases.
  - E.g., AFL, Libfuzzer

Fuzzing Overview – American Fuzzy Lop (AFL)



Fuzzing Overview – AFL Fuzzing Process



Wang, Yanhao, et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization."

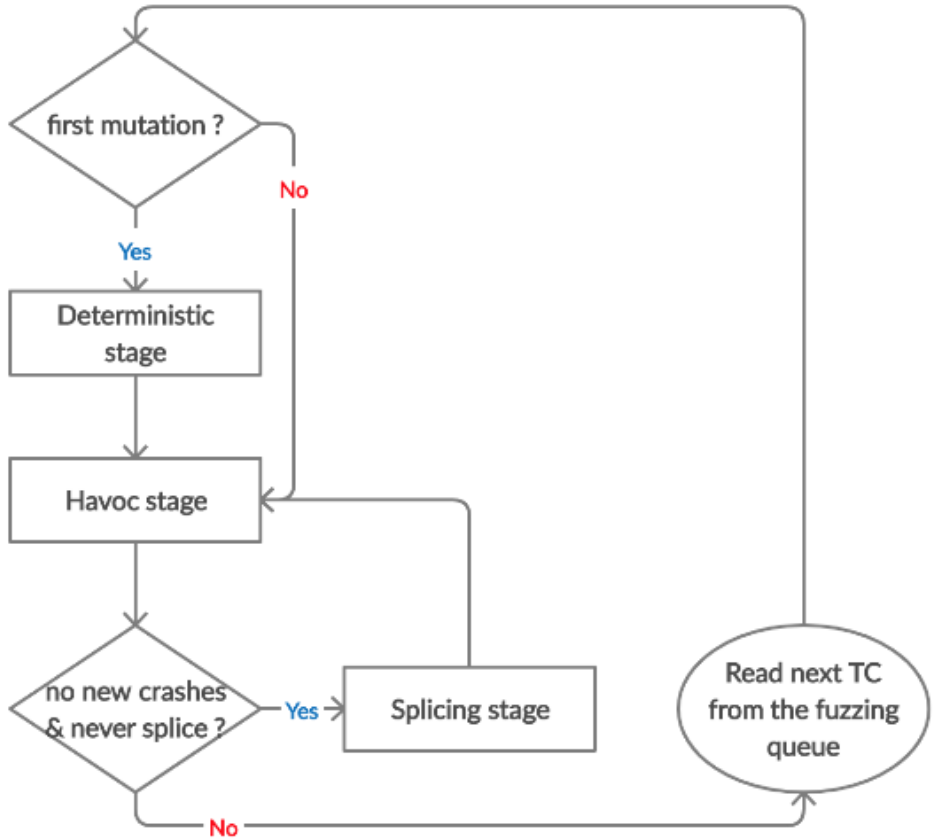
- 2가지 단계로 구성
  - Instrumentation Phase
    - Preliminary analysis
    - Runtime execution feedback
  - Fuzzing Loop Phase
    - Input prioritization
- 용어 정리
  - ***Queue*** : sample queue
  - ***CrashSet*** : a set of crashes
  - ***CovFb, AccountingFb*** : coverage/accounting feedback map
  - ***accCov, accAccounting*** : global accumulated structures to hold all covered transitions and their maximum hit counts for each feedback map
  - ***TopCov, TopAccounting*** : a hash map with edges as keys and inputs as values to maintain the best input for each edge

Favored 값이 지정된 seed는 100%, 아닌 경우 1% 확률로 퍼징

Algorithm 1 Fuzzing algorithm with coverage accounting

```
1:  function FUZZING(Program, Seeds)
2:      P ← INSTRUMENT(Program, CovFb, AccountingFb)        ▷ Instr. Phase
3:      // AccountingFb is FunCallMap, LoopMap, or InsMap
4:      INITIALIZE(Queue, CrashSet, Seeds)
5:      INITIALIZE(CovFb, accCov, TopCov)
6:      INITIALIZE(AccountingFb, accAccounting, TopAccounting)
7:      // accAccounting is MaxFunCallMap, MaxLoopMap, or MaxInsMap
8:      repeat                                              ▷ Fuzzing Loop Phase
9:          input ← NEXTSEED(Queue)
10:         NumChildren ← MUTATEENERGY(input)
11:         for i = 0 → NumChildren do
12:             child ← MUTATE(input)
13:             IsCrash, CovFb, AccountingFb ← RUN(P, child)
14:             if IsCrash then
15:                 CrashSet ← CrashSet ∪ child
16:             else if SAVE_IF_INTERESTING(CovFb, accCov) then
17:                 TopCov, TopAccounting ←
18:                     UPDATE(child, CovFb, AccountingFb, accAccounting)
19:                 Queue ← Queue ∪ child
20:             end if
21:         end for
22:         CULL_QUEUE(Queue, TopCov, TopAccounting)
23:     until time out
24: end function
```


- **The first time to mutate this test case ?**
- **If yes, Deterministic stage**
  - bitflip, byteflip, arithmetic inc/dec, interesting values,
  - auto extras, user extras.
- **If no, Havoc stage**
  - bitflip, byteflip, arithmetic inc/dec, interesting values,
  - random byte, delete bytes, insert bytes, overwrite bytes
- **AFL mutate all the TCs in the queue but discovers no crashes or path && this TC has not entered splicing stage for this time ?**
- **If yes, Splicing stage**
  - cross over
- **If no, read next TC from the fuzzing queue**



■ **How to seed a fuzzer?**

- Seed inputs must cover different branches
- Remove duplicate seeds covering the same branches
- Small seeds are better → Why?

■ **Some branches might be very hard to get past as the # of inputs satisfying the conditions are very small**

- Manually / automatically transform / remove those branches

■ Hard to fuzz code

```
void test (int n) {
  if (n==0x12345678)
     crash();
}
```

needs 2^32 or 4 billion attempts
In the worst case

- The probability to generate feasible inputs is very low

```
1  int main() {
2    unsigned v, w, x, y, z = input();
3    if(x == 3 && y == 4 && z*z < 40000) {
4      if (z > 195) {
5        ......
6        // crash 1
7      }
8      if(15 <= z+v && z+v <= 25) {
9        ......
10       // crash 2
11     }
12   }
13 }
```

Line 3:

$$x==3 \;\&\&\; y==4 \;\&\&\; z*z < 40000$$

seed 1: (v, x, y, z) = (100, 3, 4, 20)

The probability for a mutated input satisfying the condition at Line 3:

$$\underset{<10^{-9}}{x}\quad \underset{<10^{-9}}{y}\quad \underset{<10^{-7}}{z}$$

$$1/2^{32} * 1/2^{32} * 200/2^{32} < 10^{-25}$$

Fuzzing Overview – Dataflow guided Fuzzing

- **A.K.A., Hybrid fuzzing**
  - *Combine* coverage-guided fuzzing *with taint-analysis or symbolic execution*, and so on.

- ***Intercept the data flow*, analyze the inputs of comparisons**
  - Incurs extra overhead

- **Modify the test inputs, observe the effect on comparisons**

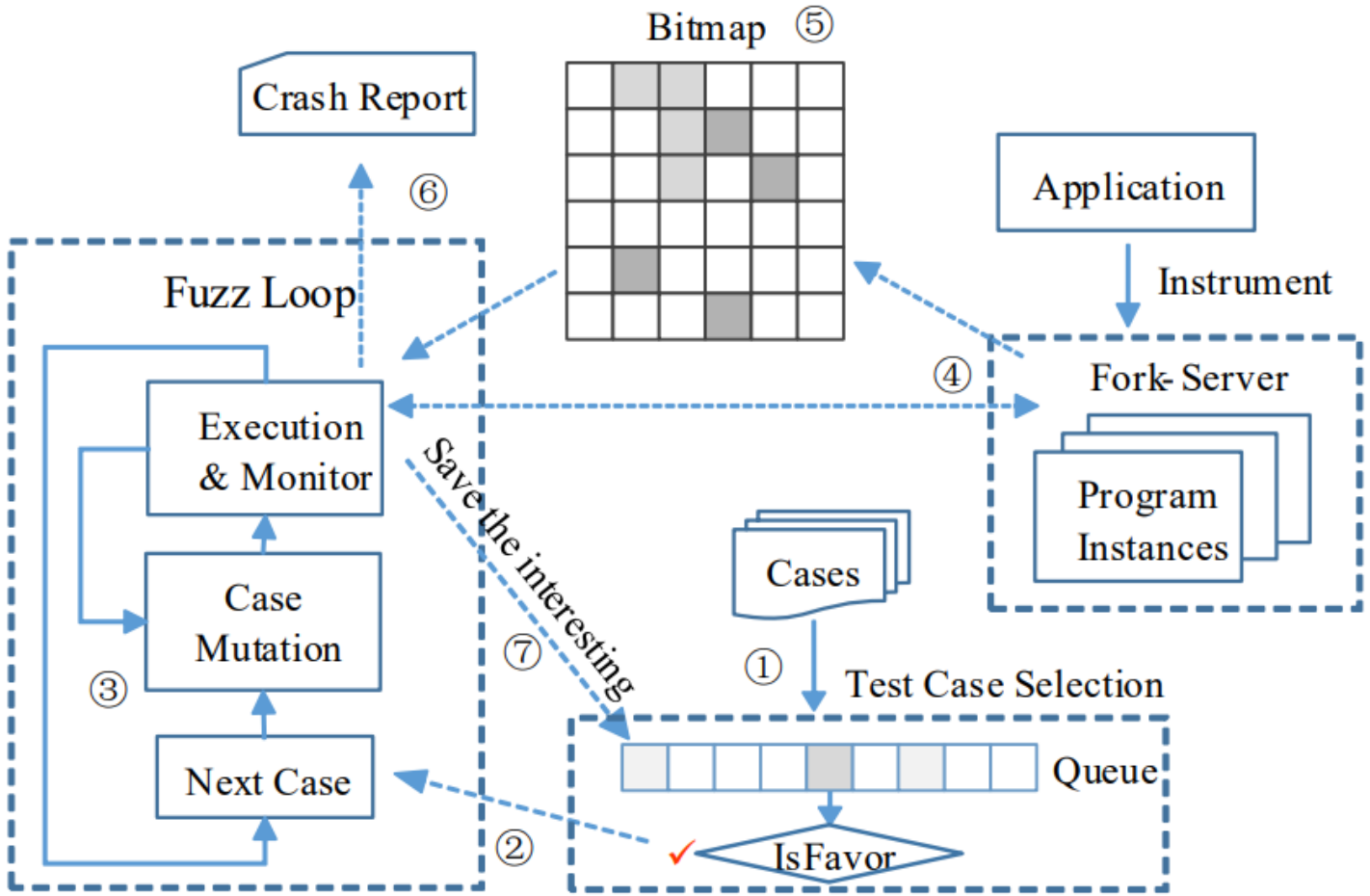
- **Prototype implementations in T-fuzz, and VUzzer.**

- **Input-format knowledge is very helpful.**
- **Generational tends to beat random, better specs make better fuzzers.**
- **Each implementation will vary different fuzzers find different bugs.**
  - More fuzzing with is better → EnFuzz
- **The longer you run, the more bugs you may find.**
  - But it reaches a plateau and saturates after a while
- **Best results come from guiding the process.**
- **Notice where you are getting stuck, use profiling (gcov, lcov)!**

■ Instrumentation

- Execution Feedback
  - Where feedback ?
    - ✓ Source code, Binary Code
  - What feedback?
    - ✓ Edge coverage (Branch coverage)
    - ✓ Condition/Decision, MCDC …
  - How new path ?
    - ✓ Hybrid
    - ✓ Symbolic Execution, Taint Analysis

- In-memory (Persistent) Fuzzing
  - When fork ?
    - ✓ Enable to skip start up code
    - ✓ Need to skip initial I/O routine
  - How reproduce ?
    - ✓ Hard to reproduce error



AI-based SMART Fuzzing – EnFuzz (Ensemble Fuzzing)

■ Motivation

- Existing Base Fuzzers are optimized to be Application Specific, so they show optimal performance in certain applications but low performance in other applications.
- This paper defined stable performance as the Generalization Ability.
- In other words, this paper showed that generalization ability is improved when Ensemble Fuzzing is performed, and better performance than simply performing a base fuzzer simultaneously.

■ Solution

- Base fuzzer selection
- Ensemble architecture

■ Ensemble Effectiveness

```
void crash(char* A, char* B){
    if (A == "Magic Str"){              => T1
        if (B == "Magic Num") {
            bug();                      => T4
        }else{
            normal();                   => T3
        }
    }else if (A == "Magic Num"){        => T2
        if (B == "Magic Str"){
            bug();                      => T5
        }else{
            normal();                   => T6
        }
    }
}
```

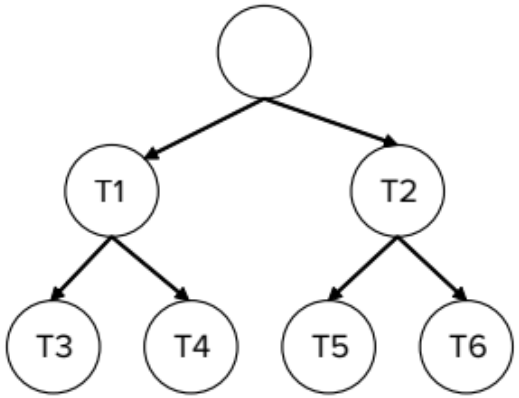


Table 1: covered paths of each fuzzing option

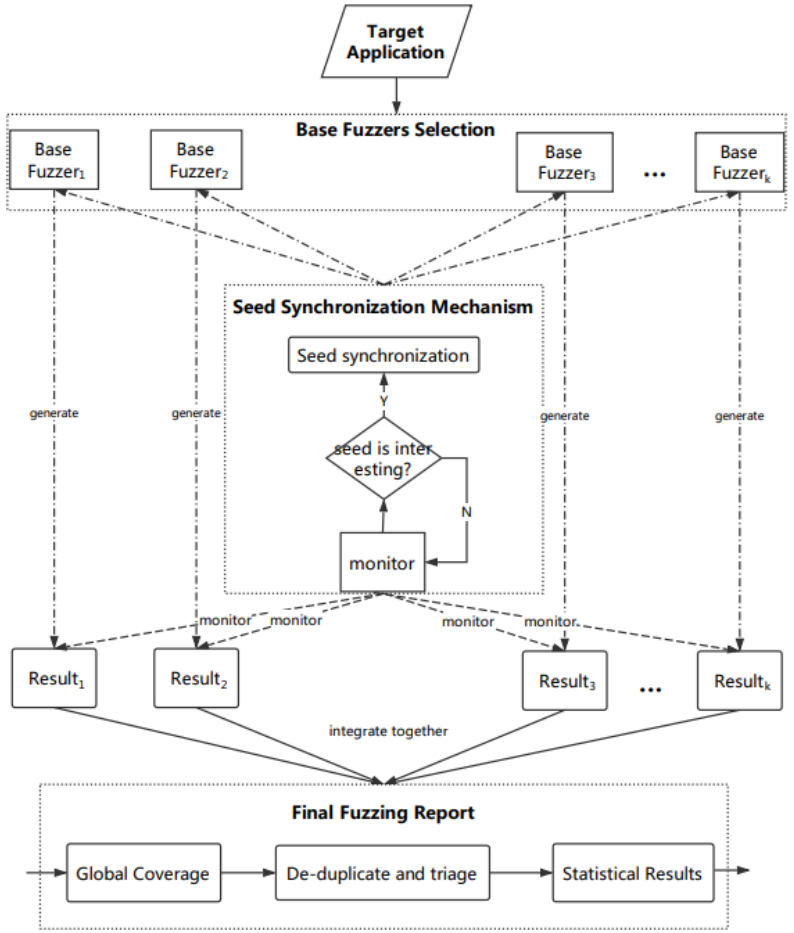
Tool	T1-T3	T1-T4	T2-T5	T2-T6
fuzzer1	✓			
fuzzer2				✓
ensemble fuzzer1 and fuzzer2 without seed synchronization	✓			✓
ensemble fuzzer1 and fuzzer2 with seed synchronization	✓	✓	✓	✓

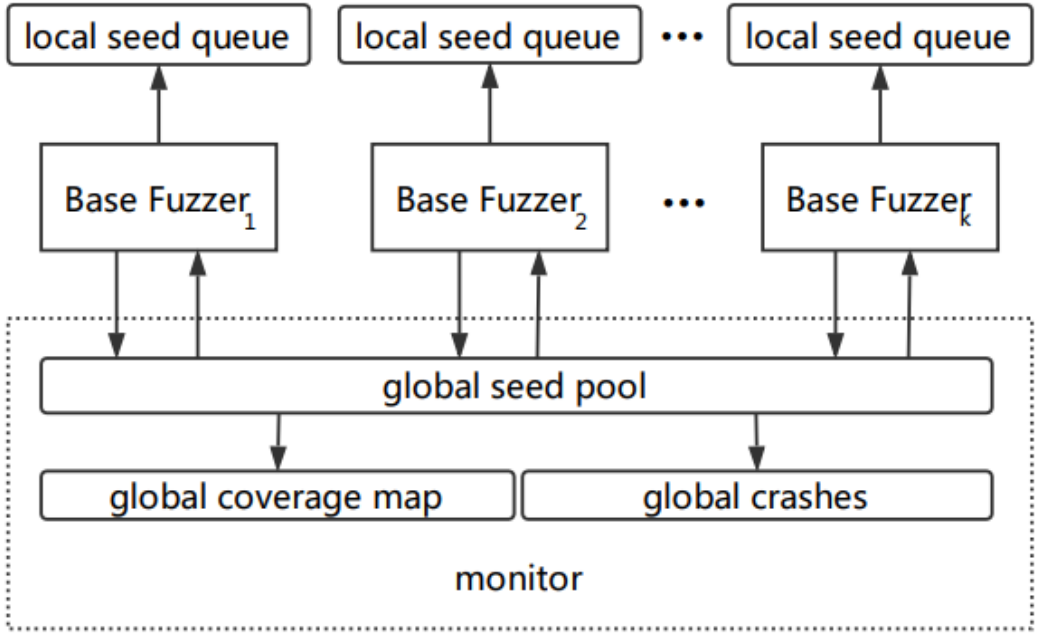
■ Base fuzzer selection

- Seed mutation and selection strategy based heuristic: the diversity of base fuzzers can be determined by the variability of seed mutation strategies and seed selection strategies. For example, AFLFast selects seeds that exercise low-frequency paths and mutates them more times, FairFuzz strives to ensure that the mutant seeds hit the rarest branches.
- Coverage information granularity based heuristic: many base fuzzers determine interesting inputs by tracking different coverage information. Hence, the coverage information is critical, and different kinds of coverage granularity tracked by fuzzers enhances diversity. For example, libFuzzer guides seed mutation by tracking block coverage while AFL tracks edge coverage.

1. If $C_1 \cap C_2 \cap C_3 \ldots \cap C_k \neq \emptyset$, then $|C| > |C_j|$ where $j \in 1, \ldots, k$, it means that the ensemble fuzzer always performs better than that of any base fuzzer.
2. The smaller $|C_i \cap C_j|$ is, the bigger $|C_i \cup C_j|$ is, and the bigger $|C|$ is, where $0 \leq i, j \leq k, i \neq j$.

■ Ensemble Architecture





■ Problem Define

- The crash triggered *seeds are crucial* for fuzzing efficiency.

- The existing seed generation algorithms *rely on human experts*.
  - Fuzzing with Taint analysis
    - ✓ Dynamic taint analysis has a ***low false positive rate*** by discovering bugs in this way.
    - ✓ However, since the analysis needs to add necessary checks and traces the track of data when the application is running, dynamic taint analysis has ***low execution efficiency***.
  - Fuzzing with Symbolic Execution analysis
    - ✓ Symbolic execution can ***accurately locate the bugs*** in an application.
    - ✓ However, it has a ***poor scalability***. When the objective application is large, the final equation becomes too complicated to be solved for.
  - Above all, the legacy methods *did not utilize fuzzing results information* obtained through several experiments.

■ Challenges

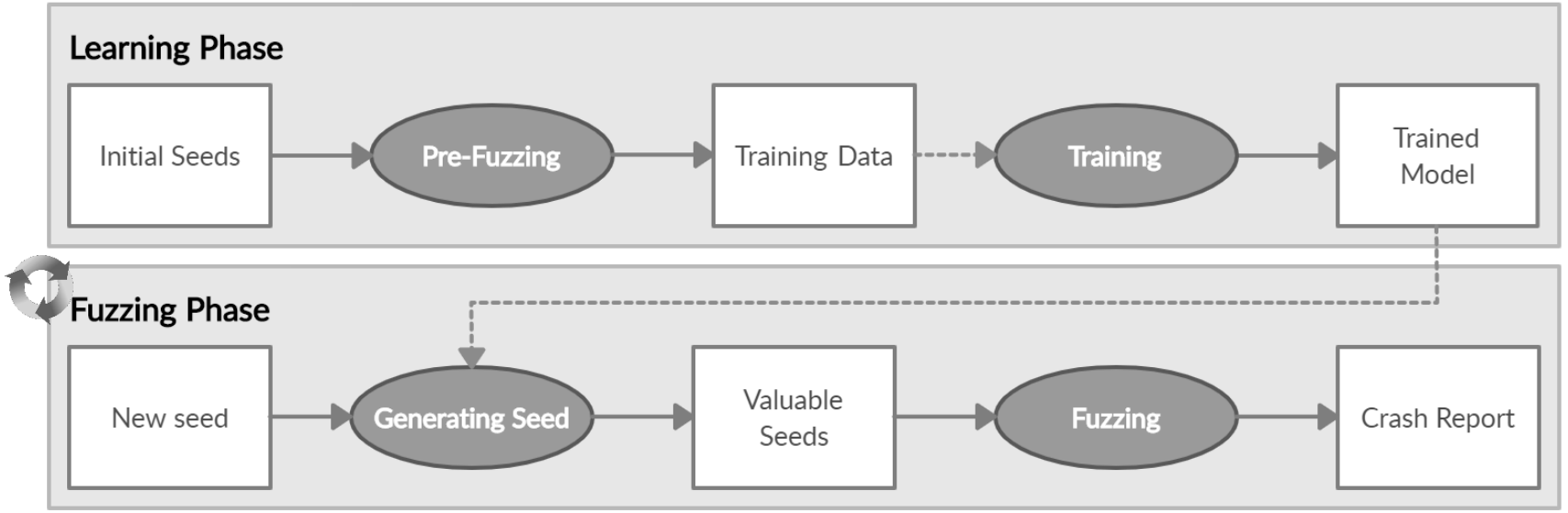
- *How to obtain training data to generate valuable seed files?*
  - AFL을 이용하여 unique path, unique crash를 발생하면 input data와 mutated data를 기록하도록 수정하여 training data set 확보

- *How to automatically generate valuable seeds leveraging state-of-the-art deep learning techniques?*
  - Fuzzing result file을 training data로 하여 sequence-to-sequence model을 학습시키고, trained model을 통해 valuable seed 파일 생성
  - Crash triggered seed file의 core 정보를 기반으로 다양한 valuable seed 파일을 생성하기 위해 RNN based-seq2seq, seq2seq with attention, transformer model 등 다양한 기법 적용

- *How to use different kinds of fuzzing tools in a compatible manner?*
  - Training phase와 fuzzing phase를 분리하여 다양한 base fuzzer에 쉽게 적용할 수 있도록 설계

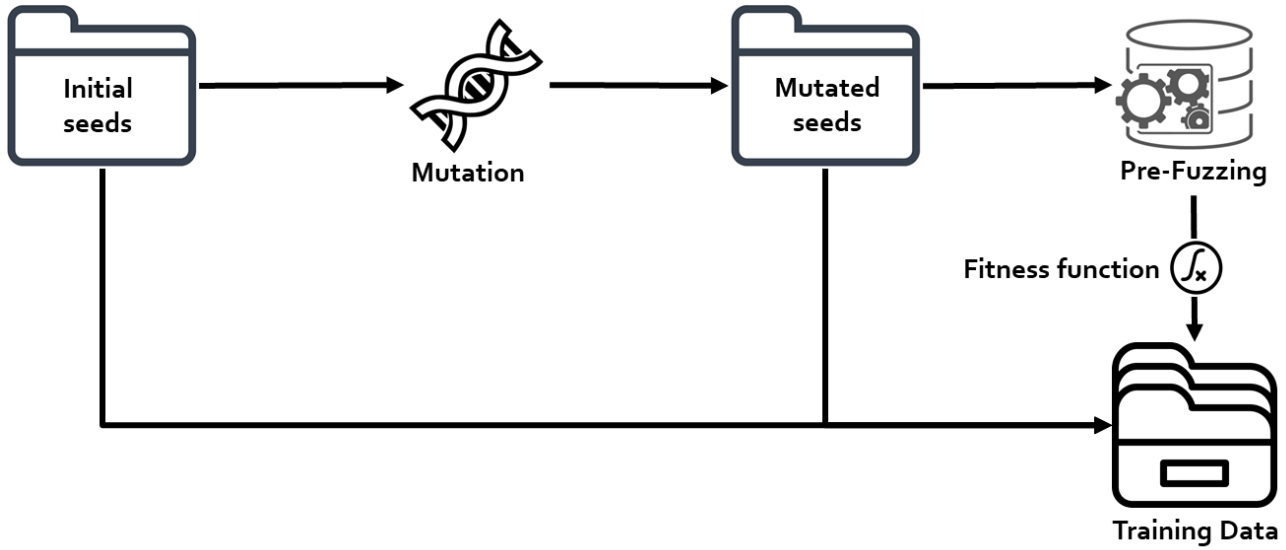
■ Overview of proposed method

- Learning Phase
  - To gather training data using dry fuzzing (triggering crash and new coverage)
  - A seed generative model based on a sequence-to-sequence model
- Fuzzing Phase
  - To obtain valuable seeds through trained transformer model
  - Easy to integrate with base fuzzers since separation between learning and fuzzing phase.



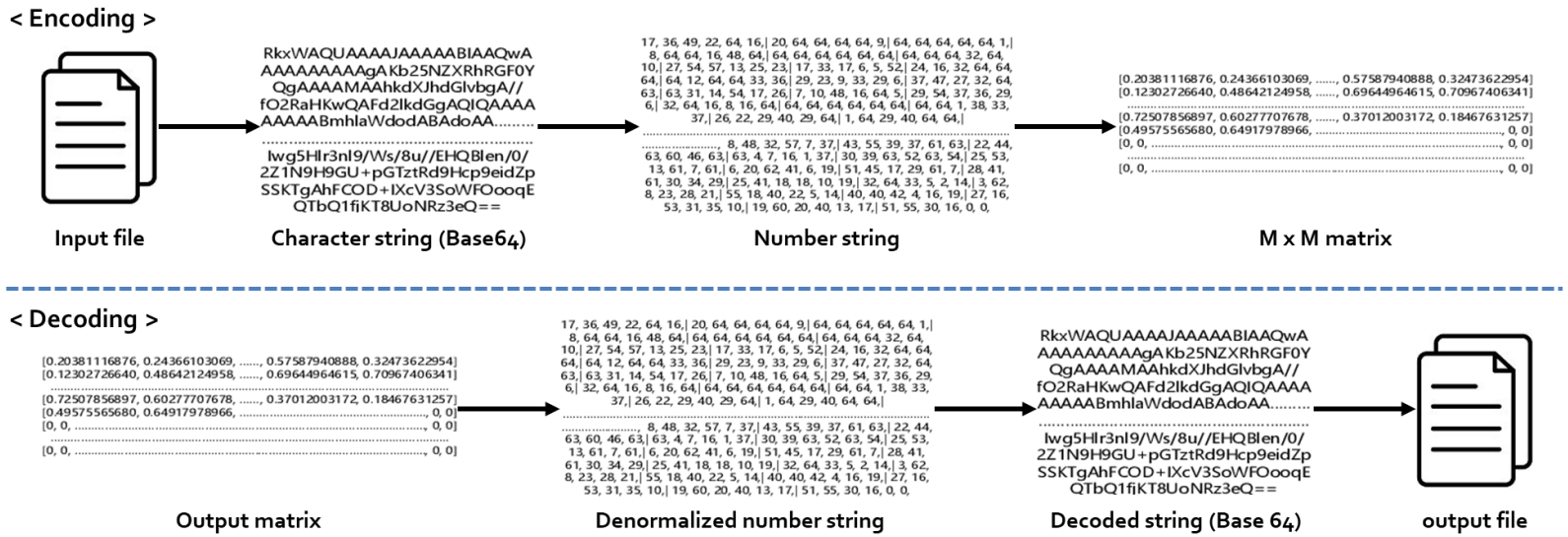
■ Data gathering

- Gathering mp3, bmp, flv as an initial seed files through the internet.
- Obtain training dataset using pre-fuzzing of AFL.
- Fitness function (for training data selection)
  - Unique crash and path
  - Fitness score (above threshold)
  - Touch of crashed source code (by defect clustering)



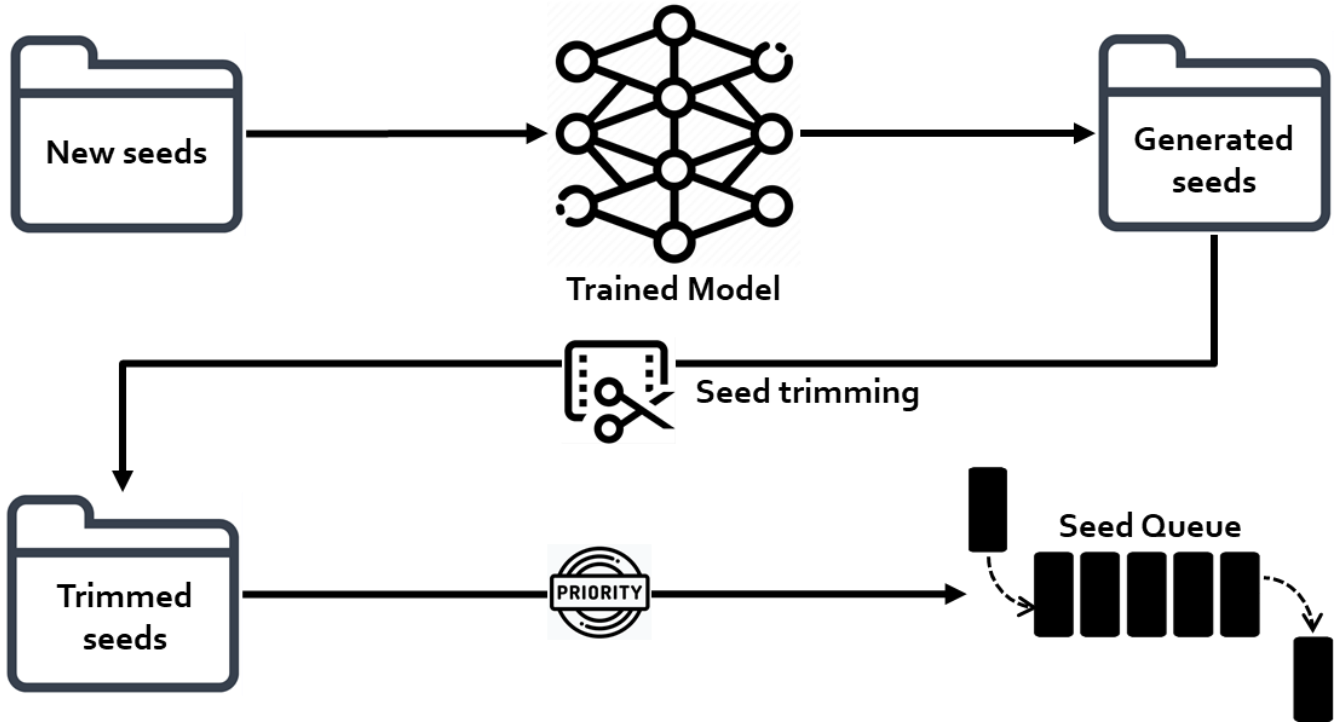
■ Data encoding & decoding

- Encoding
  - Encoding the string (e.g., Base64)
  - Convert number string into matrix (split & padding)
  - Normalize
- Decoding
  - De-normalized
  - Convert matrix into number string
  - Decoding the string (round off)



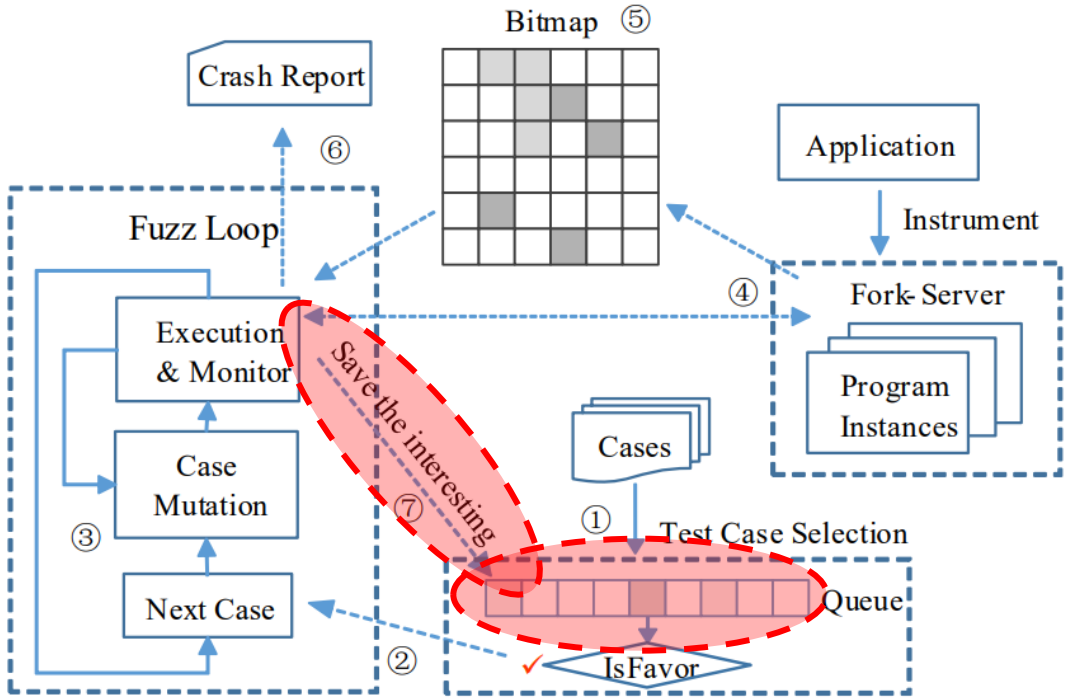
■ Seed Generation

- Generate output data by using the new seed file as input data to the trained model.
- Optimize the generated output data to mitigate duplicate fuzzing problem.
  - Seed trimming using afl-cmin, afl-tmin, or peach-fuzz
  - Seed priority using dimensionality reduction algorithm (e.g., PCA)



■ How to prioritize seed

- Similarity based seed priority (Distance-based similarity e.g., Euclidean distance)



■ Related Work

- Applying Machine Learning Techniques for Seed Generation
  - Faster Fuzzing: Reinitialization with Deep Neural Models. (arXiv, 2017)
    - ✓ Fast fuzzing uses of deep neural models to enhance the effectiveness of random mutation testing.
    - ✓ The method learns features from AFL generated samples, and generates seed files that increase the execution path through confrontation training of the Generative Adversarial Networks (GAN).
  - Skyfire: Data-driven Seed Generation for Fuzzing. (IEEE S&P 2017)
    - ✓ Extraction semantic information automatically
    - ✓ These semantic information and grammar rules are used to seed generation.
  - SmartSeed (arXiv, 2019)
    - ✓ The SmartSeed considerate only unique crash and path as valuable seed.
- Applying Machine Learning Techniques for Mutation Operator Selection

Thank You

- **Lab:** https://mose.kookmin.ac.kr
- **Email:** sh.jeon@kookmin.ac.kr