podft: On Accelerating Dynamic Taint Analysis with Precise Path Optimization

Zhiyou Tian
Xidian University
21151213645@stu.xidian.edu.cn

Cong Sun
Xidian University
suncong@xidian.edu.cn

Dongrui Zeng
Palo Alto Networks
dzeng@paloaltonetworks.com

Gang Tan
Pennsylvania State University
gtan@psu.edu

BAR 2023

- Dynamic taint analysis (DTA)
  - What is it?
  - Useful for security

- Binary-level dynamic data-flow tracking (DFT)
  - Dynamic binary instrumentation (DBI)
  - Virtual machine manager (VMM)
  - Emulator

- DBI-based DTA
  - Focus on explicit flows
  - Hold the tainting states within tagging memory

- Challenge of DTA —— significant performance penalty

High Cost !

- Existing works
  - Lift (*MICOR 2006*)
    - static fast path
  - Libdft (*VEE 2012*)
    - on Pin
    - DBI inline routines
  - TaintRabbit (*ASIA CCS 2020*)
    - on DynamoRIO
    - dynamic fast path
  - SELECTIVETAINT (*USENIX 2021*)
    - static binary rewriting
      bloat the attack surface
    - value-set analysis
      cannot work on library code

- podft advantages
  - more efficient
  - not bloat the attack surface
  - consider library code
  - flexible scalability

Our work —— podft defines and enforces various fast paths

• podft overview

- BPA-based CFG Construction
- VSA-based tainted inst identification
- Tracking policy construction
- PDG-based function abstract(from SDFT)
- Pin-based Tracker

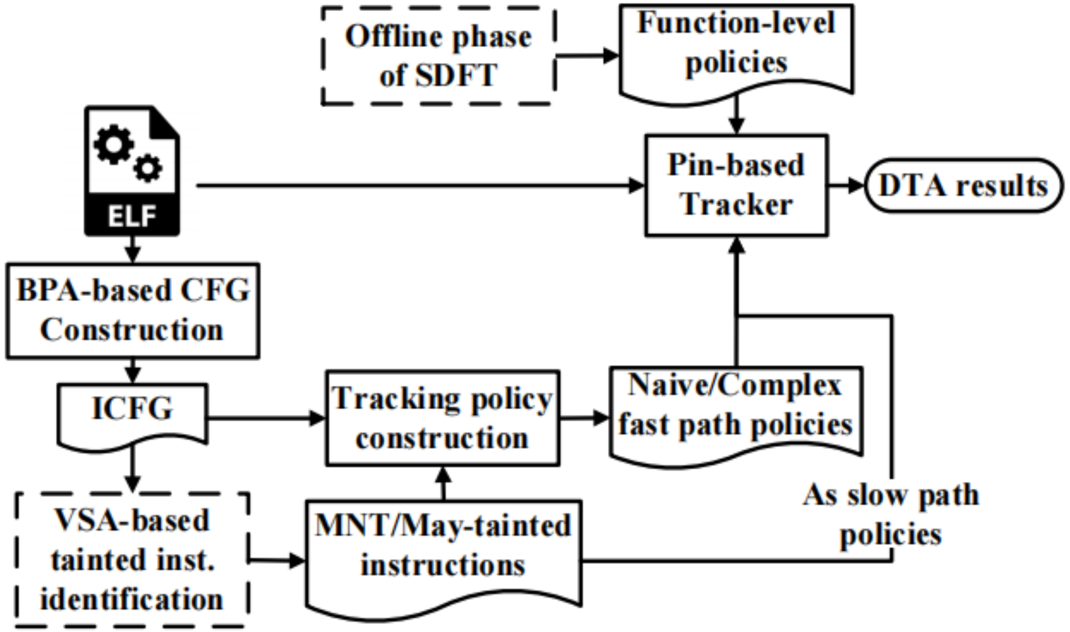
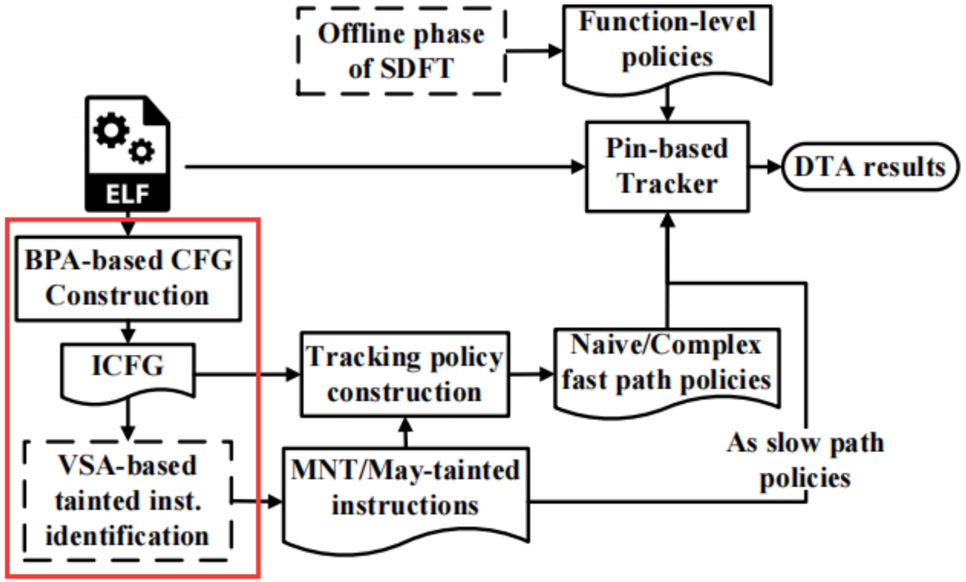


Fig1. Framework of podft (dashed block = usage of existing tools)

Next —— give a toy example to demonstrate

- BPA-based CFG Construction
- VSA-based tainted inst identification




```
void  toy_test (int  fd,  char *buf,  int  size){
    int  read_len  = read(fd,  buf,  size );  // taint  source
    if (read_len  > 0){
        printf ("read data: %s\n", buf);
        for  (int  i  = 0;  i<2; i++){
            buf[i] = i;
            write (fd,  buf[i],  1);  // taint  sink
        }
    }
}
int  main(int  argc,  char *argv[]){
    char  buffer [64]  = {0};
    int  fd  = open(argv [1],  O_RDWR);
    toy_test (fd,  buffer ,  64);
    return  0;
}
```

Fig.2 toy example

```
0x80484fd  (0x80484fd) toy_test
---------------------------------------------
80484fd: push  ebp
80484fe: mov  ebp,  esp
8048500: sub   esp,  0x28
8048503: mov  eax,  dword ptr [ebp + 0x10]
8048506: mov  dword ptr [esp + 8],  eax
804850a: mov  eax,  dword ptr [ebp + 0xc]
804850d: mov  dowrd ptr [esp + 4],  eax
8048511: mov  eax,  dword ptr [ebp + 8]
8048514: mov  dword ptr [esp],  eax
8048517: call     0x8048390 <read@plt>
```

basic block A (bbl-a)

basic block B (bbl-b)

basic block C (bbl-c)

…etc

Example demonstration

- Tracking policy construction
  - naive fast path → main(bbl-c)
  - complex fast path → toy_test+0x44(bbl-b)
  - slow path →toy_test(bbl-a)
  - function fast path →printf etc..

```
0x80484fd  (0x80484fd) toy_test
---------------------------------
80484fd: push  ebp
80484fe: mov  ebp,  esp
8048500: sub   esp,  0x28
8048503: mov  eax,  dword ptr [ebp + 0x10]
8048506: mov  dword ptr [esp + 8],  eax
804850a: mov  eax,  dword ptr [ebp + 0xc]
804850d: mov  dowrd ptr [esp + 4],  eax
8048511: mov  eax,  dword ptr [ebp + 8]
8048514: mov  dword ptr [esp],  eax
8048517: call    0x8048390 <read@plt>
```

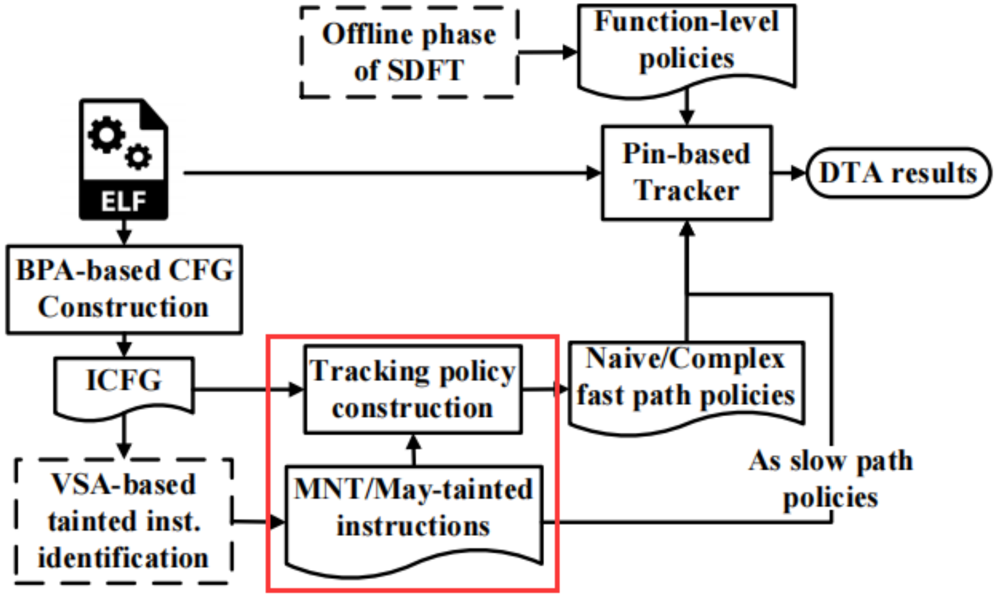
basic block A (bbl-a)

```
0x8048541  (0x80484fd) toy_test+0x44
---------------------------------
8048541: mov  edx,  dword ptr [ebp - 0x10]
8048544: mov  eax, dword ptr [ebp + 0xc]
8048547: add   edx,  eax
8048549: mov  eax,  dword ptr [ebp – 0x10]
804854c: mov   byte ptr [edx],  al
804854e: mov  edx,  dword ptr [ebp – 0x10]
8048551: mov  eax,  dword ptr [ebp + 0xc]
8048554: add    eax,  edx
8048556: movzx eax,  byte ptr [eax]
8048559: movsx eax,  al
804855c: mov  dword ptr [esp + 8],  1
8048564: mov  dword ptr [esp + 4],  eax
8048568: mov  eax,  dword ptr [ebp + 8]
804856b: mov  dword ptr [esp],  eax
804856e: call    0x80483f0 <write@plt>
```

basic block B (bbl-b)

```
0x804857f  (0x804857f) main
---------------------------------
804857f:  push  ebp
8048580:  mov  ebp,  esp
8048582:  push  edi
8048583:  push  esi
8048584:  push  ebx,
8048585:  and   esp,  0xfffffff0
8048588:  sub    esp,  0x60
804858b:  mov  eax,  dword ptr [ebp + 0xc]
804858e:  mov  dword ptr [esp + 0xc],  eax
8048592:  mov  eax,  dword ptr gs : [0x14]
8048598:  mov  dword ptr [esp + 0x5c],  eax
804859c:  xor    eax,  eax
804859e:  lea     ebx,  [esp + 0x1c]
80485a2:  mov  eax,  0
80485a7:  mov  edx,  0x10
80485ac:  mov  edi,  ebx
80485ae:  mov  ecx,  edx
80485b0:  rep stosd dword ptr es : [edi],  eax
```

basic block C (bbl-c)



Why?

Example demonstration

- Tracking policy construction
  - naive fast path → main(bbl-c)
    - Not contain potentially tainted instructions

because

```
0x804857f  (0x804857f) main
--------------------------------------------------
804857f:  push  ebp
8048580:  mov  ebp,  esp
8048582:  push  edi
8048583:  push  esi
8048584:  push  ebx,
8048585:  and   esp,  0xfffffff0
8048588:  sub    esp,  0x60
804858b:  mov  eax,  dword ptr [ebp + 0xc]
804858e:  mov  dword ptr [esp + 0xc],  eax
8048592:  mov  eax,  dword ptr gs : [0x14]
8048598:  mov  dword ptr [esp + 0x5c],  eax
804859c:  xor    eax,  eax
804859e:  lea     ebx,  [esp + 0x1c]
80485a2:  mov  eax,  0
80485a7:  mov  edx,  0x10
80485ac:  mov  edi,  ebx
80485ae:  mov  ecx,  edx
80485b0:  rep stosd dword ptr es : [edi],  eax
```

basic block C (bbl-c)

- Tracking policy construction

  - complex fast path → toy_test+0x44(bbl-b)
    - Contain potentially tainted instructions
    - Hot BBL (be executed multiple times)
    - TaintedMem(bbl) ∩ MergedDep(bbl) =∅.

because

```
int read_len = read(fd, buf, size); // taint source
if (read_len > 0){
    printf ("read data: %s\n", buf);
    for (int i = 0; i<2; i++){
        buf[i] = i;
        write(fd, buf[i], 1); // taint sink
    }
```

```
0x8048541 (0x80484fd) toy_test+0x44
------------------------------------------
8048541: mov edx, dword ptr [ebp - 0x10]
8048544: mov eax, dword ptr [ebp + 0xc]
8048547: add edx, eax
8048549: mov eax, dword ptr [ebp – 0x10]
804854c: mov byte ptr [edx], al
804854e: mov edx, dword ptr [ebp – 0x10]
8048551: mov eax, dword ptr [ebp + 0xc]
8048554: add eax, edx
8048556: movzx eax, byte ptr [eax]
8048559: movsx eax, al
804855c: mov dword ptr [esp + 8], 1
8048564: mov dword ptr [esp + 4], eax
8048568: mov eax, dword ptr [ebp + 8]
804856b: mov dword ptr [esp], eax
804856e: call 0x80483f0 <write@plt>
```

basic block B (bbl-b)

The data delivered to the sink are irrelevant to the tainted data from the source.

Example demonstration

- Tracking policy construction

  - Slow path →toy_test(bbl-a)
    - Contain potentially tainted instructions
    - Not hot *or* TaintedMem(bbl) ∩ MergedDep(bbl) ≠ ∅

because

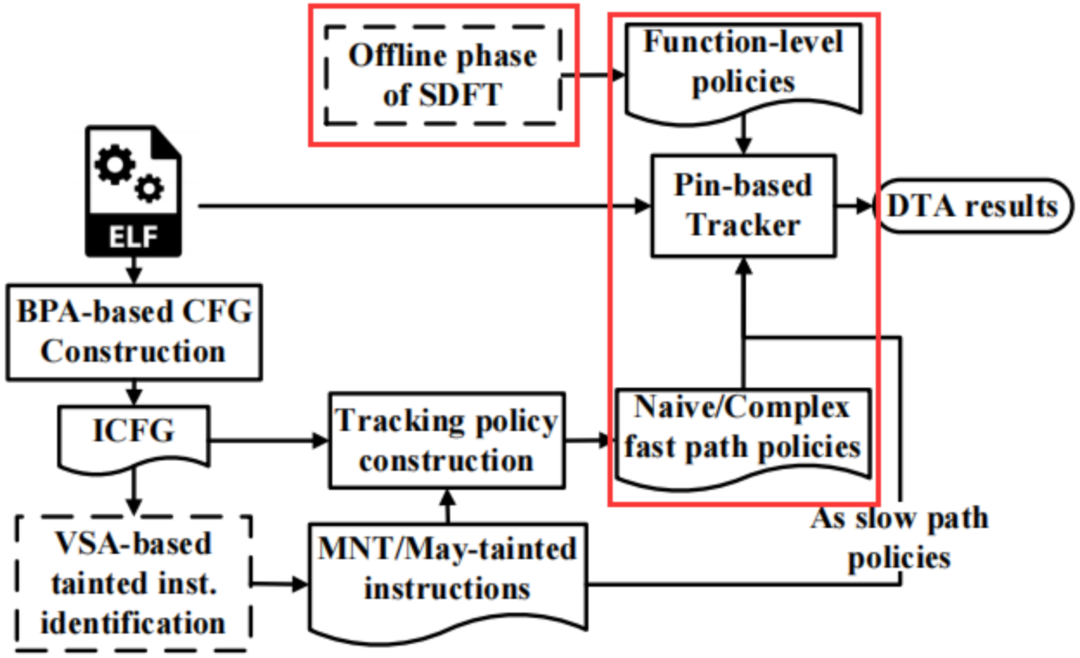
```
0x80484fd  (0x80484fd) toy_test
----------------------------------------------
80484fd: push  ebp
80484fe: mov  ebp,  esp
8048500: sub   esp,  0x28
8048503: mov  eax,  dword ptr [ebp + 0x10]
8048506: mov  dword ptr [esp + 8],  eax
804850a: mov  eax,  dword ptr [ebp + 0xc]
804850d: mov  dowrd ptr [esp + 4],  eax
8048511: mov  eax,  dword ptr [ebp + 8]
8048514: mov  dword ptr [esp],  eax
8048517: call    0x8048390 <read@plt>
```

basic block A (bbl-a)

- PDG-based function abstract
  - Function fast path →printf etc..
- Pin-based Tracker

  Input:
  - Function-level policies
  - Naive/Complex fast path policies
  - Slow path policies

  output:
  - DTA results



Next ——podft's efficiency and effectiveness.

- Experimental Settings
  - Desktop with a 2.8GHz×4 Intel Core(TM) i7-7700HQ CPU, 8GB RAM, and Linux 3.16.0 kernel (Ubuntu 14.04 32-bit).
  - The DBI framework is Pin v2.14, and libdft.

- Benchmark Programs

TABLE II. BENCHMARK PROGRAMS

Dataset ID	Description
S1	10 SPEC CPU 2k6 benchmarks
S2	3 server programs, i.e., Nginx (1.22.0), Apache httpd (2.4.7), and MySQL (5.5.62)
S3	9 CVE programs, as presented in Table V

- Efficiency of podft
  - Compare podft's efficiency with Taint Rabbit , Dytan, Triton , and Taintgrind.

podft achieves slowdowns of 1.6x to 27.9x with an average slowdown of 10.6x.
podft is more efficient than the other DTA tools.

TABLE III. PERFORMANCE IMPROVEMENT OF PODFT COMPARED WITH OTHER DTA APPROACHES ON DATASET S1 AND S2. SLOWDOWN $SD_{TOOL}=T_{TOOL}/T_{ORIG}$, S.T. TOOL=PODFT, TAINTRABBIT-ID, TAINTRABBIT-BV, TAINTGRIND, DYTAN, TRITON

Benchmark	T_{orig} (s)	podft T_{podft}(s)	podft SD_{podft}	TaintRabbit-ID T_{ID}(s)	TaintRabbit-ID SD_{ID}	TaintRabbit-BV T_{BV}(s)	TaintRabbit-BV SD_{BV}	Taintgrind $T_{Taintgrind}$(s)	Taintgrind $SD_{Taintgrind}$	Dytan T_{Dytan}(s)	Dytan SD_{Dytan}	Triton T_{Triton}(s)	Triton SD_{Triton}
400.perlbench	4.3	120.1	27.9	126.6	29.4	204.9	47.7	346.7	80.6	3,208.6	746.2	Timeout	N/A
401.bzip2	5.4	46.4	8.6	85.7	15.9	91.3	16.9	1,634.9	302.8	20,812.6	3,854.2	Timeout	N/A
429.mcf	2.5	5.2	2.1	34.2	13.7	Timeout	N/A	193.6	77.4	2,845.4	1,138.2	Timeout	N/A
445.gobmk	15.3	128.2	8.4	219.2	14.3	204.7	13.4	6,039.4	394.7	45,147.5	2,950.8	Timeout	N/A
456.hmmer	2.2	55.6	25.3	74.8	34.0	78.5	35.7	998.6	453.9	11,531.6	5,241.6	Timeout	N/A
458.sjeng	4.6	28.3	6.2	34.9	7.6	39.2	8.5	1,355.8	294.7	Failed	N/A	Timeout	N/A
462.libquantum	0.5	3.6	7.2	6.8	13.6	8.2	16.4	22.9	45.8	333.2	666.4	Timeout	N/A
464.h264ref	12.6	191.8	15.2	762.3	60.5	2,588.7	205.5	5,057.7	401.4	81,798.6	6,492.0	Timeout	N/A
471.omnetpp	0.4	9.1	22.8	54.6	136.5	54.1	135.3	259.6	649.0	1,401.5	3,503.8	Timeout	N/A
473.astar	9.1	14.5	1.6	129.6	14.2	129.2	14.2	1,467.4	161.3	19,525.1	2,145.6	Timeout	N/A
mysqlslap_myisam	0.6	4.5	7.5	27.5	45.8	404.2	673.7	77.5	129.2	951.9	1,586.5	Timeout	N/A
mysqlslap_innodb	0.7	4.9	7.0	28.8	41.1	392.3	560.4	73.8	105.4	999.1	1,427.3	Timeout	N/A
httpd_req_10k	0.6	5.3	8.8	11.7	19.5	15.9	26.5	26.7	44.5	227.6	379.3	40,509.2	67,515.3
httpd_req_100k	4.9	27.4	5.6	39.8	8.1	74.4	15.2	219.3	44.8	2,386.9	487.1	Timeout	N/A
Nginx_req_10k	0.5	5.5	11.0	9.8	19.6	11.7	23.4	23.9	47.8	201.1	402.2	33,671.6	67,343.2
Nginx_req_100k	4.2	20.1	4.8	24.7	5.9	32.8	7.8	196.4	46.8	2,062.3	491.0	Timeout	N/A
Avg. slowdown		–	10.6	–	30.0	–	120.0	–	205.0	–	2,100.8	–	67,429.3

- **Efficiency of podft**
  - Compare podft's efficiency with SELECTIVETAINT.

TABLE IV. PERFORMANCE COMPARISON OF PODFT WITH SELECTIVETAINT

Benchmark	podft SD_{podft}	STATICTAINTALL T_{All}(s)	STATICTAINTALL SD_{All}	SELECTIVETAINT T_{Select}(s)	SELECTIVETAINT SD_{Select}
400.perlbench	27.9	26.8	6.2	24.2	5.6
401.bzip2	8.6	263.1	48.7	174.1	32.2
429.mcf	2.1	39.1	15.6	3.8	1.5
445.gobmk	8.4	543.4	35.5	538.5	35.2
456.hmmer	25.3	161.4	73.4	129.2	58.7
458.sjeng	6.2	126.8	27.6	94.8	20.6
462.libquantum	7.2	6.3	12.6	4.5	9.0
464.h264ref	15.2	1,255.9	99.7	801.2	63.6
471.omnetpp	22.8	16.4	41.0	14.9	37.3
473.astar	1.6	277.4	30.5	61.7	6.8
Avg. slowdown	12.5	–	39.1	–	27.1

podft achieves slowdowns of 1.6x to 27.9x with an average slowdown of 12.5x. and is generally more efficient than SELECTIVETAINT.

- Effectiveness of podft's Dynamic Taint Analysis

Real exploits detection by podft

Develop Pintool over podft to track vulnerability of CVEs

TABLE V. EFFECT OF PODFT ON TRACKING VULNERABILITIES OF CVES (RCE=REMOTE CODE EXECUTION, S-OF=STACK OVERFLOW, H-OF=HEAP OVERFLOW)

ID	Program	Type	T_{podft}(s)
CVE-2021-41253	Zydis v3.2.0	H-OF	1.4
CVE-2019-8354	SoX v14.4.2	H-OF	2.8
CVE-2018-19655	dcraw v9.28	S-OF	1.7
CVE-2018-11575	ngiflib v0.4	S-OF	1.4
CVE-2018-6612	jhead v3.00	H-OF	1.2
CVE-2017-1000437	Gravity v0.3.5	RCE	9.8
CVE-2017-14411	MP3Gain v1.5.2	S-OF	2.2
CVE-2013-2028	Nginx v1.4.0	S-OF	1.1

- Work in progress

more scalable and more flexible to be used in traditional DTA

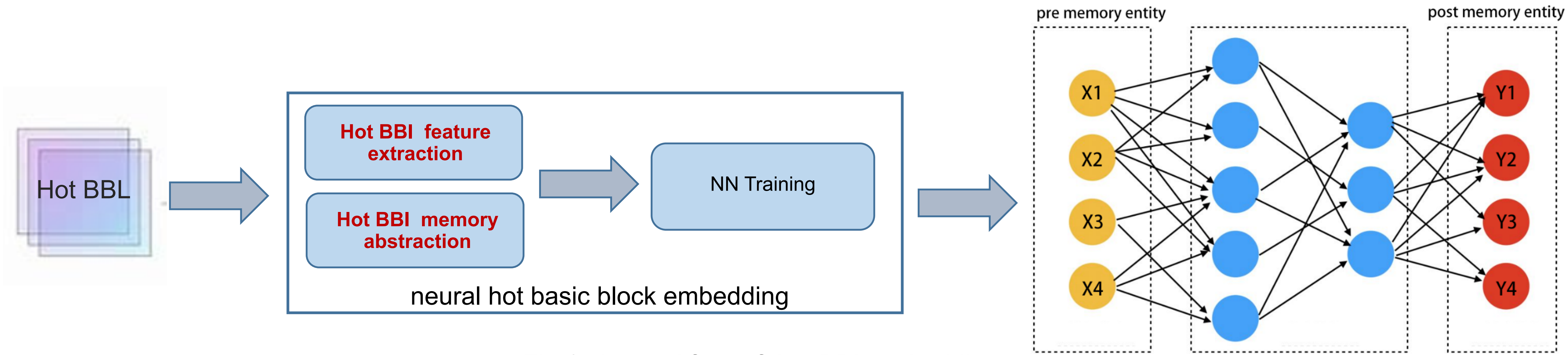


Fig.1. the workflow of hot BBL embedding

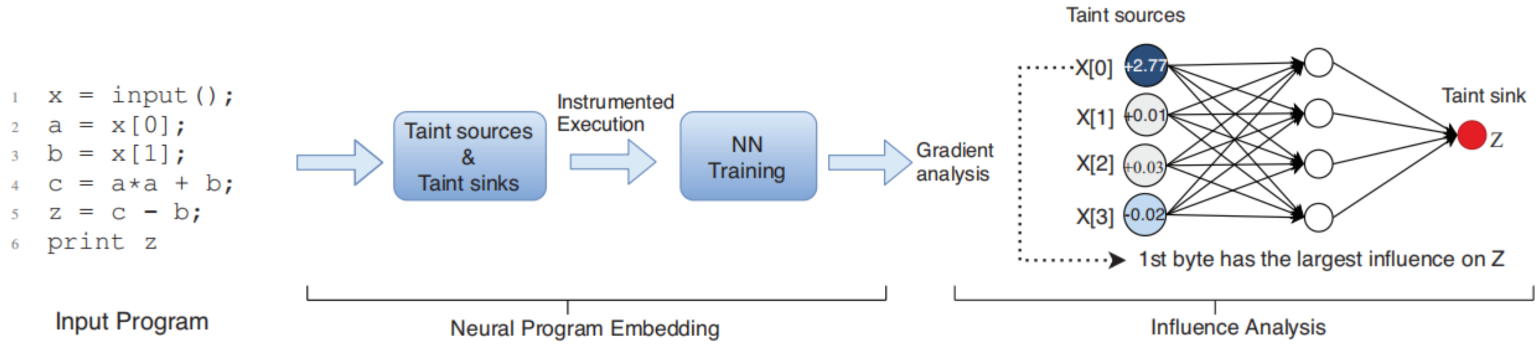


Fig.2. the workflow of NeuTaint(*SP2020*)

THANKS

Thanks for listening