V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구

Eunchong Lee

(Email: gr4ce@korea.ac.kr)

CIST, Korea University

Homepage : https://sites.google.com/site/securesiplab/

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Background - 자바스크립트의 메모리 분석 방법

- Binary Instrument Tools (LLVM, Vargrind, PIN ...)
  - ▸ 대상 프로그램에 계측 코드를 삽입하여 실행중인 메모리의 흐름을 분석.

```
X START 0:15548 at 0
X THREAD_CREATE 0:1
X 1 MALLOC 005188030 200 mystruct 0
S ffeffd0c8 8 1 S main
S ffeffd11c 4 1 S main LV A
L ffeffd11c 4 1 S main LV A
S ffeffd0f4 4 1 S main LS Arr[5]
S 000601030 4 1 G main GS myG.Ab
S 000601034 4 1 G main GS myG.Ba
L ffeffd110 8 1 S main LV ptr
L ffeffd0f4 4 1 S main LS Arr[5]
S ffeffcff8 8 1 S main
```

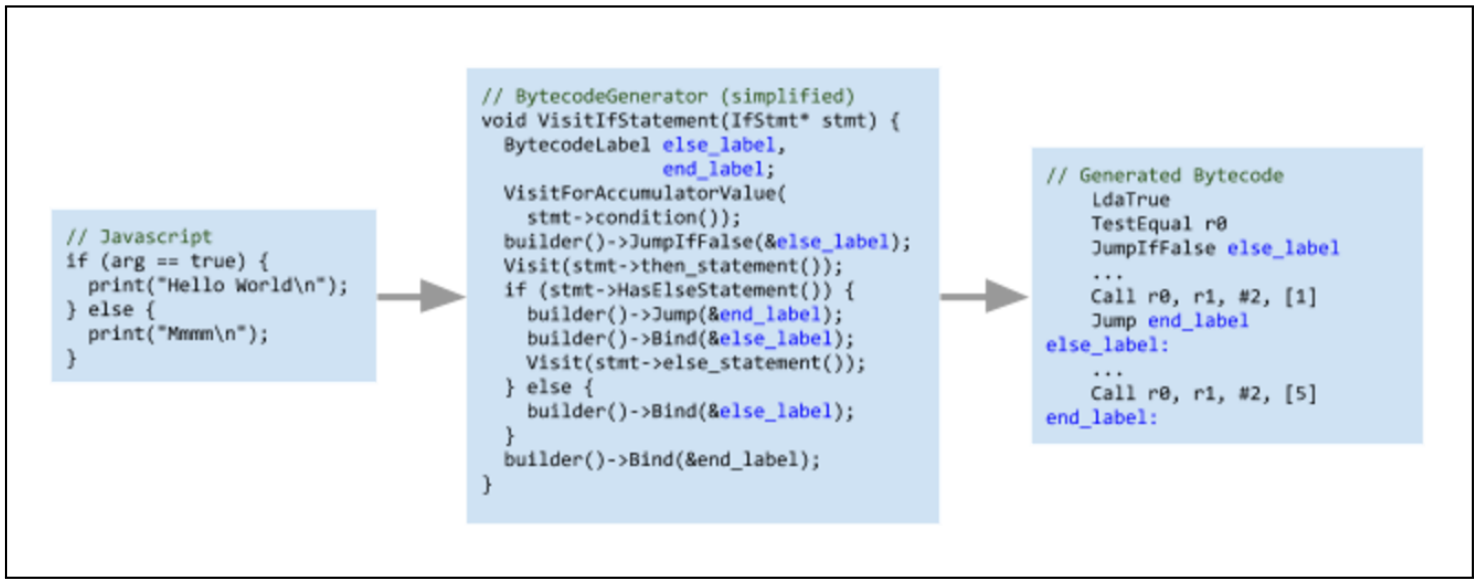
(Source from `https://cug.org/proceedings/cug2014_proceedings/includes/files/pap114.pdf`)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Limitation - in Javascript Engine

- Binary Instrument Tools (LLVM[1], Vargrind[3], PIN ...)
  - ► 자바스크립트 엔진의 실행 과정.
    - 자바스크립트는 Adaptive JIT 방식으로 중간언어를 사용한 최적화 과정을 거치고, 추적된 바이트코드의 주소로 자바스크립트 라인을 알기 어려움.



(Source from `https://github.com/v8/v8/wiki/Interpreter`)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Research Objective - 자바스크립트 메모리 추적 시스템

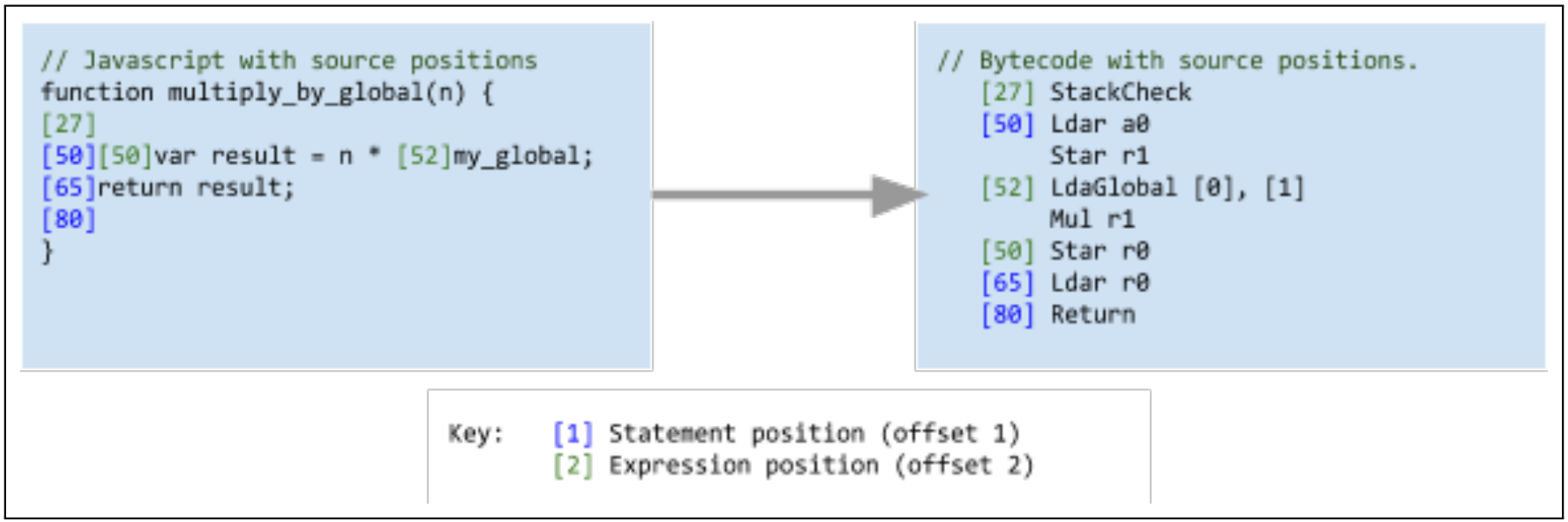
- 연구 목적
  - 자바스크립트 환경에 적합한 메모리 추적 분석 시스템
    1. 바이트코드(중간언어) 형태의 실행 정보를 자바스크립트 라인 실행 정보로 변환.
    2. 자바스크립트 라인에서의 메모리 할당,해제,읽기,쓰기 정보 추적
    3. 실제 환경에 적용 가능한 실행 성능의 확보.

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable - 자바스크립트 코드 위치 탐색
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Technical Background - 자바스크립트 코드 위치 탐색

- 기술 배경 - 자바스크립트 코드 위치 저장 테이블[5], SourcePositionTable
  - ▶ 중간언어(IR) 변환 단계에서 자바스크립트 라인의 위치를 테이블에 저장함.



(Source from https://github.com/v8/v8/wiki/Interpreter)

Technical Background - 자바스크립트 코드 위치 탐색

- JSMTrace - 바이트코드 변환 시점에서 SourcePositionTable 전달.
  - 중간언어(IR) 변환 단계에서 SourcePositionTable 정보를 JSMTrace로 전달.

```
// GRMTrace
for (SourcePositionTableIterator encoded(*table); !encoded.done();
     encoded.Advance()) {

  if(is_shared)
  {
    int source_pos = encoded.source_position();
    int line = script->GetLineNumber(source_pos) + 1;
    int column_num = script->GetColumnNumber(Handle<Script>(script),source_pos) + 1;
    base::OS::FPrint((isolate)->logger()->CodeInfoGetFP(), "02:%llx:%d:%d\n",
      code->instruction_start()+encoded.code_offset(),line,column_num);
  }

}
// GRMTrace
```

(Source from `https://github.com/v8/v8/wiki/Interpreter`)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory - 낮은 오버헤드의 추적 방법
- Experiment
- Conclusion
- Reference
- Q & A

Technical Background - 낮은 오버헤드의 추적 방법

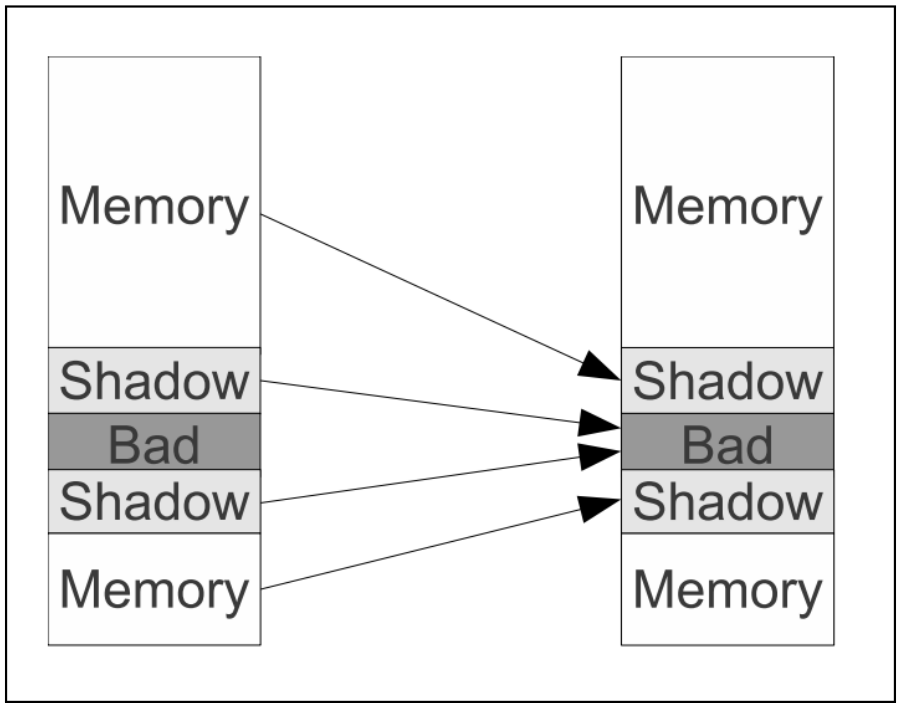
- 기술 배경 - LLVM's ASan의 메모리 추적 기법
  - ASan은 다른 도구들에 비해 메모리 에러 검출에서 빠른 성능을 가짐.

	ASan	Valgrind	Dr. Memory	Mudflap
technology	CTI	DBI	DBI	CTI
Slowdown	2x	20x	10x	2x-40x
Heap OOB	yes	yes	yes	yes
Stack OOB	yes	no	no	some
Global OOB	yes	no	no	?
UAF	yes	yes	yes	yes
UAR	yes	no	no	no
UMR	no	yes	yes	?
Leaks	yes	yes	yes	?

(Source from `https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools`)

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - LLVM's ASan의 핵심 알고리즘
  - 메모리의 상태를 나타내는 Shadow Memory 영역 할당함.



(Source from `http://research.google.com/pubs/pub37752.html`)

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - LLVM's ASan의 핵심 알고리즘
  - 메모리의 영역의 변화가 발생하면, Shadow Memory에 상태 값들을 표시함.

값	상태	값	상태	값	상태
00	Addressable	07	Partially able	f3	Stack right redzone
01	Partially able	08	Partially able	f4	Stack partial redzone
02	Partially able	fa	Heap left redzone	f5	Stack after return
03	Partially able	fb	Heap righ redzone	f8	Stack use after scope
04	Partially able	fd	Freed Heap region	f9	Global redzone
05	Partially able	f1	Stack left redzone	f6	Global init order
06	Partially able	f2	Stack mid redzone	f7	Poisoned by user

(Source from `https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterFree`)

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - LLVM's ASan의 핵심 알고리즘
  - 메모리 접근시 MEM_TO_SHADOW로 상태 확인, O(1) 성능.

```
346  static inline bool AddressIsPoisoned(uptr a) {
347    PROFILE_ASAN_MAPPING();
348    const uptr kAccessSize = 1;
349
350    u8 *shadow_address = (u8*)MEM_TO_SHADOW(a);
351    s8 shadow_value = *shadow_address;
352    if (shadow_value) {
353      u8 last_accessed_byte = (a & (SHADOW_GRANULARITY - 1))
354                              + kAccessSize - 1;
355      return (last_accessed_byte >= shadow_value);
356    }
357    return false;
358  }
```

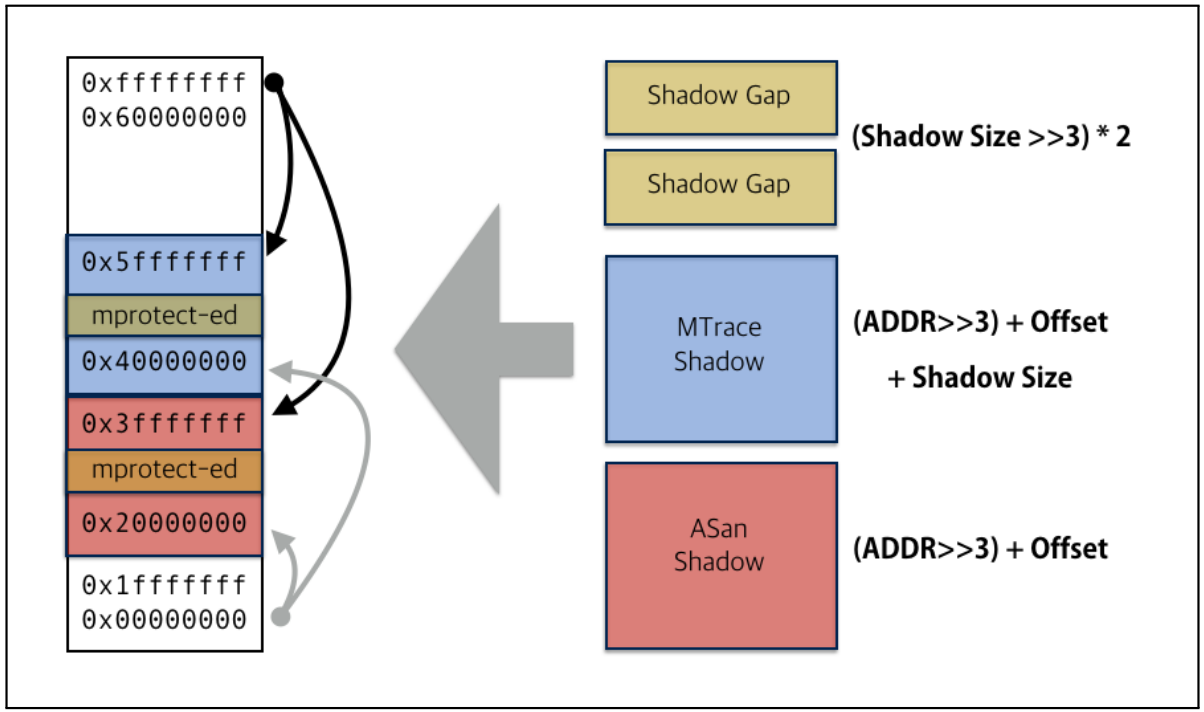
(Source from https://github.com/llvm-mirror/compiler-rt/blob/master/lib/asan/asan_mapping.h#L319)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory - JSMTrace의 오버헤드 컨트롤
- Experiment
- Conclusion
- Reference
- Q & A

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - JSMTrace의 추적 알고리즘
  - ▸ 메모리 사용 추적을 위한 MTrace Shadow Memory 할당.



(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - JSMTrace의 추적 알고리즘
  - 메모리 할당이 발생하면, PoisonMIdShadow를 통해 할당 영역에 메모리 할당 고유번호 mallocId를 표시함, O(1) 성능.

```
// GRMTrace
for(int i=0;i<stack->size;i++)
{
  if((uptr)stack->trace_buffer && ((uptr)0x7f0000000000 & (uptr)stack->trace_buffer[i]) == (uptr)0x7f0000000000 )
  {
    int midValue = *(int*)MEM_TO_SHADOW_MID_DWORD((uptr)stack->trace_buffer[i]);
    if(midValue == -1)
    {
      mallocId+=1;
      PoisonMIdShadow(addr,size,mallocId);

      if(MTRACE_LOG)
      {
        fprintf(stderr,"[m] %llx,%d,%d,%d,%llx,%d\n",addr,size,mallocId,stack->size,stack->trace_buffer[i],midValue);
        stack->Print();
      }
      break;
    }
  }
}
// GRMTrace
```

(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Technical Background - 낮은 오버헤드의 추적 방법

- 기술 배경 - JSMTrace의 추적 알고리즘
  - 메모리 접근이 발생하면, 해당하는 영역의 할당 고유 번호 mallocId를 가져와 출력함, O(1) 성능.

```
extern "C"
NOINLINE INTERFACE_ATTRIBUTE
void __asan_store_printf(uptr addr, u32 size) {
  unsigned int *mallocIdPtr = (unsigned int*)MEM_TO_SHADOW_MID_DWORD(addr);
  if(*mallocIdPtr > 0 && *mallocIdPtr < __asan_mallocIdCur) {
    Printf("[u] %llx,0,%d,store\n",addr,*mallocIdPtr);
  }
}

extern "C"
NOINLINE INTERFACE_ATTRIBUTE
void __asan_load_printf(uptr addr, u32 size) {
  unsigned int *mallocIdPtr = (unsigned int*)MEM_TO_SHADOW_MID_DWORD(addr);
  if(*mallocIdPtr > 0 && *mallocIdPtr < __asan_mallocIdCur) {
    Printf("[u] %llx,0,%d,load\n",addr,*mallocIdPtr);
  }
}
```

(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Experiment - 실험 환경 구축

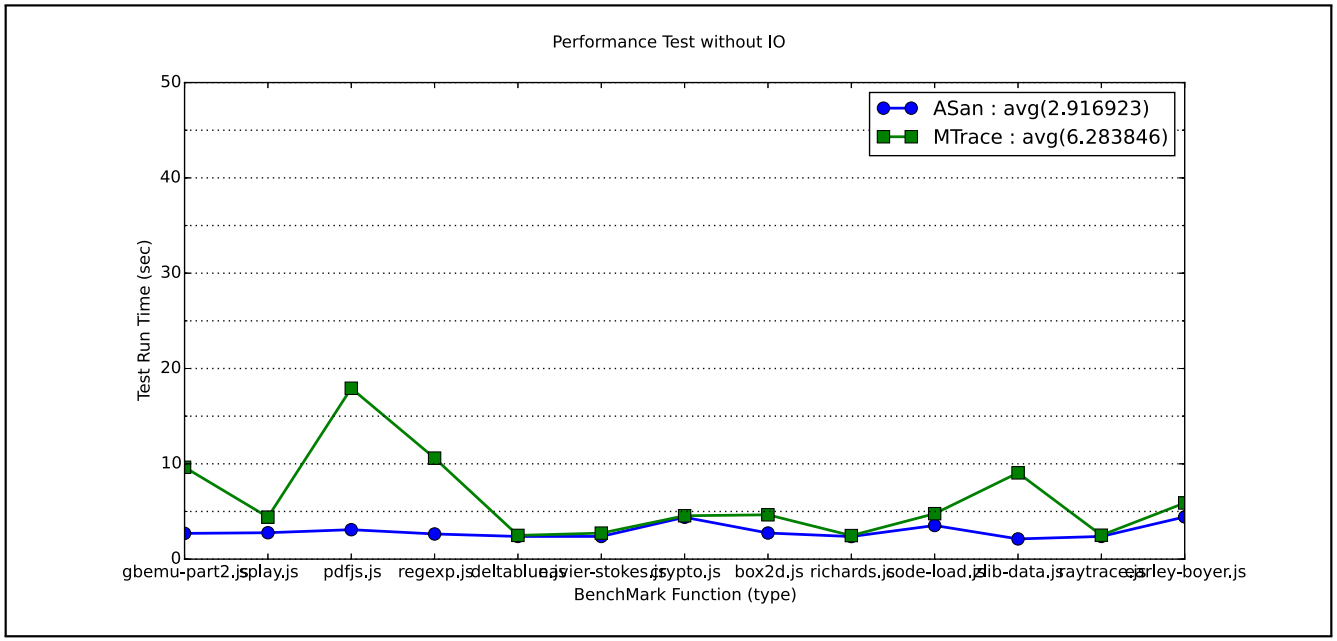
- 실험 환경 - 벤치마크 도구를 활용한 성능 실험과, 메모리 추적 분석 실험
  - 파일쓰기 사용, 파일쓰기 제거 버전의 성능 실험과 자바스크립트 라인별, 할당 그룹별 메모리 추적 분석 실험.

운영체제	Ubuntu 14.04.5 LTS, Intel(R) Xeon(R) CPU @ 2.50GHz
CPU	Intel(R) Xeon(R) CPU @ 2.50GHz
메모리	15 GB (15400548 kB)
벤치마크 도구	Chrome JS Benchmark Util 'Octane'
오차 범위	각 실행 프로그램별 5회 실행 평균값 사용

(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Experiment - 성능 실험 1

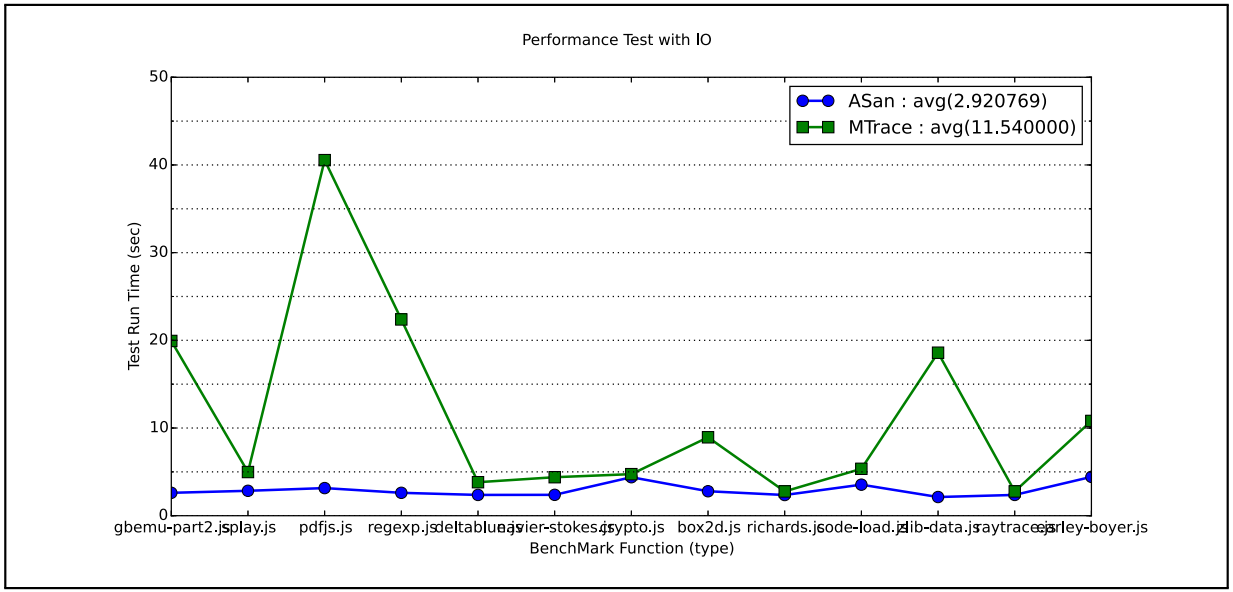
- 성능 실험 결과 1 - 파일 쓰기 제거 버전
  - 파일 쓰기 제거 버전은, 평균 1.2x의 오버헤드를 보여줌. 메모리의 사용이 많은 pdfjs.js에서 비교적 많은 오버헤드가 발생함.



(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Experiment - 성능 실험 2

- 성능 실험 결과 2 - 파일 쓰기 사용 버전
  - 파일 쓰기 사용 버전은, 평균 3.1x의 오버헤드를 보여줌. 메모리의 사용이 많은 pdfjs.js에서 역시 비교적 많은 오버헤드가 발생하였고, 전체적으로 파일 쓰기 제거 버전보다 성능 오버헤드가 발생함.



(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Experiment - 분석 실험 1

- 분석 실험 결과 1 - 자바스크립트 라인별 메모리 분석
  - 메모리의 사용 추적 데이터를 자바스크립트 라인별 주석으로 시각화한 분석 결과.

```
1  print('Hello MTrace')
2
3  function str2ab(str) {
4    var buf = new ArrayBuffer(str.length*2);
5    // [MTrace] -> ['alloc', '602000035a70', '16', '90', '17', '7fce8a66d7f6', '-1', '', '']
6    var bufView = new Uint16Array(buf);
7    for (var i=0, strLen=str.length; i < strLen; i++) {
8      bufView[i] = str.charCodeAt(i);
9      // [MTrace] -> ['store(1)', '602000035a70', '0', '90', '', '', '', '', '']
10     // [MTrace] -> ['alloc', '62d000762400', '33264', '101', '26', '7fce8a66d94a', '-1', '', '']
11   }
12   return buf;
13 }
14
15 b = str2ab('deadbeef')
16
17 print(b[0])
18 var bufView = new Uint16Array(b);
19 print(bufView[0])
20 // [MTrace] -> ['load(1)', '602000035a70', '0', '90', '', '', '', '', '']
21 print(bufView[5])
22 // [MTrace] -> ['load(1)', '602000035a7a', '0', '90', '', '', '', '', '']
23 print(bufView[1])
24 // [MTrace] -> ['load(1)', '602000035a72', '0', '90', '', '', '', '', '']
25 print(bufView[2])
26 // [MTrace] -> ['load(1)', '602000035a74', '0', '90', '', '', '', '', '']
27 bufView[4] = 'cafebabe'.charCodeAt(3);
28 // [MTrace] -> ['store(1)', '602000035a78', '0', '90', '', '', '', '', '']
29 print(bufView[4])
30 // [MTrace] -> ['load(1)', '602000035a78', '0', '90', '', '', '', '', '']
31 print(bufView[0])
32 // [MTrace] -> ['load(1)', '602000035a70', '0', '90', '', '', '', '', '']
33
34 print('End MTrace')
35 // [MTrace] -> ['free', '602000035a70', '0', '90', '9', '0', '0', '', '']
36 // [MTrace] -> ['load(2)', '62d000762408', '0', '101', '', '', '', '', '']
37 // [MTrace] -> ['free', '62d000762400', '0', '101', '7', '0', '0', '', '']
```

(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Experiment - 분석 실험 2

- 분석 실험 결과 2 - 자바스크립트 메모리 할당 그룹 분석
  - 메모리의 사용 추적 데이터를 메모리 할당 고유 아이디별로 그룹화하여 표 형태로 시각화한 분석 결과.

[MTrace Group By Js Allocation No. 101]

Type	Address	Size	mId	stack size	stack address	mIdValue	JS Code	JS Line No
alloc	62d000762400	33264	101	26	7fce8a66d94a	-1	bufView[i] = str.charCodeAt(i);	7#22
load(2)	62d000762408	0	101				print('End MTrace')	24#1
free	62d000762400	0	101	7	0	0	print('End MTrace')	24#1

[MTrace Group By Js Allocation No. 90]

Type	Address	Size	mId	stack size	stack address	mIdValue	JS Code	JS Line No
alloc	602000035a70	16	90	17	7fce8a66d7f6	-1	var buf = new ArrayBuffer(str.length*2);	4#13
store(1)	602000035a78	0	90				bufView[4] = 'cafebabe'.charCodeAt(3);	20#25
load(1)	602000035a72	0	90				print(bufView[1])	18#1
load(1)	602000035a7a	0	90				print(bufView[5])	17#1
load(1)	602000035a70	0	90				print(bufView[0])	22#1
load(1)	602000035a70	0	90				print(bufView[0])	16#1
load(1)	602000035a78	0	90				print(bufView[4])	21#1
store(1)	602000035a70	0	90				bufView[i] = str.charCodeAt(i);	7#22
load(1)	602000035a74	0	90				print(bufView[2])	19#1
free	602000035a70	0	90	9	0	0	print('End MTrace')	24#1

(Source from V8 자바스크립트 엔진에서의 동적 메모리 추적 시스템 연구)

Agenda

- Background
- Limitation - in Javascript Engine
- Research Objective
- Technical Background
  1. V8's SourcePositionTable
  2. ASan's Shadow Memory
- Experiment
- Conclusion
- Reference
- Q & A

Conclusion

- V8 자바스크립트에서 메모리를 추적하여 분석하는 도구를 구현.

- V8's SourcePositionTable - 자바스크립트 라인의 Symbol 복원
- ASan's Shadow Memory - 메모리 할당영역 추적 성능 최적화.

- **성능 실험**, 크롬 벤치마크 도구 Octane에서 파일 쓰기 제거, 파일 쓰기 사용 .
  - 각각 평균 1.2x, 3.1x의 추적 오버헤드 발생.

- **분석 실험**, 자바스크립트 라인별, 할당 그룹별 메모리 분석.
  - 분석 실험을 통해 각 자바스크립트에서 발생하는 메모리 변화를 가시적으로 볼 수 있었고, 메모리의 할당과 사용 그리고 소멸 주기도 한 눈에 볼수 있었음.

- 향후 **취약점 자동 분석 분야**에서의 활용.
  - JSMTrace에서의 자바스크립트 메모리 추적 데이터를 기반으로, 향후 Fuzzing과 Symbolic Execution 분야에서 더욱 효과적인 자동 취약점 분석이 이루어 질 수 있을거라 기대함.

Reference

1. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pages 309–318, 2012.
2. B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. ACM SIGPLAN Notices, 47(6):239–250, 2012.
3. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007.
4. C. Lattner. Llvm ir, 2015. Available online at http://www.aosabook.org/en/llvm.html.
5. R. McIlroy. Firing up the ignition interpreter, 2015. Available online at http://v8project.blogspot.kr/2016/08/firing-up-ignition-interpreter.html.
6. L. Bak. What is v8?, 2008. Available online at https://developers.google.com/v8.

Thank you !

Eunchong Lee (Email: gr4ce@korea.ac.kr)
Signal Processing and Advanced Intelligence (SPAI)
Web: https://sites.google.com/site/securesiplab/