데이터베이스를 이용한 난독화된 바이너리의 트레이스 분석

Applying Database to Trace Analysis of Obfuscated Binaries

저자
(Authors)

황준형, 한태숙
Joonhyung Hwang, Taisook Han

출처
(Source)

한국정보과학회 학술발표논문집 , 2015.06, 1739-1741 (3 pages)

발행처
(Publisher)

한국정보과학회
KOREA INFORMATION SCIENCE SOCIETY

URL

http://www.dbpia.co.kr/Article/NODE06394536

APA Style

황준형, 한태숙 (2015). 데이터베이스를 이용한 난독화된 바이너리의 트레이스 분석. 한국정보과학회 학술발표논문집, 1739-1741.

데이터베이스를 이용한 난독화된 바이너리의 트레이스 분석†

황준형[O‡] 한태숙‡
‡한국과학기술원 전산학부
envia@envia.pe.kr han@cs.kaist.ac.kr

Applying Database to Trace Analysis of Obfuscated Binaries

Joonhyung Hwang[O‡] Taisook Han‡
‡School of Computing, KAIST

요 약

난독화된 바이너리는 구조가 복잡하게 변형되어 정적으로 분석하기 어려울 때가 많다. 이때 난독화된 바이너리도 원본 바이너리와 실행 결과는 같아야 한다는 점을 이용하여 동적으로 실행 트레이스를 생성해 분석하기도 한다. 난독화된 바이너리는 실행 결과에 영향을 주지 않는 명령을 다수 포함하고 있어 비대한 트레이스를 얻게 되며, 이를 효과적으로 처리할 수 있는 기법이 필요하다.

이 논문에서는 데이터베이스를 이용해 난독화된 바이너리의 트레이스를 처리하는 도구를 설계하고 구현하여 분석 작업을 수행한 결과를 소개한다. 데이터베이스를 이용해 안정적인 분석 도구를 편리하게 구현할 수 있었다.

1. 서 론

난독화(obfuscation)는 프로그램의 내부를 알아내기 어렵게 만드는 작업이다. 난독화가 적용된 프로그램은 원본 프로그램과 동일하게 동작하지만 분석해서 원본 프로그램에 대한 다른 정보를 얻어내는 것은 어렵게 변형되어 있다. 난독화는 소스 코드 수준에서도 적용되지만 최종 배포되는 바이너리에도 적용된다. 난독화는 프로그램의 도용이나 악의적인 변형을 막기 위해 적용되지만 악성 코드의 의도를 감추기 위해서도 쓰이므로 난독화된 프로그램에 대한 분석 기술이 필요하다.

난독화된 바이너리는 구조가 변형되어 정적으로 분석하기 어려울 때가 많다. 제어 흐름 평탄화(control flow flattening)[1]나 가상화(virtualization)[2]를 이용하여 난독화된 바이너리에서는 원본의 제어 흐름이 드러나지 않는다. 이러한 바이너리를 정적으로 분석하려면 난독화 구조를 알아야 하는데, 난독화 구조를 알아내려면 먼저 기초적인 분석 작업을 수행해야 한다. 난독화 구조는 임의로 복잡하게 만들어지기 때문에 분석 작업을 보조하는 도구가 있으면 도움이 된다.

난독화된 바이너리를 분석할 때 동적으로 실행 트레이스를 만들어 분석할 수 있다. 난독화된 바이너리도 원본 바이너리와 실행 결과는 같아야 하므로 실행 결과에 영향을 준 부분을 관찰하면 프로그램의 의미를 파악할 수 있다. 또한 프로그램의 실행 과정을 분석하면 난독화 구조도 파악할 수 있다[3].

실행 트레이스를 분석할 때 겪는 문제 중 하나는 트레이스의 크기이다. 난독화된 바이너리는 분석을 어렵게 하려고 실행 결과에 영향을 주지 않는 명령을 다수 포함하고 있어 비대한 트레이스를 얻게 된다. 이를 해결하기 위해 실행 결과에 영향을 주는 명령만 골라내는 작업이 필요하며 이에 관한 연구도 수행되었지만[4], 그 전에 트레이스를 이용하기 편리하게 저장해야 작업을 수월하게 진행할 수 있을 것이다.

이 논문에서는 데이터베이스를 이용해 난독화된 바이너리의 트레이스를 처리하는 작업을 수행하였다. 데이터베이스를 이용하면 대용량 자료를 편리하게 저장하고 처리할 수 있다. 트레이스에서는 이벤트가 발생할 때마다 해당되는 정보들을 저장하는데, 이벤트의 종류가 같으면 저장되는 정보들의 종류도 같으므로 이벤트 종류마다 테이블을 만들어 자연스럽게 저장할 수 있다.

데이터베이스를 사용했기 때문에 분석의 상당 부분은 질의(query)를 이용해서 처리하였다. 트레이스에서 서로 관련된 부분은 조인(join) 연산을 통해 묶어서 처리할 수 있다. 데이터베이스를 사용하지 않을 때에는 정보가 필요할 때마다 트레이스를 다시 읽거나, 필요한 정보를 모두 메모리에 올려 두어야 했다. 트레이스를 다시 읽는 것은 시간이 많이 걸렸으며, 정보를 모두 메모리에 올릴 때에는 메모리가 부족하여 실행이 중단되거나 잦은 스왑(swap)이 발생해 성능이 저하되었다.

2. 트레이스와 데이터베이스의 구조

트레이스를 저장하기 위해 XML 파일을 생성하였다. XML 파일의 구조는 그림 1과 같다. 전체 트레이스에 해당하는 `trace` 원소 안에 명령에 해당하는 `ins` 원소, 명령의 실행에 해당하는 `execution` 원소, 읽기에 해당하는 `read` 원소, 쓰기에 해당하는 `write` 원소가 들어 있다. 각 원소마다 구조가 동일하기 때문에 이를 바탕으로 테이블을 만들 수 있다.

† 본 연구는 2013년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (NRF-2011-0023847)

테이블을 생성한 SQL 구문은 그림 2와 같다. 테이블은 XML 파일의 ins 원소에 해당하는 ins 테이블, execution 원소에 해당하는 execution 테이블, read와 write 원소에 해당하는 access 테이블이 있다. 여기서 access 테이블은 type을 이용하여 read와 write를 구분하며, read는 'r'로, write는 'w'로 나타낸다. 또한 access 테이블의 행은 바이트 단위로 생성되는데, address 열은 바이트의 주소를 나타내고 name 열과 size 열은 원래 주소를 남겨두기 위해 사용한다.

```
<trace>…
<ins address="instruction address">
<rtn>routine name</rtn><sec>section name</sec>
<img>image ID</img><hex>instruction in hexadecimal</hex>
<dis>instruction disassembled</dis></ins>
<execution id="execution ID"
address="instruction address" tid="thread id" />
<read … execution="execution ID"
ip="instruction address" address="operand address"
size="operand size" tid="thread id" … />
<write … execution="execution ID"
ip="instruction address" address="operand address"
size="operand size" tid="thread id" … />
…</trace>
```

그림 1 트레이스 파일의 구조

```
CREATE TABLE ins (address INT PRIMARY KEY NOT
NULL, rtn TEXT, sec TEXT, img INT, hex TEXT, dis
TEXT);
CREATE TABLE execution (id INT PRIMARY KEY NOT
NULL, address INT NOT NULL, tid INT NOT NULL);
CREATE TABLE access (type TEXT NOT NULL,
execution INT NOT NULL, address INT NOT NULL,
name TEXT, size INT NOT NULL, tid INT NOT NULL,
value TEXT, PRIMARY KEY (type, execution,
address));
```

그림 2 테이블 생성

```
/* Header for Marker and Other Headers */
int main(int argc, char *argv[]) {
	int i, n, sum;
	n = atoi(argv[1]);
	/* Start Marker */
	for (i = sum = 0; i <= n; i++)
		sum += i;
	/* End Marker */
	printf("0 + ... + %d = %d\n", n, sum);
	return 0;
}
```

그림 3 정수 덧셈 프로그램

3. 실험 결과

트레이스는 Intel에서 나온 동적 바이너리 인스트루멘테이션(dynamic binary instrumentation) 도구인 Pin을 이용하여 Windows에서 생성하였다. 예제는 그림 3에 있는 정수 덧셈 프로그램을 이용하였다. 바이너리에 상용 난독화 도구인 VMProtect 2.13.6, Code Virtualizer 2.0.8, Themida 2.3.2를 적용하였다. 트레이스는 프로그램의 인자를 0, 1, 2, 9로 바꾸어 가면서 생성하였다.

트레이스 분석은 Linux에서 Python 2.7.9를 이용하였으며, 데이터베이스는 내장된 SQLite를 이용하였다. 시스템 CPU는 Intel Core i7-4790K를 사용하였고 7200 RPM 하드디스크와 32GB 메모리를 장착하였다. 트레이스와 데이터베이스 파일의 크기 및 변환 시간은 표 1과 같다.

표 1 트레이스 파일의 데이터베이스 변환

형태	인자	트레이스 크기(B)	DB 파일 크기(B)	시간(s)
원본	0	295,907,974	240,128,000	33
	1	295,904,423	240,084,992	31
	2	295,920,147	240,138,240	30
	9	296,024,444	240,149,504	30
VMProtect	0	333,330,658	269,631,488	36
	1	350,416,060	282,893,312	37
	2	367,438,883	296,609,792	39
	9	486,822,926	390,998,016	49
Code Virtualizer	0	350,024,133	289,451,008	37
	1	371,606,098	310,160,384	38
	2	393,195,075	330,068,992	42
	9	544,471,362	475,064,320	59
Themida	0	108,215,841,603	102,075,291,648	11,509
	1	109,577,658,477	103,376,094,208	11,488
	2	109,727,929,731	103,519,558,656	11,287
	9	108,741,868,803	102,597,793,792	11,277

분석으로는 각 명령의 실행 횟수를 세는 작업을 수행하였고, 실행 중 읽기만 하고 쓰지는 않는 메모리 주소 각각의 읽기 횟수를 세는 작업을 수행하였다. 명령의 실행 횟수를 세는 SQL 질의는 그림 4와 같다. 테이블에서 address를 기준으로 묶어서 세는 것을 볼 수 있다. 메모리의 읽기 횟수를 세는 SQL 질의는 그림 5와 같다. 각 메모리의 읽기 횟수를 센 다음 쓰기가 있는 메모리와 관련된 부분을 제거한다. 4096과 비교하는 부분은 레지스터와 관련된 부분을 제거하는 부분이다. 트레이스를 데이터베이스에 넣을 때 레지스터에 읽고 쓰는 것을 주소가 4096 이하인 메모리에 읽고 쓰는 것으로 옮겼다.

```
SELECT address, COUNT(*) FROM execution GROUP BY
address ORDER BY address;
```

그림 4 각 명령의 실행 횟수를 세는 질의

```
SELECT address, COUNT(*) FROM access WHERE
type="r" AND address > 4096 AND address NOT IN
(SELECT DISTINCT address FROM access WHERE
type="w" AND address > 4096) GROUP BY address
ORDER BY address;
```

그림 5 읽기만 하는 메모리 각각의 읽기 횟수를 세는 질의

명령 실행 분석을 수행할 때 걸린 시간과 사용한 메모리는 표 2와 같다. 전반적으로 데이터베이스를 사용한 쪽이 시간은 덜 걸리지만 메모리는 더 사용하는 경향을 보였으며, Themida로 난독화된 파일을 분석할 때에는 시간도 더 걸렸다. 데이터베이스를 쓰면 필요한 부분만 읽으며 분석할 수 있지만 처리 중 정렬을 수행해야 한다.

정렬이 필요하지 않도록 address의 인덱스를 만들면 메모리 사용량이 줄어드는 것을 확인할 수 있었다.

메모리 읽기 분석을 수행한 결과는 표 3과 같다. 여기서는 데이터베이스를 사용한 쪽이 전반적으로 시간과 메모리를 덜 사용한 것을 볼 수 있다. 특히 Themida로 난독화된 파일을 분석할 때 메모리 사용량의 차이가 크다. 이는 트레이스 파일을 직접 분석할 때에는 각 메모리 주소의 읽기 횟수 전부를 메모리에 저장했기 때문이다. Themida로 난독화된 파일은 읽기만 하고 쓰지는 않는 메모리 주소의 수가 다른 파일의 10배 정도 되었는데, 이것이 그대로 반영되었다.

표 2 명령 실행 분석을 위한 시간 및 메모리 사용

형태	인자	XML		DB	
		시간(s)	메모리(KB)	시간(s)	메모리(KB)
원본	0	3.10	10,048	0.22	14,188
	1	2.40	10,024	0.21	14,260
	2	3.30	9,816	0.21	14,224
	9	3.37	10,056	0.21	14,164
VMProtect	0	1.03	10,012	0.23	14,720
	1	1.08	10,056	0.24	14,232
	2	1.14	10,024	0.25	17,744
	9	1.54	9,984	0.34	17,556
Code Virtualizer	0	1.10	12,944	0.25	14,268
	1	1.14	12,852	0.26	17,672
	2	1.21	12,976	0.28	17,792
	9	1.66	13,048	0.36	17,772
Themida	0	728.80	20,756	841.43	1,486,304
	1	750.95	20,772	846.76	1,504,048
	2	664.54	20,584	865.02	1,507,460
	9	685.71	20,716	864.57	1,496,532

표 3 메모리 읽기 분석을 위한 시간 및 메모리 사용

형태	인자	XML		DB	
		시간(s)	메모리(KB)	시간(s)	메모리(KB)
원본	0	5.28	17,104	2.44	14,692
	1	5.23	17,072	2.44	14,680
	2	5.28	17,108	2.34	14,552
	9	5.24	17,044	2.48	14,628
VMProtect	0	6.74	17,060	4.04	14,648
	1	6.42	17,152	3.25	14,640
	2	6.57	17,108	3.46	14,572
	9	8.63	16,924	4.37	14,592
Code Virtualizer	0	6.28	17,108	3.14	14,648
	1	6.86	17,184	3.29	14,568
	2	6.90	16,944	3.12	14,604
	9	10.09	17,180	4.38	14,492
Themida	0	1956.33	170,392	1592.37	34,420
	1	1955.46	170,172	1691.82	34,492
	2	1946.67	170,404	1718.90	36,748
	9	1941.89	170,332	1731.35	36,476

4. 관련 연구

트레이스를 데이터베이스에 저장하는 것은 1980년대부터 시도되었다. 초기에는 병렬 시스템에 응용하는 것에 주로 초점을 맞추어 연구되었지만[5], 일반적인 디버깅에 적용하기도 하였다[6]. 트레이스뿐만 아니라 분석을 위한 보조 정보를 데이터베이스에 넣기도 하였다[7].

본 연구에서는 난독화가 적용된 바이너리에 초점을 맞추어 트레이스를 데이터베이스에 저장하였으며 수억 개의 실행이 담긴 대용량 트레이스를 대상으로 실험을 수행하였다.

5. 결론

난독화된 바이너리의 트레이스를 데이터베이스를 이용해서 분석하였다. 데이터베이스의 기능을 이용해서 기초적인 분석 도구를 용이하게 구현할 수 있었다. 앞으로 더욱 복잡하고 의미 있는 분석 작업에 데이터베이스를 적용해 보고자 한다.

실험 결과 분석 성능도 파일을 직접 분석할 때보다 전반적으로 양호하게 나타났다. 데이터베이스의 구조나 질의를 개선하면 더 나은 성능을 얻을 수 있을 것으로 기대한다.

참고문헌

[1] C. Wang, J. Davidson, J. Hill, J. Knight, “Protection of Software-Based Survivability Mechanisms”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 193-202, 2001.

[2] B. Anckaert, M. H. Jakubowski, R. Venkatesan, “Proteus: Virtualization for Diversified Tamper-Resistance”, *Proceedings of the 6th ACM Workshop on Digital Rights Management (DRM)*, pp. 47-58, 2006.

[3] M. Sharif, A. Lanzi, J. Giffin, W. Lee, “Automatic Reverse Engineering of Malware Emulators”, *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP)*, pp.94-109, 2009.

[4] K. Coogan, G. Lu, S. Debray, “Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach”, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pp.275-284, 2011.

[5] R. Snodgrass, “Monitoring in a Software Development Environment: A Relational Approach”, *Proceedings of the ACM Symposium on Practical Software Development Environments (SDE)*, pp.124-131, 1984.

[6] G. Pothier, É. Tanter, J. Piquer, “Scalable Omniscient Debugging”, *Proceedings of the 22nd ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp.535-552, 2007.

[7] B. Cleary, P. Gorman, E. Verbeek, M.-A. Storey, M. Salois, F. Painchaud, “Reconstructing Program Memory State from Multi-gigabyte Instruction Traces to Support Interactive Analysis”, *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pp.42-51, 2013.