

# Automated Crash Filtering for ARM Binary Programs

2015 IEEE 39<sup>th</sup> Annual International Computers, Software & Applications Conference  
Ki-Jin Eom, Joon-Young Paik, Seong-Kyun Mok, Hyeon-gu Jeon, Eun-Sun Cho, Dong-Woo Kim, Jaecheol Ryu

김영철

2016. 1. 20.

# Abstract

- 취약점 중에서 보안과 관련된 크래쉬를 구별
- "CrashFilter" 도구 제안  
: ARM Binary를 분석하여 exploit 가능한지 분석

# Introduction

- 테스트 도구와 퍼저를 이용하여 버그와 크래시를 탐지
  - ➔ 대부분 보안과 무관, 필터링 비용 발생
- exploit 가능한 크래시를 자동으로 구별하는 툴 존재
  - ➔ separate exploitable point를 가짐
- “!exploitable” : 크래시 지점에서 exploitable point를 추적
  - ➔ 완전한 exploitable point를 찾아주지는 않음
  - ➔ x86 machine에서만 작동

# Introduction

- "CrashFilter" 분석 도구 제안
  - ➔ "!exploitable"과 유사한 원리
  - ➔ ARM Binary에 초점
  - ➔ Static taint analysis & Binary analysis 기반으로 exploitable point 탐지

# Background

- Preliminaries

“crash point” – 크래시를 일으키는 명령

“exploitable point” – 예정된 명령을 조작할 수 있는 명령

➔ “CrashFilter”는 “crash point”에서 “exploitable point” 추적

# Background

- Taint analysis
  - ➔ taint된 데이터(source)의 target(sink)에 대한 효과를 탐지
  - ➔ 동적 방법 & 정적 방법
  - ➔ "Information Leak Detection"
    - source : secret information creation
    - sink : write operation
  - ➔ "CrashFilter"
    - source : crash point
    - sink : exploitable point

# Background

- ARM Binary Analysis  
    바이너리 분석의 어려움
  - ➔ Value Set Analysis : 값을 memory location으로 간주
  - ➔ LLVM : binary를 LLVM bitcode로 변환
- ➔ BinNavi
  - : ARM에서도 작동, IDA pro에 기반, REIL, 체계적인 명령 set
- ➔ CrashFilter
  - : BinNavi 플러그인 형태로 개발

# Static taint analysis on ARM binaries

- Source and Sink of Taint Analysis

Crash points

- ➔ 관련된 Load & Store instruction 추출
- ➔ 보안 관련 크래시는 memory operation 중에 발생

Exploitable points

- ➔ B, BL / STR, STM, SWP 등
- ➔ 분기문의 target을 조작하여 프로그램 흐름 변화
- ➔ 함수 포인터 등을 overwrite하여 흐름 변화



# Static taint analysis on ARM binaries

- Tracing Flows with Taint Analysis  
Reaching-definition analysis phase
  - ➔ 각 프로그램 포인트마다 numbering
  - ➔ gen : 새롭게 정의되는 변수
  - ➔ kill : 두 def의 dst가 같은 registe0이면 def2가 def1을 kill

Def-use analysis phase

- ➔ Def의 Dst와 Use의 Src가 같을 때, Def-use 성립

# Static taint analysis on ARM binaries

- Exploitability Detection  
Def-use의 use가 exploit 가능한지 분석

STR R1, [R2]

- ➔ R1이 taint 되면, 의도하는 값을 저장
- ➔ R2가 taint 되면, 의도하는 주소에 저장

BLX R1

- ➔ R1이 taint 되면, 의도하는 주소로 점프

# Static taint analysis on ARM binaries

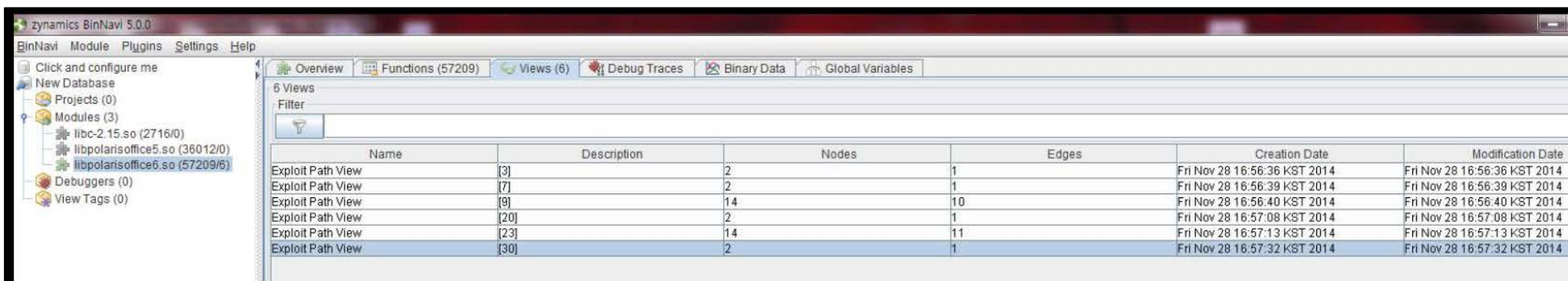
- Dealing with Memory

```
LDR R0, [R1]  
LDR R3, [R0, #0x30]  
BLX R3
```

- ➔ 명령의 subsequence를 교묘하게 사용하는 것은 추출 불가
- ➔ 위와 같이 L-L-B Pattern 과 같은 것들을 검출하기 위해 노력

# Implementation

- CrashFilter v1.0
  - ➔ BinNavi의 Java 플러그인 형태
  - ➔ BinNavi MonoReil 프레임워크를 통해 CFG를 다룸
  - ➔ E, PE, PNE로 분류
  - ➔ Exploitable points 리스트



The screenshot shows the BinNavi 5.0.0 application window. The 'Views' tab is active, displaying a table of 'Exploit Path View' entries. The table has columns for Name, Description, Nodes, Edges, Creation Date, and Modification Date. The entries are filtered by 'Exploit Path View'.

Name	Description	Nodes	Edges	Creation Date	Modification Date
Exploit Path View	[3]	2	1	Fri Nov 28 16:56:36 KST 2014	Fri Nov 28 16:56:36 KST 2014
Exploit Path View	[7]	2	1	Fri Nov 28 16:56:39 KST 2014	Fri Nov 28 16:56:39 KST 2014
Exploit Path View	[9]	14	10	Fri Nov 28 16:56:40 KST 2014	Fri Nov 28 16:56:40 KST 2014
Exploit Path View	[20]	2	1	Fri Nov 28 16:57:08 KST 2014	Fri Nov 28 16:57:08 KST 2014
Exploit Path View	[23]	14	11	Fri Nov 28 16:57:13 KST 2014	Fri Nov 28 16:57:13 KST 2014
Exploit Path View	[30]	2	1	Fri Nov 28 16:57:32 KST 2014	Fri Nov 28 16:57:32 KST 2014

<exploitable points와 관련된 데이터>

# Implementation

- Experimental Result
  - ➔ 여러 상황에 대해 기대한 결과를 출력
  - ➔ Polaris Office ver. 6에 대한 실험

Instruction Category of Crash Points	Total Crash	# of Security-related Crashes detected by CrashFilter1.0	
		<i>Optimistic Assumption</i>	<i>Pessimistic Assumption</i>
<b>LDR (load)</b>	55	6	7
<b>STR (store)</b>	14	0	2
<b>Arithmetic</b>	0	0	0
<b>Coprocessor</b>	1	0	0
<b>Total</b>	70	6	9

<Polaris Office ver.6에 대한 실험 결과>

# Related works and discussions

- “!exploitable”와 비슷한 동적 분석 도구들 존재
  - ➔ 크래쉬 이후의 분석 불가능
  - ➔ separate exploitable point 탐지 불가
- “CrashFilter”의 기능 증명에 대한 실험 더 필요
  - ➔ ARM에서 충분한 크래쉬를 확보하기 어려움
  - ➔ “Exploit generation” 도구를 활용하여 exploit 증명
  - ➔ ARM binary에 대한 “Exploit generation” 도구는 없다

# Conclusions and future works

- 보안 관련 크래쉬를 자동으로 필터링하는 도구 제안  
→ Static taint analysis를 이용
- memory analyzer를 추가
- 실제 ARM 취약점에 대한 더 많은 실험 수행