안드로이드 악성앱 분석을 위한 언패커 제작

팀　　명 : 압축풀어조
지도 교수 : 양환석 교수님
팀　　장 : 서동훈
팀　　원 : 강민영
전유민
정재훈

2022. 11.
중부대학교 정보보호학과

목 차

4. 결론

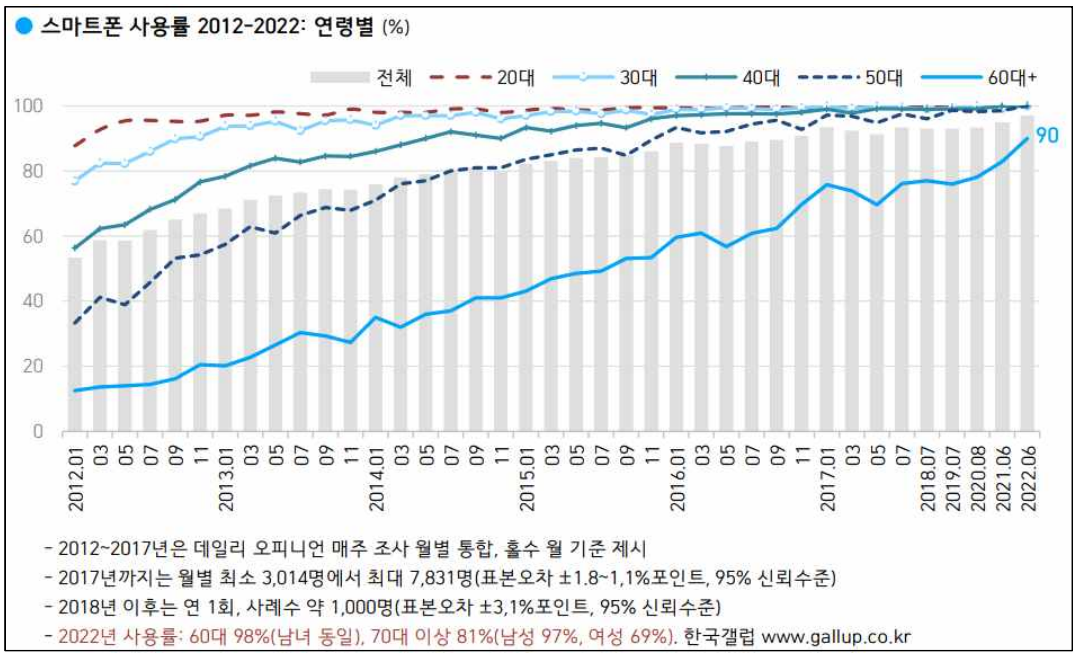
5. 별첨

1. 서론

1.1 연구 배경

1.1.1 스마트폰 사용 증가에 따른 피해 증가

갤럽과 같은 여론조사 기관에 따르면 스마트폰 사용량이 2014년에는 80%를 기록했지만 2022년 6월을 기준으로 17.1%가 증가한 97.1%를 기록하였다. 또한 그 중에서도 73%가 안드로이드 운영체제를 사용하고 있다. 그리고 [그림 2]와 같이 경찰청에서 조사한 정보에 따르면 2014년도부터 정보통신망을 이용한 범죄가 계속해서 증가하고 있음을 확인할 수 있는데 이는 스마트폰 사용량이 증가함에 따라 이를 대상으로 한 범죄 또한 증가하고 있음을 확인할 수 있다.



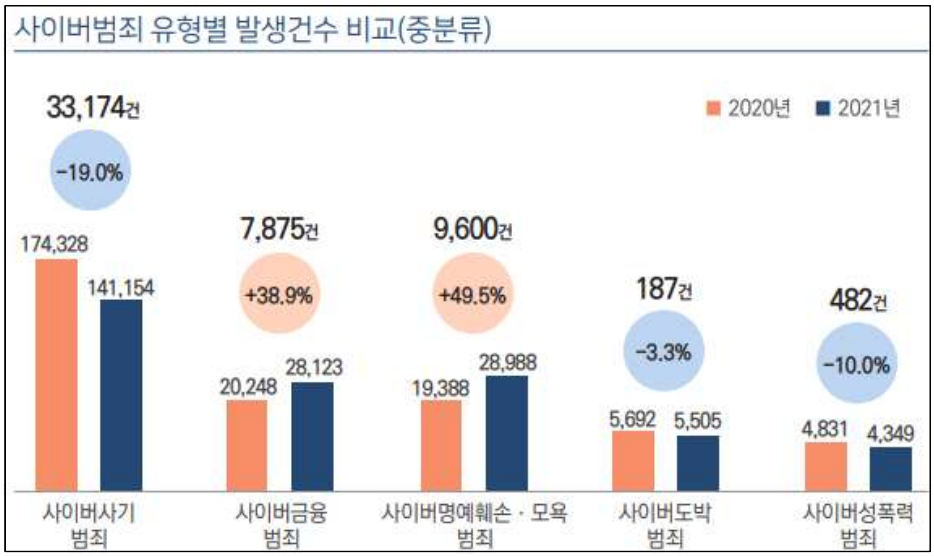
[그림 1 갤럽 - 2012~2022 스마트폰 사용률]

구분	총계			정보통신망침해범죄			정보통신망이용범죄			불법콘텐츠범죄		
	발생건수(건)	검거		발생건수(건)	검거		발생건수(건)	검거		발생건수(건)	검거	
		건수(건)	인원(명)		건수(건)	인원(명)		건수(건)	인원(명)		건수(건)	인원(명)
2014	110,109	71,950	59,220	2,291	846	1,171	89,519	56,461	38,579	18,299	14,643	19,470
2015	144,679	104,888	75,250	3,154	842	1,098	118,362	86,658	50,777	23,163	17,388	23,375
2016	153,075	127,758	75,400	2,770	1,047	1,261	121,867	103,172	42,871	28,438	23,539	31,268
2017	131,734	107,489	59,369	3,156	1,398	1,141	107,271	88,779	36,103	21,307	17,312	22,125
2018	149,604	112,133	60,138	2,888	902	1,048	123,677	93,926	35,738	23,039	17,305	23,352
2019	180,499	132,559	67,020	3,638	1,007	1,340	151,916	112,398	39,508	24,945	19,154	26,172
2020	234,098	157,909	74,256	4,344	911	1,037	199,594	134,969	43,541	30,160	22,302	29,678

[그림 2 경찰청 - 전체 사이버범죄 발생·검거 현황]

1.1.2 악성 APK 공격 증가

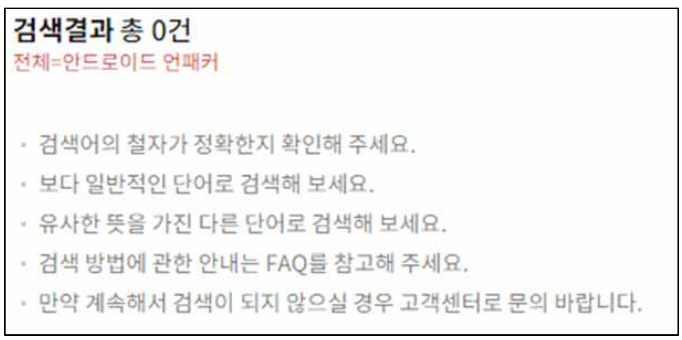
경찰청에서 공개한 문서에 따르면 사이버범죄 중 사이버금융과 관련된 범죄가 2020년도에 비해 2021년도 38.9%가 증가했음을 확인할 수 있다. 구체적으로 메신저피싱(12,402건→16,505건, 33.1%↑), 몸캠피싱(2,583건→3,026건, 17.2%↑), 스미싱(822건→1,336건, 62.5%↑)의 범죄가 특히 증가했음을 확인할 수 있다. 여기서 중요한 부분은 증가한 범죄 모두 주로 악성 APK를 통해 이루어진다는 점을 확인할 수 있다.



[그림 3 경찰청 - 사이버범죄 유형별 발생건수 비교]

1.1.3 국내 안드로이드 패커 관련 연구 불충분

국내 학술 데이터 베이스 사이트인 DBpia나 기술 블로그 등과 같은 곳에서 안드로이드 언패커와 관련한 연구가 거의 이루어지지 않다는 것을 확인할 수 있다.



[그림 4 DBpia - 안드로이드 언패커 검색 결과]

1.2 연구 필요성

이전 1.1에서 말한것 과 같이 휴대폰 사용률 급증에 따른 피해도 증가하고 있다. 또한 정보통신망이용범죄 중 사이버금융범죄가 특히 증가하고 있는데 특히 구체적으로 메신저피싱(12,402건→16,505건, 33.1%↑), 몸캠피싱(2,583건→3,026건, 17.2%↑), 스미싱(822건→1,336건, 62.5%↑)의 범죄가 증가하고 있음을 확인할 수 있다. 문제는 이와 같은 범죄에서 주로 악성 APK를 이용하여 범죄가 이루어지고 있는데 아직까지 국내에서는 이와 관련된 연구가 부족하여 이와 같은 주제의 연구가 필요하다고 생각되었다.

1.3 연구 목적 및 주제 선정

이 연구의 주 목적은 패킹된 악성 APK에서 덱스를 추출하는 도구를 제작하여 관련된 종사자나 연구자, 혹은 이외의 사용자들에게 도움을 주기 위한 목적으로 진행되었다.

주제 선정은 이전 목차에서 말한 것 과 같이 현재 메신저피싱, 몸캠피싱, 스미싱과 같은 사이버금융범죄에서 악성 APK를 이용한 범죄가 증가하고 있는데 아직까지 국내에서는 이와 관련된 연구가 부족하다고 생각하여 이와 같은 주제로 선정하게되었다.

2. 관련 연구

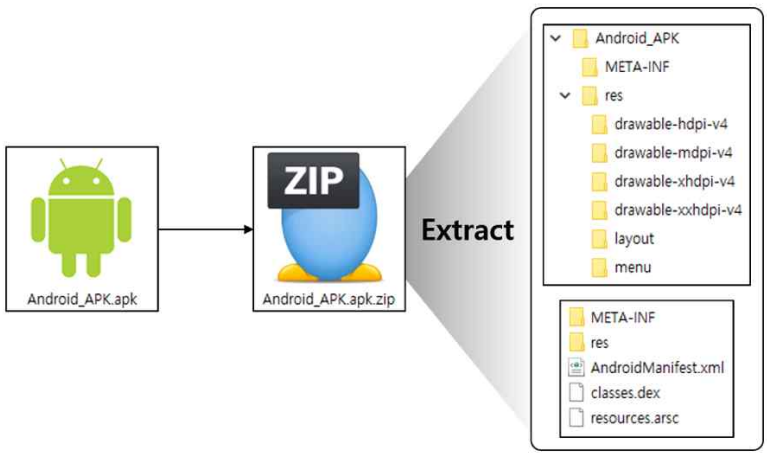
2.1 안드로이드

2.1.1 APK 구조

APK(Android Application Package)는 안드로이드의 소프트웨어와 미들웨어 배포에 사용되는 패키지 파일이며, '.apk'확장자를 가진다. 또한 ZIP파일 기반인 JAR을 기반으로 하는 압축 파일의 종류중 하나이다.

이러한 압축 파일인 APK는 다음과 같은 구조를 가지고 있다.

- AndroidManifest.xml : 애플리케이션에 대한 주요 정보(패키지 이름, 애플리케이션 구성 요소, 실행하는데 필요한 권한 및 액세스시 필요한 권한, 호환성)가 포함되어 있다.
- META-INF : 인증 정보가 포함된 폴더
- assets : 앱 실행에 필요한 자원이 모여있는 디렉토리로 주로 동영상, 일부 문서 템플릿와 같이 용량이 큰 데이터를 가지고 있으며, 빌드가 되지 않는다.
- res : 앱 실행에 필요한 자원이 모여있는 디렉토리로 빌드시 설치 파일에 포함되어 설치된다.
- kotlin : 애플리케이션이 코틀린으로 작성된 경우만 생성되며, 코틀린과 관련된 데이터가 포함되어있다.
- lib : 라이브러리 파일이 저장되어 있는 디렉토리
- resources.arsc : 컴파일된 리소스를 포함한다.
- classes.dex : Dalvik이 인식할 수 있도록 자바로 짜여진 코드가 컴파일되어 바이트 코드로 변환 된 소스 파일



[그림 5 APK 내부 구조]

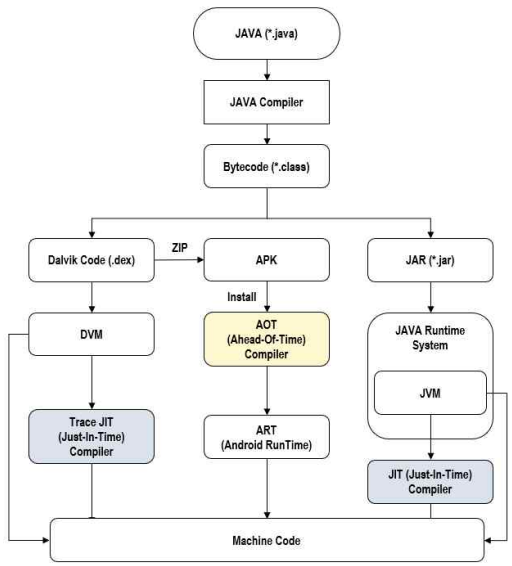
2.1.2 DVM, ART

DVM(Dalvik Virtual Machine)

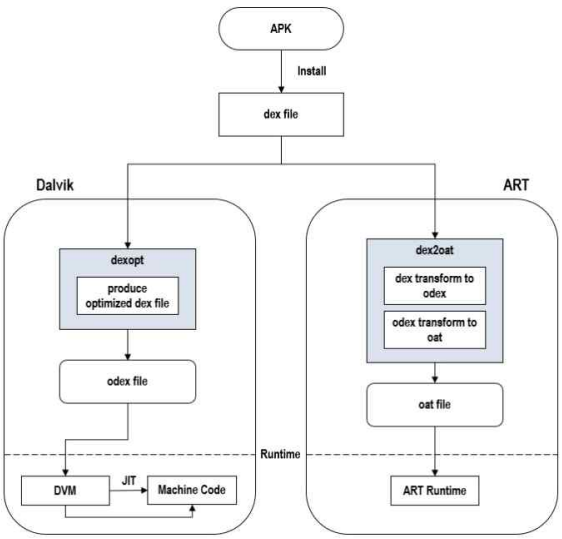
- 모바일 기기 특성상 배터리 수명, 컴퓨팅 파워 및 메모리가 데스크탑 환경에 비해 열악하기 때문에 모바일 기기 환경에 맞춰 나온 가상 머신
- JIT 컴파일러 사용
  - 자주 사용하는 부분에 대해 미리 컴파일하여 기계어로 해석해놓기 때문에 실행 성능을 향상시킬 수 있다.
  - 인터프리터 방식으로 실행하다 적절한 시점에 바이트코드 전체를 컴파일하여 네이티브 코드로 변경하고, 이후에는 인터프리팅 하지 않고 네이티브 코드로 직접 실행하는 방식
- 컴파일 과정
  - dexopt라는 도구를 사용해 dex 파일로 odex 파일을 만든다.
  - odex 파일은 Dalvik이 바로 실행할 수 있는 형태의 dex 파일이다.

ART(Android Runtime)

- 안드로이드 애플리케이션 런타임 환경으로 새로운 디버깅 기능과 좀 더 정확한 고수준의 애플리케이션 프로파일링 기능 제공
- 도입 시기
  - Android 4.4 (API 19)에서 처음 등장, 도입 (DVM과 선택적 사용)
  - Android 5.0 (API 21) 이후, 기본 런타임으로 지정
  - Android 7.0 이후로 AOT + JIT
- AOT 컴파일러 사용
- 설치 시점에 이미 컴파일을 완료하여 기계어로 해석을 끝냄
- 실행 시 해석 과정 없이 곧바로 기계어 실행
- 컴파일 과정
  - AOT 컴파일 시 dex2oat라는 도구를 사용해 dex 파일을 odex로 변경한 후 한번 더 oat 파일로 변경한다.
  - oat 파일은 Native machine code로 되어 있기 때문에 VM 없이 실행 가능하지만 Dalvik과의 하위 호환성을 제공해야 하기 때문에 VM 위에서 돌아가는 것 처럼 실행된다.



[그림 6 DVM, ART, JVM 컴파일 과정]



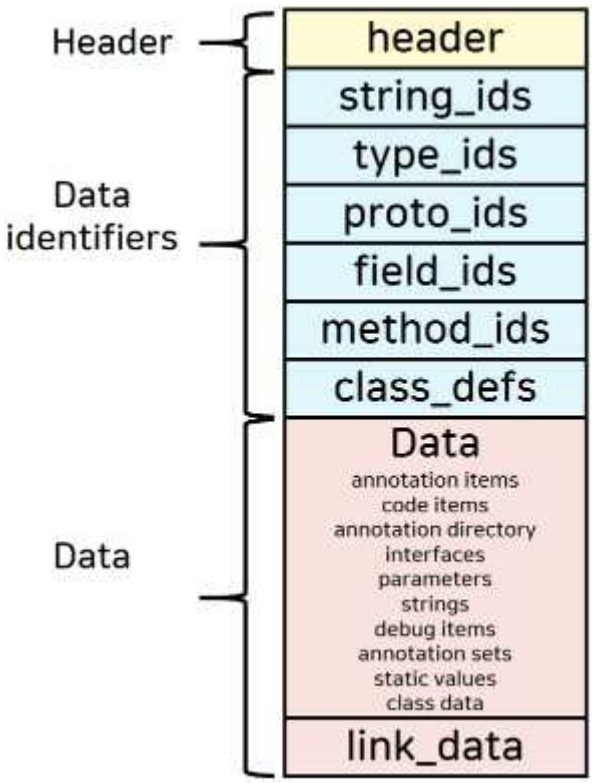
[그림 7 DVM, ART 상세 과정]

2.2 덱스(DEX)

2.2.1 덱스 구조

바이트 코드로 구성되어있으며, 세 가지 주요(헤더, 식별, 데이터) 섹션으로 이루어져 있다.

- 헤더 섹션 : 덱스 파일의 정보(체크섬, 크기, 오프셋 등)이 저장되어 있다.
- 데이터 식별 섹션 : 식별자 섹션에는 정의된 클래스(string_ids, type_ids, proto_ids, field_ids, method_ids, class_defs) 6개의 식별 목록이 포함되어 있다.
  - string_ids : 덱스 파일 내에서 사용하는 모든 문자열을 저장하는 영역
  - type_ids : string_ids 영역에 저장된 문자열의 성격을 저장하고 있는 영역
  - proto_ids : 덱스 파일 내에서 함수의 구조를 저장하고 있는 영역
  - field_ids : 클래스의 이름, 타입, 클래스, 패키지 이름을 저장하는 영역
  - method_ids : 메서드의 이름, 타입, 소속, 클래스 이름을 저장하는 영역
  - class_defs : 클래스에 대한 전체적인 정보와 데이터에 대한 기초 정보를 저장하는 영역
- 데이터 섹션 : 바이트 코드와 관련된 정보가 저장되어 있다.



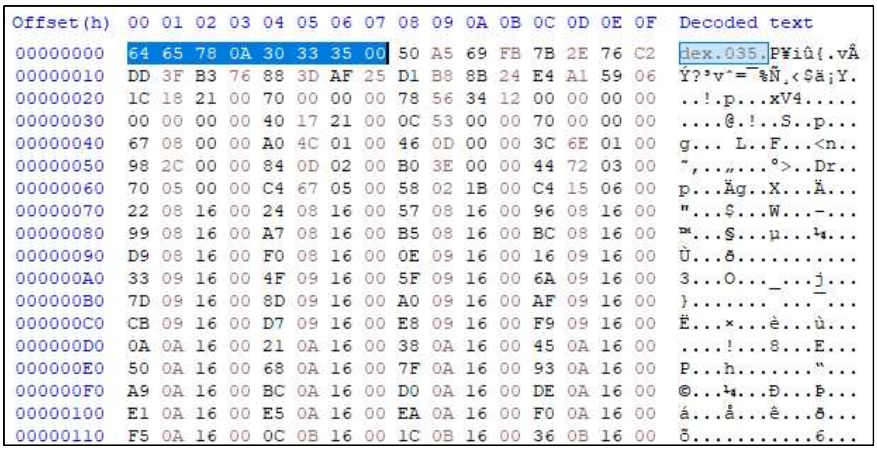
[그림 8 덱스 구조]

2.2.2 덱스 분석

DEX 파일의 가장 첫 부분인 header 구조를 확인해보면 다음과 같다.

파일 매직 넘버(File Magic Number) - 8바이트

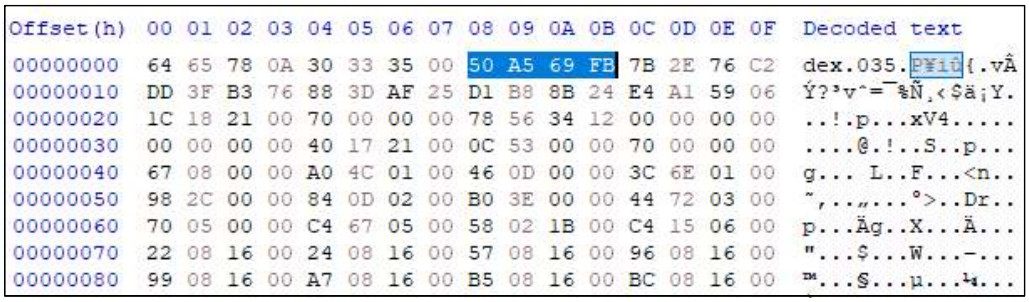
- 해당 파일이 무슨 파일인지 알려주는 부분
- 처음에 dex라는 텍스트와 버전 번호가 존재



[그림 9 파일 매직 넘버]

체크섬(Checksum) - 4바이트

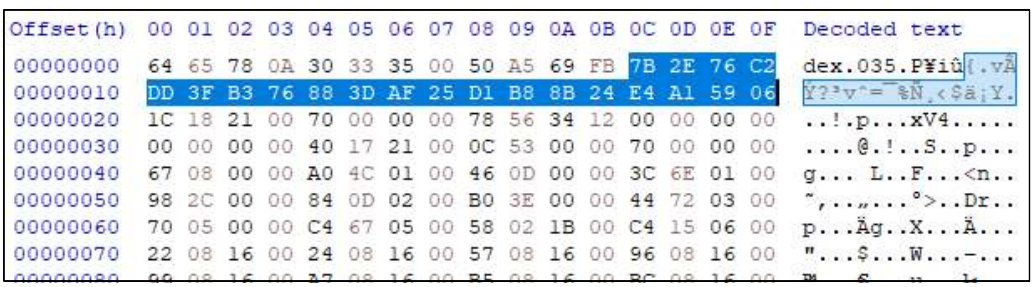
- 해당 파일이 변조 되었는지 확인하는 부분
- 파일 바이트 값이 손상된 경우 체크섬이 맞지 않아 Android Framework에서 apk 설치를 거부



[그림 10 체크섬]

시그니처(Signature) - 20바이트

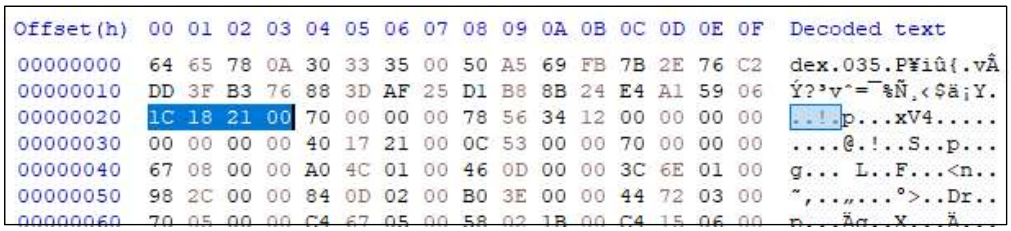
- 위의 매직 넘버, 체크섬 그리고 시그니처를 제외한 부분의 SHA-1 해시 값이며 파일을 고유하게 식별하는 데 사용



[그림 11 시그니처]

파일 크기(File_size) - 4바이트

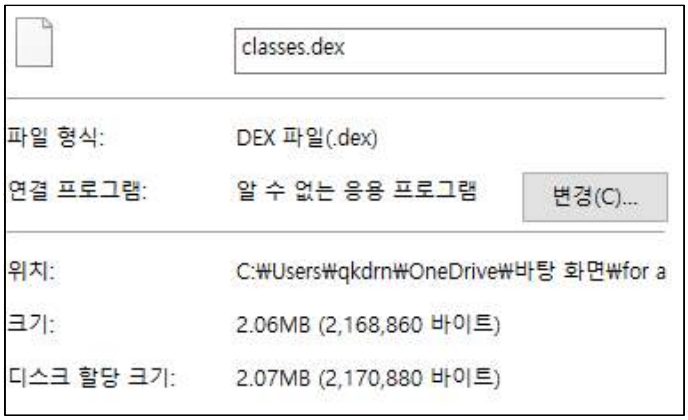
- 파일의 크기를 알려주는 부분
- 리틀 엔디안 방식으로 구성되어 있음



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...

[그림 12 파일 크기]

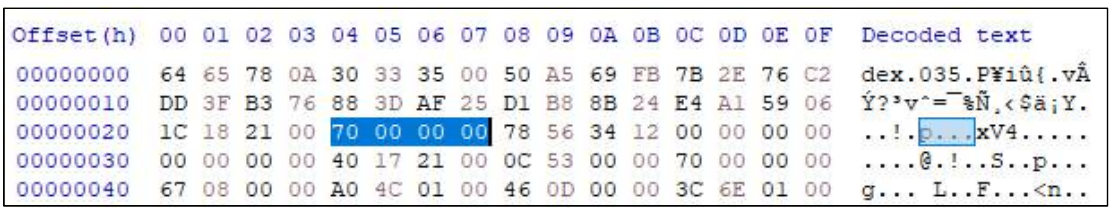
해당 부분은 파일 크기 영역이다. 위 내용은 리틀 엔디안 방식으로 구성되어 있으며, 변환해보면 0x0021181c이며, 이것을 10진수로 변환하면 2,168,860 바이트라는 것을 확인 가능하다.



[그림 13 실제 파일 크기]

헤더 크기(Header_size) - 4바이트

- 헤더의 크기를 보여주며, 값은 0x70으로 정해져 있다.

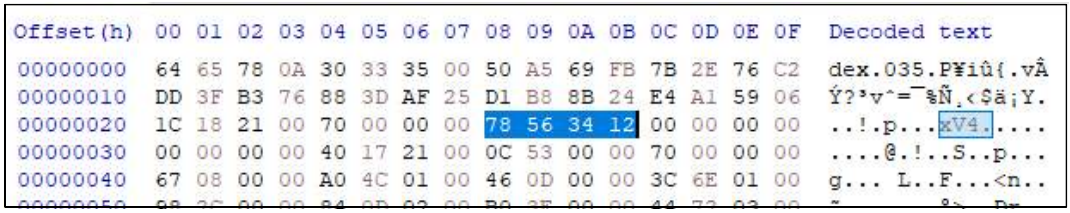


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..

[그림 14 헤더 크기]

엔디안 태그(Endian_tag) - 4바이트

- 이전에 크기를 구할 경우에 리틀 엔디안 방식으로 되어 있다고 언급하였는데, 정확히 알려면 해당 부분을 확인한다.
- 12345678로 되어 있다면 빅 엔디안, 78563412로 되어 있으면 리틀 엔디안

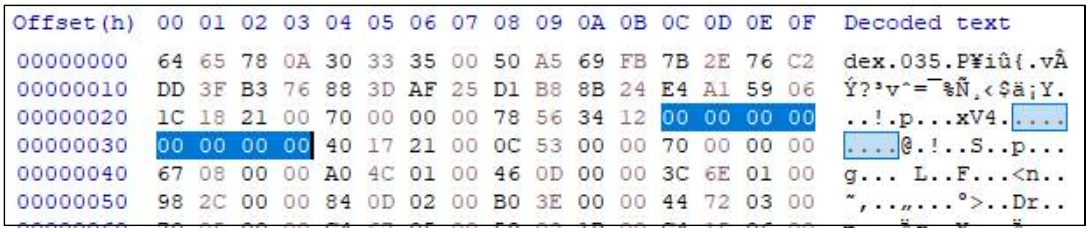


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..

[그림 15 엔디안 태그]

링크 사이즈, 링크 오프셋(link_size, link_off) - 4바이트

- 링크 사이즈는 처음 4바이트로 연결 섹션의 크기를 나타내고, 파일이 정적으로 연결되지 않은 경우 0을 갖는다.
- 링크 오프셋은 뒤에 4바이트로 파일의 시작 부분에서 연결 섹션까지의 오프셋이며 링크 크기가 0인 경우에 0이다.

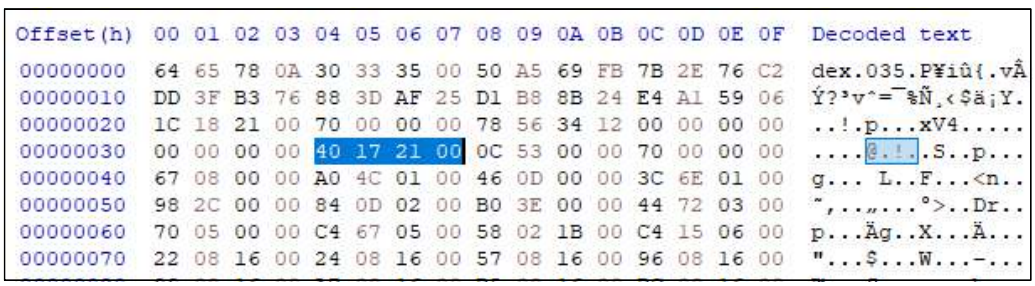


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..

[그림 16 링크 사이즈 및 오프셋]

맵 오프셋(Map_off) - 4바이트

- 맵 오프셋은 파일 시작 부분에서 맵 항목 까지의 오프셋
- 오프셋은 0이 아니어야 하고, data 섹션으로의 오프셋이어야 한다.

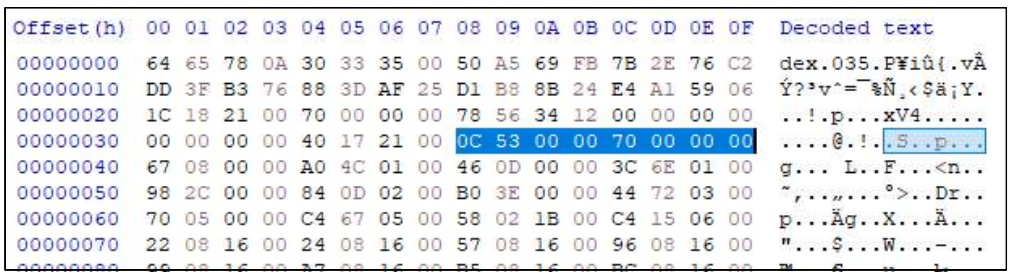


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...–...

[그림 17 맵 오프셋]

String_ids_size, String_ids_off - 4바이트 / 4바이트

- String_ids_size는 문자열 식별자 목록의 문자열 수
- String_ids_off는 파일의 시작 부분에서 문자열 식별자 목록까지의 오프셋



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...–...

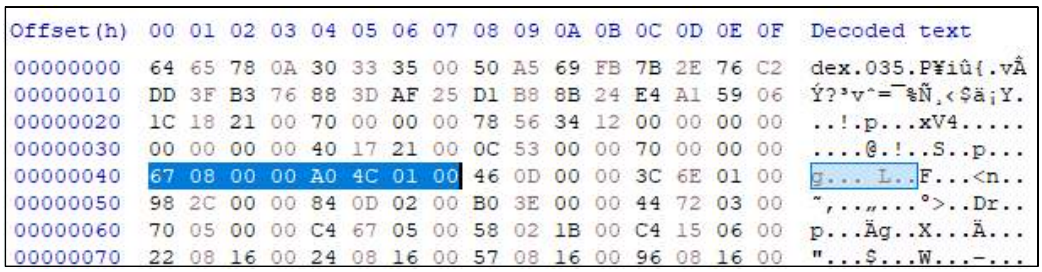
[그림 18 String_ids 섹션]

여기서 String_ids는 다음과 같다.

- DEX 파일에는 각 특징마다 고유한 섹션을 가지고 있다.
- String_ids 섹션은 DEX 파일 내에서 사용하는 모든 문자열을 저장하는 영역
  - String_ids_size : 0x0000530C
  - String_ids_off : 0x00000007

Type_ids_size, Type_ids_off - 4바이트 / 4바이트

- Type_ids 섹션은 String_ids 영역에 저장된 문자열의 성격을 저장하는 영역이다.
- type_ids_size는 형식 식별자 목록의 요소 개수
- type_ids_off는 파일의 시작 부분에서 형식 식별자 목록까지의 오프셋
- 오프셋이 0이 아닌 경우 type_ids 섹션의 시작 부분까지여야 한다.
  - type_ids_size : 0x00000867
  - type_ids_off : 0x00014CA0

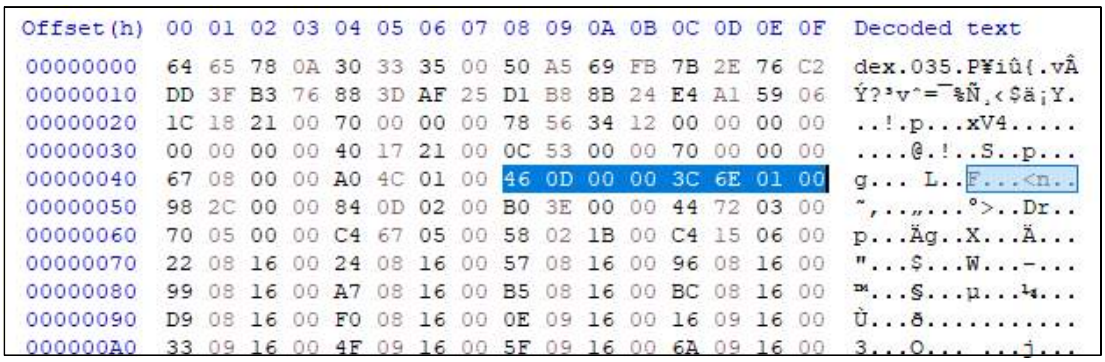


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...–...

[그림 19 Type_ids 섹션]

proto_ids_size, proto_ids_off - 4바이트 / 4바이트

- proto_ids 섹션은 dex 파일 내에서 함수의 구조를 저장하고 있는 영역이다.
- proto_ids_size는 프로토타입 식별자 목록의 요소 개수
- proto_ids_off는 파일의 시작 부분에서 프로토타입 식별자 목록까지의 오프셋
- 오프셋이 0이 아닌 경우 proto_ids 섹션의 시작 부분까지여야 한다.
  - proto_ids_size : 0x00000D46
  - proto_ids_off : 0x00016E3C

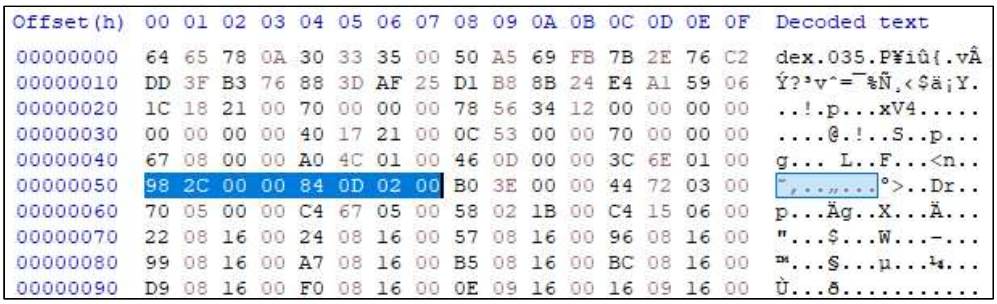


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...–...
00000080	99	08	16	00	A7	08	16	00	B5	08	16	00	BC	08	16	00	™...§...µ...¼...
00000090	D9	08	16	00	F0	08	16	00	0E	09	16	00	16	09	16	00	Ù...ð..........
000000A0	33	09	16	00	4F	09	16	00	5F	09	16	00	6A	09	16	00	3...O... ...j...

[그림 20 proto_ids 섹션]

field_ids_size, field_ids_off - 4바이트 / 4바이트

- field_ids 섹션은 클래스의 이름, type, class , package 이름을 제공하는 영역이다.
- field_ids_size는 필드 식별자 목록의 요소 개수
- field_ids_off는 파일의 시작 부분에서 필드 식별자 목록까지의 오프셋
- 오프셋이 0이 아닌 경우 field_ids 섹션의 시작 부분에 있어야 한다.
  - field_ids_size : 0x00002C98
  - field_ids_off : 0x00020D84

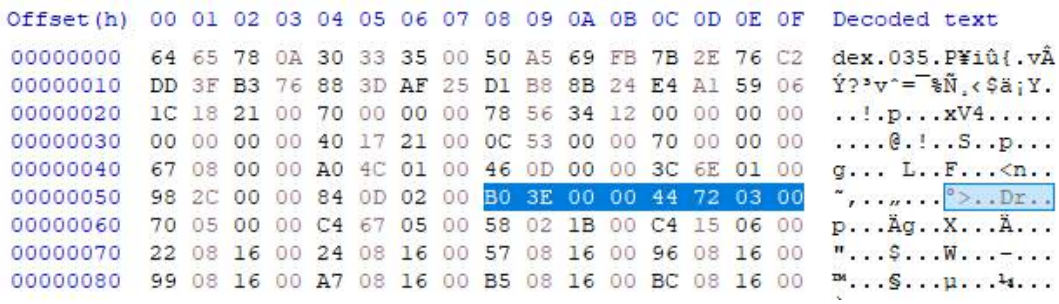


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,...„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...-...
00000080	99	08	16	00	A7	08	16	00	B5	08	16	00	BC	08	16	00	™...§...µ...¼...
00000090	D9	08	16	00	F0	08	16	00	0E	09	16	00	16	09	16	00	Ù...ð...........

[그림 21 field_dis 섹션]

method_ids_size, method_ids_off - 4바이트 / 4바이트

- method_ids 섹션은 method의 이름, type, 소속 class 이름을 저장하는 영역이다.
- method_ids_size는 메서드 식별자 목록의 요소 개수
- method_ids_off는 파일의 시작 부분에서 메서드 식별자 목록까지의 오프셋
- 오프셋이 0이 아닌 경우 method_ids 섹션의 시작 부분에 있어야 한다.
  - method_ids_size : 0x00003EB0
  - method_ids_off : 0x00037244

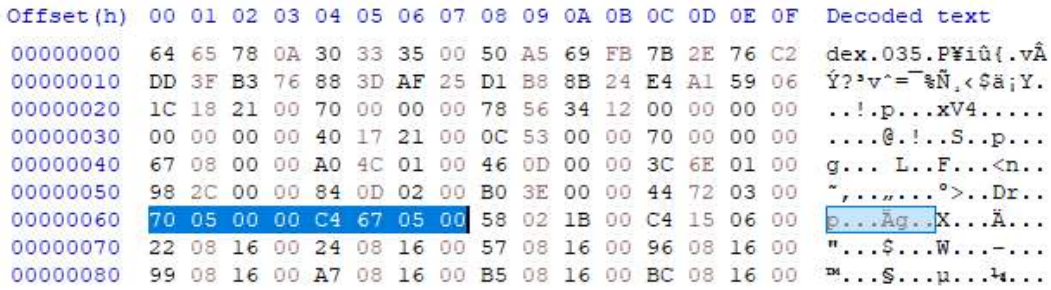


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,...„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...-...
00000080	99	08	16	00	A7	08	16	00	B5	08	16	00	BC	08	16	00	™...§...µ...¼...

[그림 22 method_ids 섹션]

class_defs_size, class_defs_off - 4바이트 / 4바이트

- class_defs 섹션은 class에 대한 전체적인 정보와 데이터에 대한 기초 정보를 저장하는 영역이다.
- class_defs_size는 클래스 정의 목록의 요소 개수
- class_defs_off는 파일의 시작 부분에서 클래스 정의 목록까지의 오프셋
- 오프셋이 0이 아닌 경우 class_defs 섹션의 시작 부분까지여야 한다.
  - class_defs_size : 0x00000570
  - class_defs_off : 0x000567C4



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	50	A5	69	FB	7B	2E	76	C2	dex.035.P¥iû{.vÂ
00000010	DD	3F	B3	76	88	3D	AF	25	D1	B8	8B	24	E4	A1	59	06	Ý?³v^=¯%Ñ¸‹$ä¡Y.
00000020	1C	18	21	00	70	00	00	00	78	56	34	12	00	00	00	00	..!.p...xV4.....
00000030	00	00	00	00	40	17	21	00	0C	53	00	00	70	00	00	00	....@.!..S..p...
00000040	67	08	00	00	A0	4C	01	00	46	0D	00	00	3C	6E	01	00	g... L..F...<n..
00000050	98	2C	00	00	84	0D	02	00	B0	3E	00	00	44	72	03	00	˜,..„...°>..Dr..
00000060	70	05	00	00	C4	67	05	00	58	02	1B	00	C4	15	06	00	p...Äg..X...Ä...
00000070	22	08	16	00	24	08	16	00	57	08	16	00	96	08	16	00	"...$...W...–...
00000080	99	08	16	00	A7	08	16	00	B5	08	16	00	BC	08	16	00	™...§...µ...¼...

[그림 23 class_defs 섹션]

data_size, data_off - 4바이트 / 4바이트

- data_size는 바이트 단위로 나타낸 data 섹션의 크기
- 여기서 sizeof(unit)의 짝수 배수여야 함
- data_off는 파일의 시작 부분에서 data 섹션 시작 부분까지의 오프셋
  - data_size : 0x001B0258
  - data_off : 0x000615C4

2.3 패커, 언패커

2.3.1 패킹

안드로이드는 바이트 코드로 이루어져 있으면서 덱스 파일 구조가 공개되어 있어 역공학에 취약하다. 이러한 역공학에 대한 피해를 줄이기 위해 난독화를 사용하였지만 최근 이를 무력화하는 방법이 많아졌다. 그래서 이를 보완할 수 있는 다른 방법이 바로 패킹 기법이다.

여기서 패킹이란 원본 덱스 파일을 보호하기 위해 사용하는 방법으로 주로 덱스 파일 은닉, 덱스 파일 덤핑 방지, 안티 리버싱 등을 사용한다.

2.3.2 덱스 파일 은닉

덱스 파일 수정(Dex file modification)

- 앱이 실행 중일 때 네이티브 코드를 사용하여 메모리의 Dex 파일을 수정한다
- 2015년 Baidu 패커에 의해 패킹된 앱은 메서드가 호출되기 직전에 유효한 명령어로 특수 메서드를 채우고 실행 후 삭제한다.
- 우회를 하기 위해 관련 동작을 캡쳐하고 적절한 순간에 인스트럭션을 덤프할 수 있다.

동적 클래스 로드(Dynamic class loding)

- 패커는 선택한 함수의 바이트 코드를 분리된 덱스 파일에 넣고 함수가 호출될 때 로드한다.
- 필요한 클래스를 로드하기 전에 덱스 파일을 암호화하고암호를 해독하기도 한다.
- 런타임 기능을 추적하면 로드된 후 Dex 파일을 덤프할 수 있다.

네이티브 방법(Native Method)

- 패커는 선택한 덱스 기능을 기본 메서드로 전환한 후 덱스 파일에서 JNI(Java Native Interface)를 통해 호출할 수 있다.
- 바이트코드를 재생성하기 위한 네이티브 코드를 리버싱하도록 설계되지 않았지만 크로스 레이어 모니터링 구성으로 네이티브 메서드에 대한 정보를 제공할 수 있다.

2.3.3 덱스 파일 덤핑 방지

패커는 일반적으로 언패커가 메모리에 있는 실제 코드의 덤프를 방지하기위해 세 가지 방법을 사용한다.

에뮬레이터 감지(Emulator detection)

- 대부분의 동적 분석 시스템이 안드로이드 에뮬레이터에 의존하기 때문에 특정 기술(Evading Android Runtime Analysis via Sandbox Detection)을 사용하여 패킹된 앱이 에뮬레이터에서 실행중이라면 강제로 종료한다.

안티 디버그(Anti-debug)

- 언패커를 사용하면 패키징된 앱에 디버거로 연결하여 앱을 모니터링하거나 덱스 파일을 가져올 수 있다.
- 패킹된 앱은 ptrace와 같은 함수를 통해 디버깅을 방지할 수 있다.

후킹(Hooking)

- 언패커가 메모리에 있는 덱스 파일에 액세스하고 덤프할 수 있다.
- 이를 방지하기 위해 패킹된 앱은 종종 파일 및 메모리 작업과 관련된 기능을 연결하여 후킹을 사용하지 못하도록 한다

2.3.4 안티 리버싱

- 정적 코드 분석을 통해 내부 로직을 이해하는 것을 방해하기 위해 난독화 등과 같은 기술을 사용한다.
- 리버싱을 방해하고 분석을 방해하는 기술
- 안티 리버싱 기법에는 안티 디버깅, 안티 디스어셈블링, 안티 템퍼링, 코드 암호화 및 난독화 등이 존재한다.
- 안티 리버싱이 상위 집합이고, 하위 집합으로 안티 디버깅, 안티 디스어셈블링, 안티 템퍼링 등으로 나뉘게 된다.
- 단지 안티(~에 반대되는)의 레벨이 디버깅 / 디스 어셈블리 / 메모리 레벨인지에 따라 분류되는 것이다.
- 이러한 기법뿐 아니라 코드를 암호화시키는 것도 리버싱을 방해하는 것이므로 안티 리버싱의 일종으로 보면 된다.
- 안티 디버깅 : 프로그램을 실행하면서 분석하는 디버깅을 방지
- 안티 디스어셈블 : 프로그램을 실행하지 않고 코드와 구조를 분석하는 디스어셈블링을 방지

- 안티 템퍼링 : 메모리 조작 방지
- 안티 메모리 덤프 : 메모리 덤프를 통해 크리덴셜 탈취 방지
- *안티 메모리 패치 : 실시간 메모리 패치로 프로그램 로직을 조작하는 것을 보호
- 안티 모니터, 안티 API 스캔 : API 분석으로 프로그램의 기능 파악을 방지
- 암호화 : 중요 데이터 보호
- 코드 가상화 및 코드 난독화 : 코드 분석을 어렵게 한다.

2.4 Yara

2.4.1 Yara Rules

Yara는 악성코드 샘플에 포함된 패턴을 이용하여 특성과 행위를 기준으로 악성 파일을 분류하는데 사용되는 도구이며, 리눅스와 윈도우 운영체제에서 모두 사용이 가능하다. Yara의 시그니처 탐지는 대표적으로 문자열 탐지와 바이너리 탐지가 존재한다.
문자열 탐지는 Value 타이틀에 속해 있는 문자열들을 탐지하는 방법이며, Condition을 이용하여 결괏값이 참인지 거짓인지 식별한다.

```
rule Alibaba : packer
{
  strings:
    $lib = "libmobisec.so"

  condition:
    $lib
}
```

[그림 24 Yara 문자열 탐지 방법]

바이너리 탐지는 파일 내부의 Hex 값을 탐지하는 기법으로 문자일 뿐만 아니라 16진수 값을 Rule에 적용할 수 있으며, [그림 25]와 같이 Wild Cards 를 사용하여 바이트를 랜덤으로 대체가 가능하다.

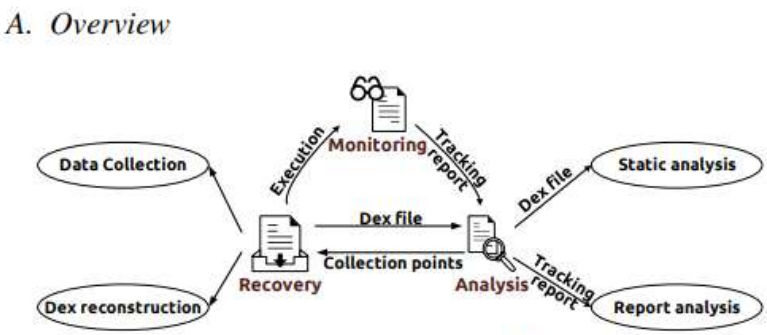
```
rule is_apk : file_type
{
  strings:
    $zip_head = {50 4B ?? ??}

  condition:
    $zip_head
}
```

[그림 25 Yara 바이너리 탐지 방법]

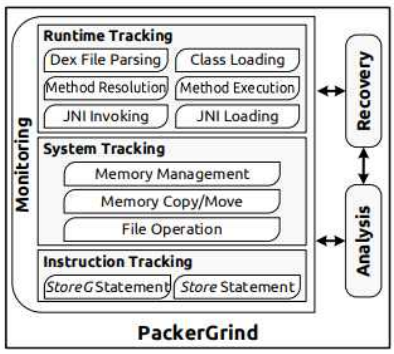
2.5 PackerGrind

2.5.1 서론

- PackerGrind는 언패킹 기술과 패킹 기술을 포함한 도구로 이전에 개발된 DexHunter, Androidunpacker, AppSpear 의 부족한 부분을 보완하여 성능을 개선한 도구이다.
- 총 세 가지 프로세스로 구성되며, 패킹된 앱을 실행할 때 런타임, 시스템, 명령어를 포함한 3개의 레이어를 통해 추적 보고서를 생성하거나 원본 덱스를 복구한다.
- 언패킹(덱스 복원)의 경우 DVM, ART 각각 환경에 맞는 방법을 통해 진행된다.



[그림 26 PackerGrind 과정]



[그림 27 PackerGrind 구조]

2.5.2 덱스 복원 (DVM)

덱스 파일 파싱

- 덱스 파일은 openDexFileNative()를 통해 저장소에서 가져오거나 openDexFile(byte[] array())를 통해 메모리에서 가져올 수 있다.
- 두 메서드 모두 dexFileParse()를 호출하여 덱스 파일을 구문 분석하고 런타임에 이 덱스 파일을 알아내기 위해 DexFile 구조를 반환한다
- DexFile은 dexFileParse()의 덱스 파일 헤더에 따라 달라지므로 dexFileParse()를 첫 번째 덱스 데이터 수집 지점으로 정한다.

클래스 로딩

- DVM은 defineClassNative()를 통해 클래스가 로드되며, 이 함수에서 dvmDefineClass()가 호출되어 클래스를 로드하고 클래스 정보(예: 필드, 메서드 등)가 포함된 ClassObject 구조를 반환한다.
- 덱스 파일에서 class_def_item 구조를 읽은 후 class_def_item의 오프셋에 따라 덱스 파일에서 class_data_item 구조를 분석한 후 ClassObject가 초기화된다.
- 따라서 두 번째 덱스 데이터 수집 지점으로 dvmDefineClass()를 지정한다.

Resolving Methods

- dexReadClassDataMethod()를 호출하여 덱스 파일에서 DexMethod를 가져온다.
- 그런 다음 LoadMethodFromDex() 에서 DexMethod에 따라 Method 구조를 생성한다.
- Method 초기화 중 dexCompareNameDexcriptorAndMethod()를 호출하여 finalize 메서드인지 확인한 다음 dexGetCode()를 호출하여 Dex 파일에서 코드 정보를 가져와

Method를 채운다.

- 인라인 함수 및 정적 함수의 기호는 libdvm.so에서 내보내지 않으므로 dexGetCode() 대신 dexCompareNameDexcriptorAndMethod()를 세 번째 Dex 데이터 수집 지점으로 설정합니다.

메소드 실행

- 네이티브 코드는 dvmInvokeMethod(), dvmCallMethodA() 및 dvmCallMethodV()와 같은 함수를 사용하여 Java 리플렉션 또는 JNI 리플렉션을 통해 Java 메서드를 호출할 수 있다.
- dvmInterpret()는 fast-interpreter, portable-interpreter 모두 사용하기 때문에 네 번째 덱스 숯비 지점으로 선택한다.

2.5.3 덱스 복원 (ART)

덱스 파일 파싱

- 덱스 파일은 openDexFileNative()를 통해 저장소에서 가져오거나 openDexFile(byte[] array())를 통해 메모리에서 가져올 수 있다.
- 두 메서드 모두 dexFileParse()를 호출하여 덱스 파일을 구문 분석하고 런타임에 이 덱스 파일을 알아내기 위해 DexFile 구조를 반환한다
- DexFile은 dexFileParse()의 덱스 파일 헤더에 따라 달라지므로 dexFileParse()를 첫 번째 덱스 데이터 수집 지점으로 정한다.

클래스 로딩

- DVM은 defineClassNative()를 통해 클래스가 로드되며, 이 함수에서 dvmDefineClass()가 호출되어 클래스를 로드하고 클래스 정보(예: 필드, 메서드 등)가 포함된 ClassObject 구조를 반환한다.
- 덱스 파일에서 class_def_item 구조를 읽은 후 class_def_item의 오프셋에 따라 덱스 파일에서 class_data_item 구조를 분석한 후 ClassObject가 초기화된다.
- 따라서 두 번째 덱스 데이터 수집 지점으로 dvmDefineClass()를 지정한다.

Resolving Methods

- dexReadClassDataMethod()를 호출하여 덱스 파일에서 DexMethod를 가져온다.
- 그런 다음 LoadMethodFromDex() 에서 DexMethod에 따라 Method 구조를 생성한다.
- Method 초기화 중 dexCompareNameDexcriptorAndMethod()를 호출하여 finalize 메서드인지 확인한 다음 dexGetCode()를 호출하여 Dex 파일에서 코드 정보를 가져와 Method를 채운다.
- 인라인 함수 및 정적 함수의 기호는 libdvm.so에서 내보내지 않으므로 dexGetCode() 대신 dexCompareNameDexcriptorAndMethod()를 세 번째 Dex 데이터 수집 지점으로 설정합니다.

메소드 실행

- 네이티브 코드는 dvmInvokeMethod(), dvmCallMethodA() 및 dvmCallMethodV()와 같은 함수를 사용하여 Java 리플렉션 또는 JNI 리플렉션을 통해 Java 메서드를 호출할 수 있다.
- dvmInterpret()는 fast-interpreter, portable-interpreter 모두 사용하기 때문에 네 번째 덱스 수비 지점으로 선택한다.

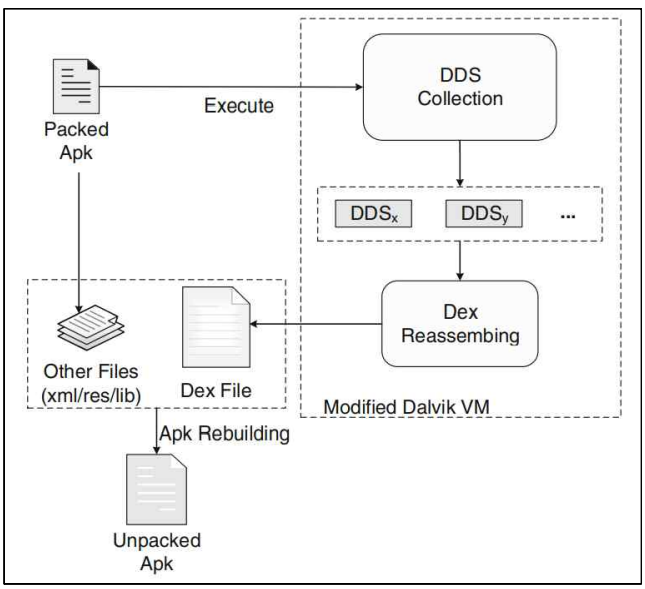
2.6 AppSpear

2.6.1 서론

- 안드로이드 악성코드 탐지 기술이 발전함에 따라 악성코드도 고급 코드 암호화를 구축해 대응하고 있다.
- 안드로이드 패커들은 종종 복잡한 분석 방지 방어를 채택하고 있으며 자주 진화하고 있다.
- AppSpear는 바이트 코드 해독 및 DEX(Dalvik Executive File) 재조립 방식으로, 패커 모르게 보호되는 바이트 코드를 효과적으로 복구할 수 있다.
- AppSpear는 DDS(Dalvik Data Struct)에서 해독된 바이트 코드 정보를 수집하도록 Dalvik VM에 직접 계측하고, 새로운 DEX 파일을 만들기 위해 정제된 재조립 프로세스를 수행하여 압축을 푼다.

2.6.2 AppSpear 언패킹

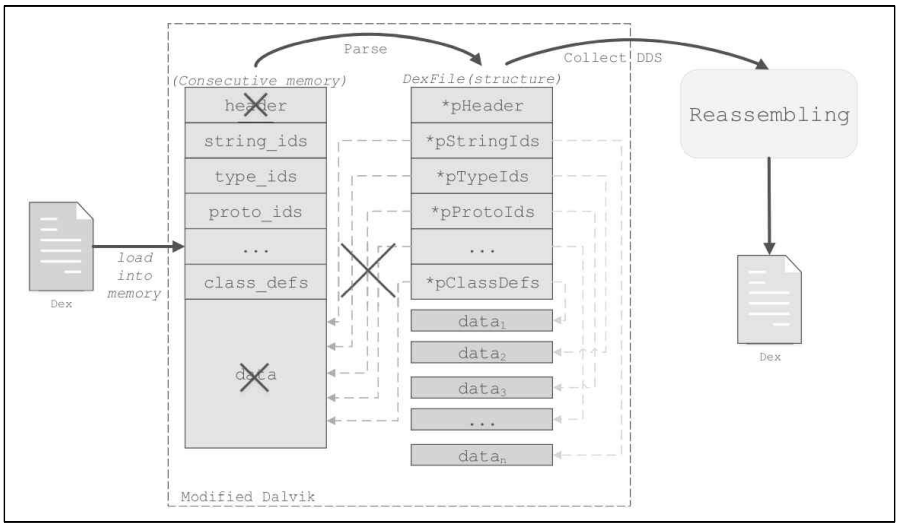
1. Android 패커의 다양한 분석 방지 조치를 우회
2. 메모리에서 DDS를 수집하고 DEX 파일을 재생성하기 위해 수정된 몇 가지 수정된 방법으로 수집된 DDS에 대해 재조립 프로세스를 수행
3. 분석 방지 코드를 절제하고 매니페스트 파일 및 기타 리소스와 DEX 파일을 추가로 합성



[그림 28 AppSpear 언패킹 과정]

2.6.3 Dex Reassembling

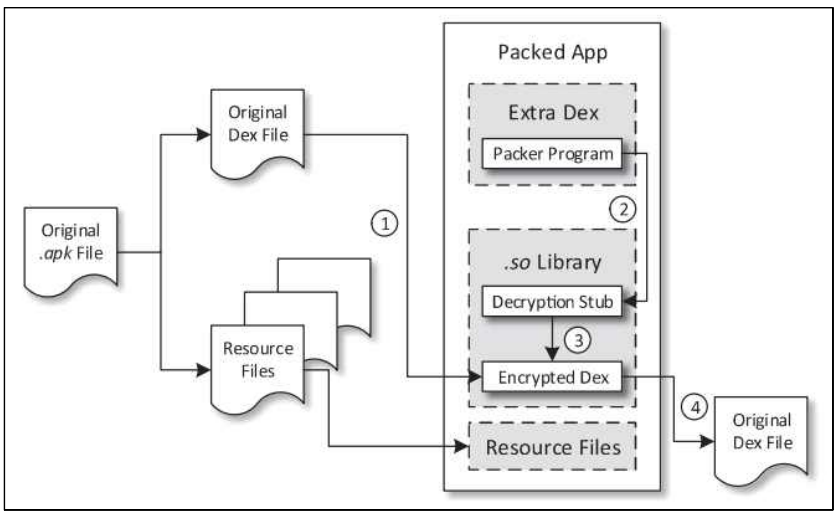
- 세분화된 바이트코드 수준 계측을 수행
- Dalvik VM의 고속 인터프리터를 수정하여 각 명령어의 인터프리팅 핸들러에 계측 스텁을 삽입
- 모든 opcode의 해석 코드의 맨 처음에 함수 호출 스텁을 삽입



[그림 29 Dex Reassembling Process]

2.6.4 패킹 앱 생성 및 실행

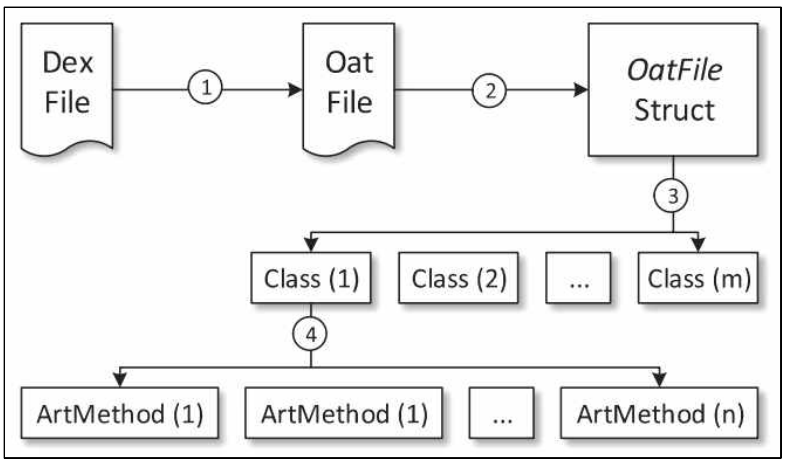
- apktool과 같은 자동화 도구에 의해 dex 파일이 디컴파일되는 것을 방지하기 위해 Android 패커는 먼저 전체 dex 파일을 암호화하고 .so 파일에 숨긴다.
- 패커는 패커 프로그램이 포함된 추가 dex 파일을 추가한다.
- .so 파일, 추가 dex 파일 및 원본 리소스 파일이 압축된 .apk 파일로 결합된다.
- 압축된 앱이 실행되기 시작하면 패커 프로그램은 .so 파일의 해독 스텁을 호출하고 암호화된 코드를 동적으로 해독한다.
- Dalvik VM 또는 ART는 DexClassLoader의 도움으로 해독된 dex를 로드하고 마침내 실행한다.



[그림 30 패킹된 앱의 생성 및 실행 과정]

2.6.5 ART 코드 패킹

- Compiling Bytecode: .apk 파일 설치 시 dex 파일을 oat 파일로 컴파일합니다. 일부 패커는 시스템에서 dex2oat를 사용하여 바이트 코드를 컴파일하고 컴파일 전에 실제 코드를 릴리스하도록 선택합니다.
- Parsing Oat File: oat 파일은 ART의 OatFile 구조체로 구문 분석됩니다. 일부 패커는 OatFile::open()을 코드 릴리스 지점으로 선택합니다.
- Loading Classes: ClassLinker::LoadClass()는 클래스 데이터를 로드하고 클래스 데이터를 Class 구조체로 구문 분석하는 데 사용됩니다. 코드 릴리스 포인트로도 사용됩니다.
- Loading Methods: ART는 ArtMethod 구조체를 사용하여 각 Java 메서드를 나타냅니다. 일부 패커는 ClassLinker::LoadMethod()가 호출될 때 각 메서드의 코드를 해제합니다.



[그림 31 ART의 코드 릴리스 포인트]

2.7 DexHunter

2.7.1 서론

- ART와 DVM에서 압축된 앱의 Dex 파일을 복구하는 DexHunter를 개발한다.
- DexHunter는 DVM과 ART를 포함하여 Android 가상 머신의 클래스 로딩 프로세스를 활용한다.
- DexHunter를 패킹된 앱에 적용하여 앱을 효과적으로 보호하고 원본 Dex 파일을 복구할 수 있다.
- 패커가 코드를 난독화하는 방법을 건드리지 않고 패킹된 앱에서 숨겨진 Dex 파일을 추출하는 방법에 중점을 두도록 한다.

2.7.2 덱스 복원 (DVM)

- dex 또는 jar 파일을 로드한 후 DVM은 파일 정보를 기록하는 DexOrJar라는 구조를 생성한다.
- fileName이라는 하나의 멤버는 파일의 위치를 나타내고, 열린 odex 파일을 나타내는 DvmDex 개체는 해당 DexOrJar 개체와 연결된다.
- DvmDex 개체에는 열린 dex 파일의 해당 메모리 영역을 유지 관리하는 memMap이라

는 멤버가 있는데, addr 멤버는 시작 주소를 저장하고 length 멤버는 메모리 영역의 길이를 나타낸다.

- 원하는 dex 파일을 덤프하기 위해 선택한 함수 Dalvik dalvik 시스템 DexFile defineClassNative에 코드를 추가하고 fileName의 값을 지정한다.
- fileName을 통해 dex 파일을 찾으면 대상 odex 파일의 메모리 영역도 관련 DvmDex 개체를 통해 알아낼 수 있다.
- memMap의 멤버 addr은 시작 주소를 나타내고 length 멤는 길이를 저장한다.
- smali/backsmali를 사용하여 odex 파일과 dex 파일을 복구한다.

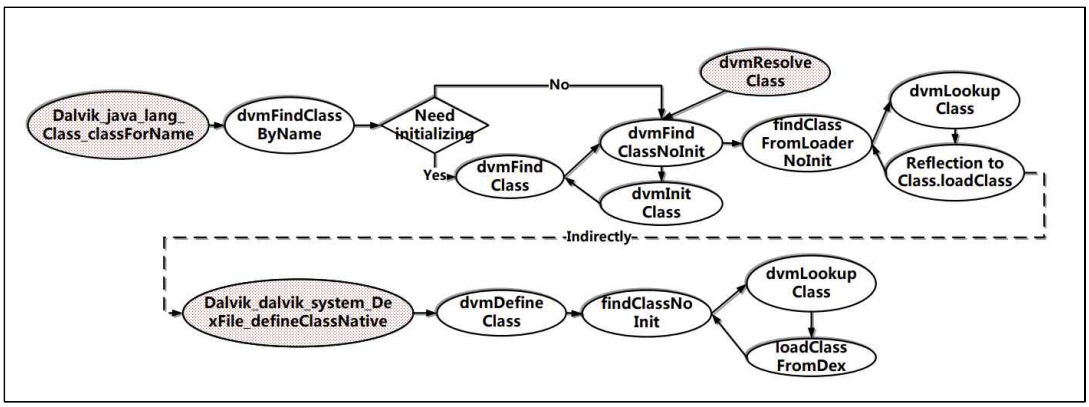
2.7.3 덱스 복원 (ART)

덱스 파일 피싱

- DexFile 개체에 다음을 추가하여 Dex 파일을 찾을 수 있도록 한다.
- 클래스가 로드될 때 위치 값을 확인하기 위해 DefineClass 함수에 코드 추가한다.

클래스 로딩

- libart.so에서 DexFile openDexFileNative라는 기본 메소드를 사용하여 dex 또는 jar 파일을 읽는다.
- 해당 oat 파일이 없으면 ART는 dex2oat라는 도구를 호출하여 dex 또는 jar 파일을 oat 파일로 컴파일한다.
- Class.forName을 호출하면 기본 메소드인 Class classForName이 호출된다.



[그림 32 클래스 로딩 과정]

메소드 실행

- DefineClass 함수의 매개변수이기도 한 DexFile 객체가 전달되고 스레드는 DexFile 객체가 참조하는 메모리 영역을 얻을 수 있다.
- DexFile::Begin() 및 DexFile::Size() 메서드를 호출하여 원본 dex 파일을 포함하는 메모리 영역의 시작 주소와 길이를 얻을 수 있기 때문에 원본 Dex 파일을 복구할 수 있다.

2.8 Native Unpacker

데프콘 22에서 공개된 언패커로 다음 패커에 대해 지원한다.

- Bangcle (SecNeo)
- APKProtect
- LIAPP (prerelease demo)
- Qihoo Android Packers
- Jaigu

2.8.1 로직 분석

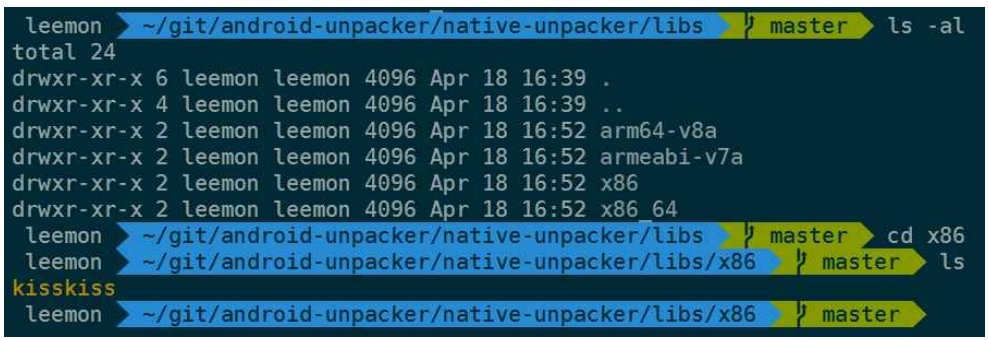
언패커의 전체적인 흐름은 다음과 같이 진행된다.

1. 프로세스와 스레드 메모리에 접근을 한다.
2. 패커 시그니처를 사용하여 사용한 패커의 종류를 판별한다.
3. 덱스, 오덱스 매직 넘버를 사용하여 덱스 혹은 오덱스를 추출한다.

2.8.2 도구 사용

Build

환경변수 설정 이후 make를 진행하면 다음과 같이 ./libs/[architecture]/kisskiss 가 생성된 것을 확인할 수 있다.



[그림 33 빌드 후 모습]

Setting

다음 명령어를 사용하여 분석을 진행할 기기에 빌드한 프로그램을 옮긴다.

- adb push ./kisskiss /data/local/tmp

실행

패킹된 앱을 설치한 기기에서 해당 앱을 실행하고 인자로 넘겨준다.

다음 [그림 34]를 통해 알 수 있듯이 /data/local/tmp/edu.killerud.diceroll.dumped_dex_0 에 언패킹한 파일이 저장되어 있다는 것을 알 수 있다.

```
beyond1q:/data/local/tmp/test # kisskiss edu.killerud.diceroll
WARNING: linker: /system/xbin/kisskiss: unsupported flags DT_FLAGS_1=0x8000001
[*] Android Dalvik Unpacker/Unprotector - <strazz@gmail.com>
 [+] Hunting for edu.killerud.diceroll
 [+] 7602 is service pid
 [+] 7627 is clone pid
 [+] Attempting to detect packer/protector...
  [*] Nothing special found, hunting for all dex and odex magic bytes...
  [*] No packer found on clone_pid 7627, falling back to service_pid 7602
 [+] Attempting to detect packer/protector...
  [*] Nothing special found, hunting for all dex and odex magic bytes...
 [+] Found 1 potentially interesting memory locations...
 [+] Attempting to search inside memory region 0xb5758000 to 0xb591a000
  [+] Memory region 0xb5758000 to 0xb591a000 contained anticipated class path edu/killerud/diceroll
 [+] Unpacked/protected file dumped to : /data/local/tmp/edu.killerud.diceroll.dumped_odex_0
1|beyond1q:/data/local/tmp/test # ls
beyond1q:/data/local/tmp/test #
```

[그림 34 Native Unpacker 실행 결과]

마지막으로 adb pull 명령어를 사용하여 언패킹한 파일을 추출한다.

```
C:\Users\leemon\Desktop\project\packer\18_diceroll_qihoo>adb pull /data/local/tmp/edu.killerud.diceroll.dumped_odex_0
/data/local/tmp/edu.killerud.diceroll.dumped_odex_0: 1 file pulled, 0 skipped. 33.8 MB/s (1843200 bytes in 0.052s)
```

[그림 35 언패킹 파일 추출]

2.8.3 오류 분석

툴을 사용하면 다음과 같은 결과들을 확인할 수 있다.

Install Error

- Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE: Package edu.killerud.kitchentimer signatures do not match the previously installed version; ignoring!]
  ⇒ 동일한 앱을 설치할 시 발생하는 에러
- Failure [INSTALL_FAILED_VERSION_DOWNGRADE]
  ⇒ 현재 설치하려는 앱 보다 이미 높은 버전의 앱이 러

No such file or directory

출력 결과를 통해 확인해보면 해당 프로세스와 스레드의 pid를 받아오지 못해 종료되는 것을 알 수 있다.

```
WARNING: linker: /data/local/tmp/kisskiss: unsupported flags DT_FLAGS_1=0x8000001
[*] Android Dalvik Unpacker/Unprotector - <strazz@gmail.com>
 [+] Hunting for nl.ttys0.simplec25k
 [+] -1 is service pid
  [!] Unable to check status of pid : No such file or directory
 [+] -1 is clone pid
  [!] Unable to check status of pid : No such file or directory
 [!] An error occurred attaching and finding the memory : No such process
```

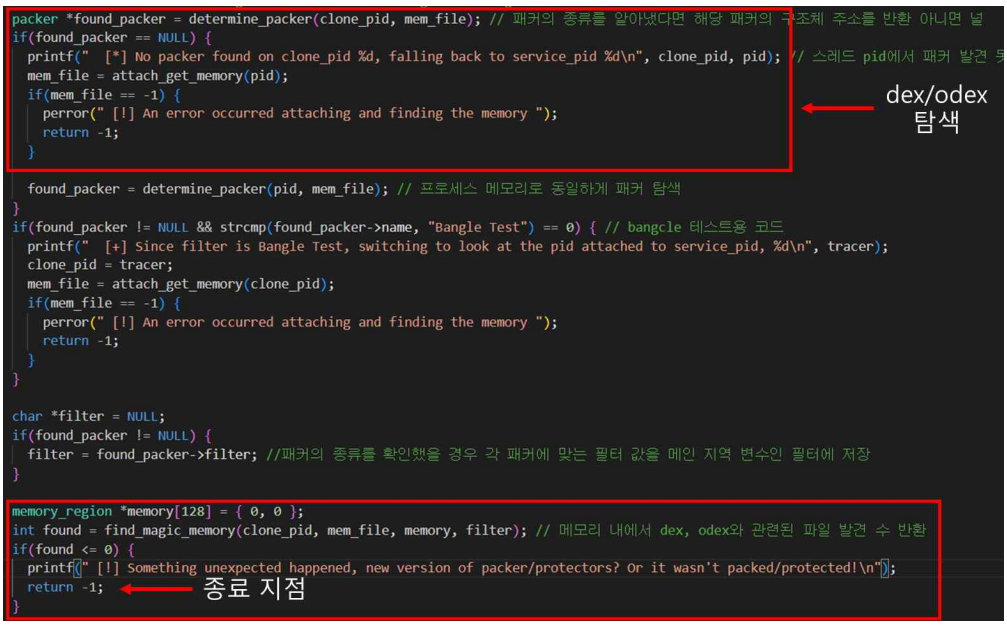
[그림 36 에러 출력 결과]

Something unexpected happened, new version of packer/protectors? Or it wasn't packed/protected!

```
WARNING: linker: /data/local/tmp/kisskiss: unsupported flags DT_FLAGS_1=0x8000001
[*] Android Dalvik Unpacker/Unprotector - <strazz@gmail.com>
 [+] Hunting for com.passcard
 [+] 16235 is service pid
 [+] 16246 is clone pid
 [+] Attempting to detect packer/protector...
  [*] Nothing special found, hunting for all dex and odex magic bytes...
  [*] No packer found on clone_pid 16246, falling back to service_pid 16235
 [+] Attempting to detect packer/protector...
  [*] Nothing special found, hunting for all dex and odex magic bytes...
 [!] Something unexpected happened, new version of packer/protectors? Or it wasn't packed/protected!
```

[그림 37 에러 출력 결과]

소스코드에서 확인해보면 메모리에서 dex와 odex를 탐색하였지만 발견하지 못해 종료되는 것을 확인할 수 있다.




```
packer *found_packer = determine_packer(clone_pid, mem_file); // 패커의 종류를 알아냈다면 해당 패커의 구조체 주소를 반환 아니면 널
if(found_packer == NULL) {
  printf("  [*] No packer found on clone_pid %d, falling back to service_pid %d\n", clone_pid, pid); // 스레드 pid에서 패커 발견 못
  mem_file = attach_get_memory(pid);
  if(mem_file == -1) {
    perror(" [!] An error occurred attaching and finding the memory ");
    return -1;
  }

  found_packer = determine_packer(pid, mem_file); // 프로세스 메모리로 동일하게 패커 탐색
}
if(found_packer != NULL && strcmp(found_packer->name, "Bangle Test") == 0) { // bangcle 테스트용 코드
  printf("  [+] Since filter is Bangle Test, switching to look at the pid attached to service_pid, %d\n", tracer);
  clone_pid = tracer;
  mem_file = attach_get_memory(clone_pid);
  if(mem_file == -1) {
    perror(" [!] An error occurred attaching and finding the memory ");
    return -1;
  }
}

char *filter = NULL;
if(found_packer != NULL) {
  filter = found_packer->filter; //패커의 종류를 확인했을 경우 각 패커에 맞는 필터 값을 메인 지역 변수인 필터에 저장
}

memory_region *memory[128] = { 0, 0 };
int found = find_magic_memory(clone_pid, mem_file, memory, filter); // 메모리 내에서 dex, odex와 관련된 파일 발견 수 반환
if(found <= 0) {
  printf(" [!] Something unexpected happened, new version of packer/protectors? Or it wasn't packed/protected!\n");
  return -1;
}
```

[그림 38 에러 처리 코드]

An error occurred attaching and finding the memory : Operation not permitted

```
WARNING: linker: /data/local/tmp/kisskiss: unsupported flags DT_FLAGS_1=0x8000001
[*] Android Dalvik Unpacker/Unprotector - <strazz@gmail.com>
 [+] Hunting for com.hlidskialf.android.pomodoro
 [+] 587 is service pid
 [+] 621 is clone pid
 [+] Attempting to detect packer/protector...
  [*] Nothing special found, hunting for all dex and odex magic bytes...
  [*] No packer found on clone_pid 621, falling back to service_pid 587
 [!] An error occurred attaching and finding the memory : Operation not permitted
```

[그림 39 에러 출력 결과]

코드에서 확인해보면 해당 apk에 사용된 패커를 찾지 못한 후 프로세스 메모리에 접근하려 했지만 해당 메모리에 접근하지 못해 종료된 것을 확인할 수 있다.

```
packer *found_packer = determine_packer(clone_pid, mem_file); // 패커의 종류를 알아냈다면 해당 패커의 구조체 주소를 반환 아니면 널
if(found_packer == NULL) {
  // 스레드 pid에서 패커 발견 못함 -> 프로세스 pid로 변경해서 진행
  printf("  [*] No packer found on clone_pid %d, falling back to service_pid %d\n", clone_pid, pid);
  mem_file = attach_get_memory(pid);
  if(mem_file == -1) {
    perror(" [!] An error occurred attaching and finding the memory ");
    return -1;
  }
```

종료 지점

[그림 40 에러 처리 코드]

2.8.4 언패킹 결과

이를 통해 전체적으로 다음과 같은 결과가 나왔다. 이때 안드로이드 5 버전으로 진행한 것은 안드로이드 7에서 라이브러리 오류로 인해 실행 자체가 되지 않아 분석을 진행할 수 없어 다시 진행하였다.

15 패커 종류	원본 Dex 추출 여부	16 패커 종류	원본 Dex 추출 여부	18 패커 종류	원본 Dex 추출 여부	19 패커 종류	원본 Dex 추출 여부
15_Ali	X	16_Ali	X				
15_Baidu	X	16_Baidu	O	18_Baidu	O	19_Baidu	X
15_Bangcle	X	16_Bangcle	X	18_Bangcle	X	19_Bangcle	X
15_Ijiami	O	16_Ijiami	X	18_Ijiami	O	19_Ijiami	X
15_Qihoo	O	16_Qihoo	O	18_Qihoo	O	19_Qihoo	X
15_Tencent	X	16_Tencent	X	18_Tencent	X	19_Tencent	X

[그림 41 Native Unpacker 언패킹 결과]

2.9 Frida Unpacker

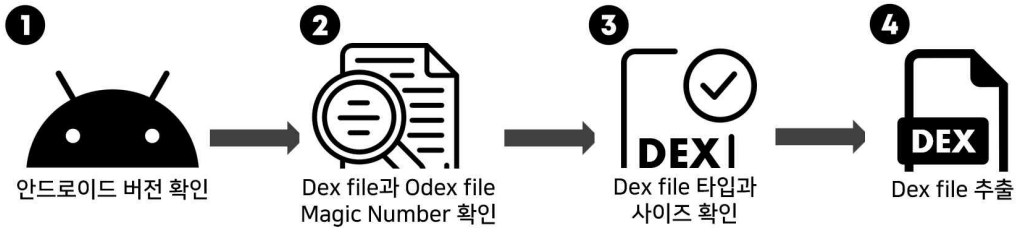
2020년 Github에 공개된 Fridroid-unpacker는 다음 패커에 대해 지원한다.

- Jiagu
- DexProtector
- DexGuard
- Yidun
- Tencent Legu
- Mobile Tencent Protect

2.9.1 로직 분석

Fridroid-unpacker의 전체적인 흐름은 다음과 같이 진행한다.

1. 안드로이드 버전을 확인한다.
2. 메모리에 접근하여 Dex 혹은 Odex의 매직 넘버를 확인한다.
3. 덱스 파일의 타입과 사이즈를 확인한다.
4. 매직 넘버가 확인이 가능하다면 덱스 파일을 추출하게 된다.



[그림 42 Frida Unpacker 언패킹 과정]

2.9.2 도구 사용

adb shell 접속

우선 애뮬레이터의 시스템 파일 확인을 위해 adb shell 명령어를 이용하여 접속한다.

```
C:₩Users₩yumin>nox_adb devices
List of devices attached
127.0.0.1:62001 device


C:₩Users₩yumin>nox_adb shell
←7←[r←[999;999H←[6n←8dream2qltechn:/ #
dream2qltechn:/ #
dream2qltechn:/ #
```

[그림 43 adb 접속 과정]

Frida 서버 실행

Frida 사용을 위해 애뮬레이터에 설치한 Frida 서버를 실행한다.

```
dream2qltechn:/data/local/tmp #
dream2qltechn:/data/local/tmp # ./frida-server-15.1.1-android-x86
```

[그림 44 Frida 서버 실행]

APK 설치

APK를 설치하게 되면 기본적으로 /data/data/ 경로에 패키지 시스템 파일이 저장된다. 설치된 패키지와 경로를 확인하였으므로 본격적으로 악성 apk를 설치한다.

```
15_ali.bat - Windows 메모장
파일(F)  편집(E)  서식(O)  보기(V)  도움말(H)
@ECHO OFF

nox_adb install D:₩adaptiveunpacker₩samples₩15₩2048_ali.apk
nox_adb install D:₩adaptiveunpacker₩samples₩15₩adsdroid_ali.apk
nox_adb install D:₩adaptiveunpacker₩samples₩15₩amplayer_ali.apk
nox_adb install D:₩adaptiveunpacker₩samples₩15₩androidcpg_ali.apk
nox_adb install D:₩adaptiveunpacker₩samples₩15₩Shopt_ali.apk
nox_adb install D:₩adaptiveunpacker₩samples₩15₩Shorty_ali.apk
```

[그림 45 APK 자동 설치 스크립트]

Frida 스크립트 실행

Frida를 실행하기 위한 명령문은 다음과 같다.

- frida -U -f com.package.target -l dexDump.js —no-pause

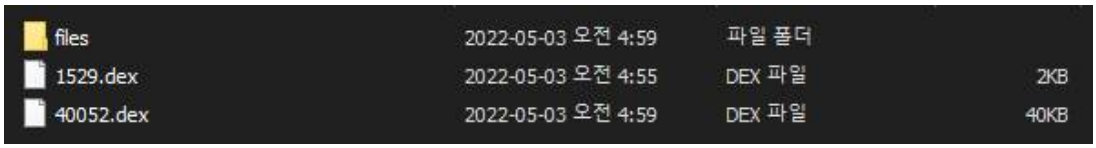
언패킹 결과 확인 (Dex 확인)

```
dream2qltechn:/data/data/com.orphan.amplayer # ls -al
total 216
drwxr-x--x   5 u0_a44 u0_a44   4096 2022-05-03 04:26 .
drwxrwx--x 113 system system   4096 2022-05-03 04:02 ..
-rw-------   1 u0_a44 u0_a44   1530 2022-05-03 04:26 1530.dex
-rw-------   1 u0_a44 u0_a44 195776 2022-05-03 04:26 195776.dex
drwxrwx--x   2 u0_a44 u0_a44   4096 2022-05-03 04:01 cache
drwxrwx--x   2 u0_a44 u0_a44   4096 2022-05-03 04:01 code_cache
drwxrwx--x   2 u0_a44 u0_a44   4096 2022-05-03 04:26 files
lrwxrwxrwx   1 root   root       39 2022-05-03 04:01 lib -> /data/app/com.orphan.amplayer-1/lib/arm
dream2qltechn:/data/data/com.orphan.amplayer #
```

[그림 46 언패킹 결과]

adb pull → Dex 파일 로컬로 가져오기

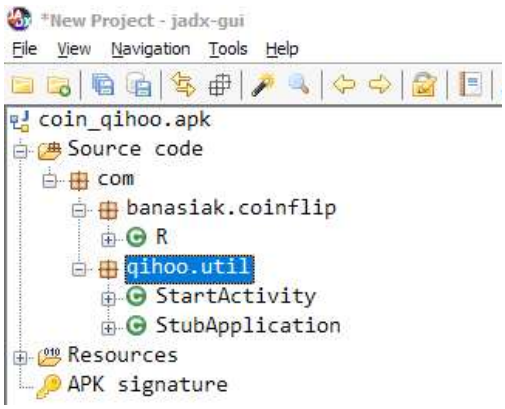
adb pull 명령어를 이용하여 Dex 파일을 로컬 컴퓨터로 복사하여 가져온다.



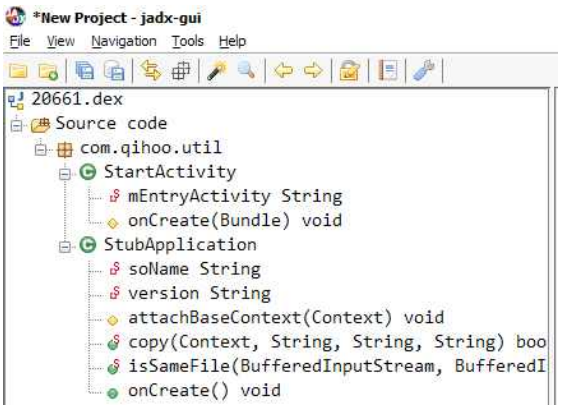
[그림 47 언패킹 결과 가져온 후]

2.9.3 사용 결과

Frida 언패킹을 통하여 원본 Dex가 추출되지 않는 예시이다. 언패킹이 제대로 이루어지지 않아 Stub 함수를 가진 Dex나 암호화 된 Dex가 그대로 추출된다. 다음 [그림 48], [그림 49]는 15 버전의 qihoo 패커가 사용된 coin.apk를 예시로 하였다. 확인해보면 해당 Dex는 원본 Dex가 아니라는 것을 예시를 통하여 확인할 수 있다.



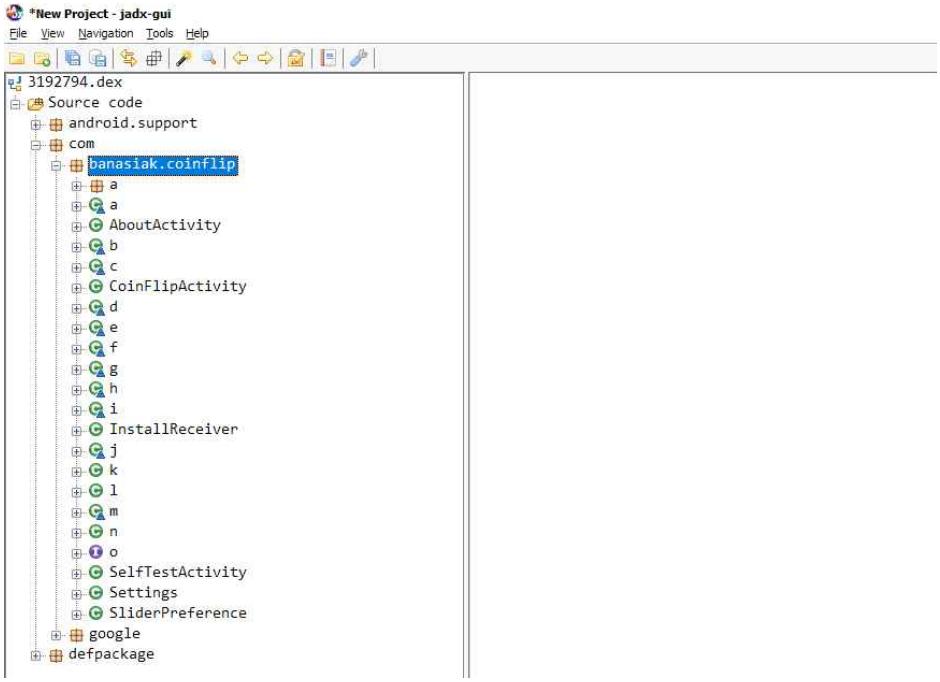
[그림 48 패킹된 앱 모습]



[그림 49 언패킹 후 모습]

이와 반대로 다음 패커는 정상적으로 원본 Dex가 추출된다.

확인해보면 apk는 암호화 된 Dex와 함께 Stub 함수가 존재하게 되며, 정상적으로 언패킹 된 Dex를 확인해보면 Stub 함수가 존재하지 않고 해당 패키지의 원본 코드로 보이는 것을 예시를 통하여 확인할 수 있다.



[그림 50 정상적으로 언패킹된 모습]

2.9.4 오류 분석

Process crashed: Trace/BPT trap – Android 7

- frida가 돌고 있는 게 탐지되면 종료시키는 에러

Process crashed: java.lang.UnsatisfiedLinkError – Android 7

- 버전 문제일 가능성이 큼
- libbaiduprotect_x86.so
- ibsecexe.x86.so → 앱을 보호하는 Package 파일의 역할이 주로 이 파일을 사용함. (ibsecexe.x86.so에 보호 모듈 존재)
- libexec.so → 버전 문제
- libprotectClass.so has unexpected e_machine: 40
- libbaiduprotect_x86.so → 얘가 대상 앱의 dex를 강화하고 패킹함 그래서 원본 덱스를 추출할 수 없음
- "/data/data/com.uberspot.a2048/.lib/libexec.so" has invalid shdr offset/size: 1177/21000 → 패킹할 때 헤더가 변경되지 않았을 때나 -android-shlib를 추가하지 않았을 때 발생하는 오류
- cannot locate symbol "__stack_chk_guard" referenced by "/data/data/앱패키지명/files/libjiagu.so"

Process crashed: java.lang.ClassNotFoundException – Android 7

- Didn't find class "com.banasiak.coinflip.CoinFlip" on path: DexPathList[[zip file "/data/app/com.banasiak.coinflip-1/base.apk"],nativeLibraryDirectories=[/data/app/com

.banasiak.coinflip-1/lib/arm, /system/fake-libs, /data/app/com.banasiak.coinflip-1/base.apk!/lib/armeabi, /system/lib, /vendor/lib]]

→ 클래스를 찾을 수 없어서 나타나는 오류

- Didn't find class "com.numguesser.tonio_rpchp.numberguesser.Guesser" on path: DexPathList[[zip file "/data/app/com.numguesser.tonio_rpchp.numberguesser-1/base.apk"],nativeLibraryDirectories=[/data/app/com.numguesser.tonio_rpchp.numberguesser-1/lib/arm, /system/fake-libs, /data/app/com.numguesser.tonio_rpchp.numberguesser-1/base.apk!/lib/armeabi, /system/lib, /vendor/lib]]

  → 클래스 찾을 수 없어서 나타나는 오류

Process crashed: Bad access due to invalid address – Android 7

- 안드로이드 버전이 맞지 않아서 발생하는 오류

  → 버전 9 이상을 쓰면 해결됨

Process terminated 오류 – Android 5

원본 Dex는 잘 추출되지만 Process terminated 오류가 가끔 발생한다.

```
C:\Users\Minyeong\minVS>frida -U -f com.idunnololz.igo -l dexDump.js --no-pause
     ____
    / _  |   Frida 15.1.17 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to SM-N976N (id=127.0.0.1:62026)
Spawning `com.idunnololz.igo`...
[09:39:07:049] [*] Android version: 5
[09:39:07:054] [*] Export index: 1754 -> _ZN3art7DexFile10OpenMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEENS3_9allocatorIcEEEEjPNS_6MemMapEPKNS_7OatFileEPS9_
[09:39:07:058] [*] ProcessName: com.idunnololz.igo
Spawned `com.idunnololz.igo`. Resuming main thread!
[SM-N976N::com.idunnololz.igo ]-> [09:39:07:089] magic : dex035
[09:39:07:089] size  : 21452
[09:39:07:089] dumped dex @ /data/data/com.idunnololz.igo/21452.dex

Process terminated
[SM-N976N::com.idunnololz.igo ]->

Thank you for using Frida!
```

[그림 51 오류 모습]

2.9.5 언패킹 결과

Frida를 이용한 원본 Dex 추출 여부에 대한 분석 결과는 다음과 같으며 다수의 결과를 토대로 작성하였다.

15 패커 종류	원본 Dex 추출 여부	16 패커 종류	원본 Dex 추출 여부	18 패커 종류	원본 Dex 추출 여부	19 패커 종류	원본 Dex 추출 여부
15_Ali	X	16_Ali	X				
15_Baidu	X	16_Baidu	X	18_Baidu	O	19_Baidu	O
15_Bangcle	X	16_Bangcle	X	18_Bangcle	O	19_Bangcle	O
15_Ijiami	X	16_Ijiami	X	18_Ijiami	X	19_Ijiami	X
15_Qihoo	X	16_Qihoo	X	18_Qihoo	X	19_Qihoo	X
15_Tencent	X	16_Tencent	X	18_Tencent	X	19_Tencent	X

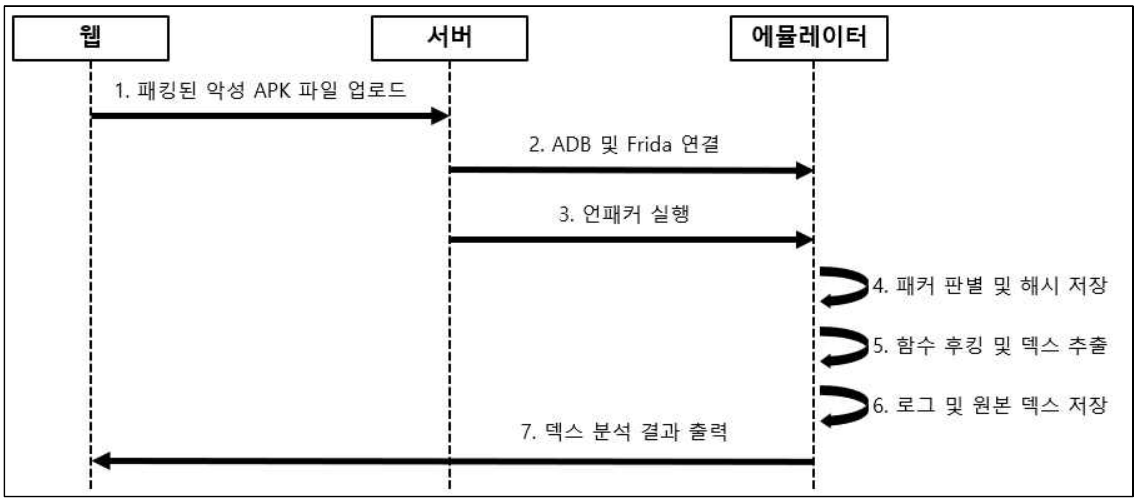
[그림 52 Frida Unpacker 언패킹 결과]

3. 본론

3.1 시스템 구성

시스템은 전체적으로 다음과 같이 진행된다.

1. 패킹된 악성 APK 파일을 웹을 통해 업로드한다.
2. 서버는 에뮬레이터와 ADB 및 프리다 서버 연결을 진행한다.
3. 업로드된 APK 파일을 사용하여 언패커를 실행한다.
4. 패커 판별을 진행하고 해당 파일의 해시 값을 저장한다.
5. 함수 후킹을 진행하여 원본 덱스를 추출한다.
6. 로그나 원본 덱스 그 이외 추가적인 정보를 저장한다.
7. 저장한 정보를 웹을 통해 출력한다.



[그림 53 시스템 흐름도]

3.2 프로그램 구성

언패커 프로그램은 내부적으로 Yara를 사용하여 시그니처를 탐지 및 파일의 해시를 저

장하는 부분과 함수를 후킹하여 원본 덱스와 원본 덱스에서 중요한 정보를 추출하여 저장하는 부분으로 이루어져 있다.

3.2.1 언패커

언패커 도구 개발은 이전 사전 연구를 통해 기존의 공개된 패커인 Native Unpacker와 Frida Unpacker를 비교 분석 후 더 성능이 뛰어났던 Frida Unpacker를 기준으로 진행하게 되었다.
언어는 Yara Rule이나 Frida Script를 사용해야한다는 점을 생각하여 Python 코드로 진행하였다.

1. ADB 연결

언패커의 시작은 먼저 에뮬레이터에 APK를 설치해야한다. 여기서 문제는 APK를 설치하기 위해서는 APK 파일을 에뮬레이터로 옮겨 설치하거나 ADB 명령을 이용하여 설치할 수 있는데 자동화를 위해서는 후자의 방법을 선택할 수 밖에 없었다. 최종적으로 Python에서 ADB 명령어를 사용하기 위해 pure-python-adb 라는 패키지를 설치하여 진행하였다. 코드에서 ADB 연결하는 부분은 다음과 같다.

```
def get_adb_connect(self):
    client = AdbClient()
    if not client:
        log.error_('Client connect error')

    devices = client.devices()
    if not devices:
        log.error_('Device not found')

    if len(devices) > 1:
        log.info_('Multiple devices detected, please select the device')
        for i in range(len(devices)):
            log.info_('%d : %s' %(i+1, devices[i].get_serial_no()))
        num = int(input('> '))
        self.adb = devices[num-1]
    else:
        self.adb = devices[0]
    log.info_('ADB connected : %s' %self.adb.get_serial_no())
```

[그림 54 ADB 연결 코드]

실제 연결 시 감지되는 기기가 하나라면 자동으로 연결되지만 다중의 기기가 감지될 시 다음과 같이 감지된 기기 중 하나의 기기를 선택하는 화면이 출력된다.

2. APK 설치

ADB 연결 후 adb.install 명령어를 사용하여 APK 파일 설치를 진행한다. 이때 중요한 점이 설치하기 직전에 에뮬레이터에 설치된 패키지 목록을 저장하고 설치 진행 후 다시 패키지 목록을 저장하는 것이다. 이러한 과정을 거치는 것은 향후 Frida 연결에서 사용하는 패키지 이름을 추출하는 과정 때문에 거치는 것이다.

```
=== Unpacking Start ===
[*] Multiple devices detected, please select the device
[*] 1 : 127.0.0.1:62026
[*] 2 : 127.0.0.1:62025
>
```

[그림 55 ADB 연결할 기기 선택 화면]

```
def install_apk(self, file):
    before = self.adb.shell('pm -l')
    if (self.adb.install(file)):
        log.info_('%s install success' %file)
    else:
        log.error_('APK install error')
    after = self.adb.shell('pm -l')
    dmp = diff_match_patch()
    diff = dmp.diff_main(before, after)
    dmp.diff_cleanupSemantic(diff)
    for i in diff:
        if (i[0] == 1):
            self.package_name = i[1][8:].strip('\n')
```

[그림 56 APK 설치 코드]

3. Frida 연결

앱의 패키지 이름을 사용하여 에뮬레이터에 저장된 Frida 서버와 연결을 진행한다. 이는 APK 설치 이후 실질적으로 언패킹을 진행하는 스크립트를 실행하기 위해선 Frida 서버와 연결이 사전에 이루어져야 하기 때문에 Frida 서버와 연결을 진행한다.

```
def get_frida_session(self):
    self.frida = frida.get_usb_device(1)
    self.pid = self.frida.spawn([self.package_name])
    self.session = frida.get_usb_device(1).attach(self.pid)
    log.info_('Frida attached : %s(%d)' %(self.package_name, self.pid))
```

[그림 57 Frida 연결 코드]

Frida 클라이언트 및 서버는 frida github를 통해 받을 수 있으며, 클라이언트의 경우 pip 명령어를 통해서 받을 수 있지만 서버의 경우 직접 환경에 맞는 파일을 받아 에뮬레이터에 설치해야한다.

```
d2q:/data/local/tmp # ls
frida-server-15.2.2-android-x86  re.frida.server
d2q:/data/local/tmp # ./frida-server-15.2.2-android-x86 &
[1] 4995
```

[그림 58 Frida 서버 실행]

```
C:\Users\okko2\OneDrive\Desktop\git\Android-Unpacker-Project\unpacker>frida --version
15.2.2
```

[그림 59 Frida 클라이언트 버전 확인]

4. 안드로이드 버전 획득

안드로이드 경우 버전에 따라 앱 실행 시 사용하는 라이브러리가 달라진다. 언패킹을 진행하기 위해서 앱 실행 시 사용하는 라이브러리를 후킹해야 하는데 이를 위해 사전에 안드로이드 버전을 획득 후 버전에 따라 후킹하는 라이브러리를 다르게 한다.

안드로이드 버전은 adb shell을 통해 "getprop ro.build.version.release" 명령어를 실행하는 것으로 쉽게 획득할 수 있다.

```
def get_android_version(self):
    self.and_ver = self.adb.shell('getprop ro.build.version.release').rstrip()
    log.info_('Android version : %s'%self.and_ver)
```

[그림 60 안드로이드 버전 획득 코드]

5. 함수 이름 획득

덱스를 추출하기 위해서 안드로이드에서 덱스를 읽는 과정에서 사용하는 함수를 추출할 필요가 있다. 이를 위해 이전 과정에서 안드로이드 버전을 얻었으며, 나머지는 라이브러리에 붙어서 해당 함수가 있는지 확인하여 추출하는 과정이 필요하다. 이를 위한 스크립트는 다음과 같다.

```
get_function = """
rpc.exports = {
    getFunc: function (and_ver) {
        var function_name = '';
        var i = 0;
        var android_version = parseInt(and_ver);
        if (android_version > 4) {
            var lib_name = android_version >= 10 ? 'libdexfile.so' : 'libart.so'
            var art = Module.enumerateExportsSync(lib_name);

            for(i = 0; i< art.length; i++){
                if(art[i].name.indexOf("OpenMemory") !== -1){
                    function_name = art[i].name;
                    return function_name;
                }else if(art[i].name.indexOf("OpenCommon") !== -1){
                    if (android_version >= 10 && art[i].name.indexOf("ArtDexFileLoader") !== -1)
                        continue;
                    function_name = art[i].name;
                    return function_name;
                }
            }
        }
        else{ //android 4
            var dvm =  Module.enumerateExportsSync("libdvm.so");
            if (dvm.length !== 0) {
                for(i = 0; i< dvm.length; i++){
                    if(dvm[i].name.indexOf("dexFileParse") !== -1){
                        function_name = dvm[i].name;
                        return function_name;
                    }
                }
            }else {
                dvm = Module.enumerateExportsSync("libart.so");
                for(i = 0; i< dvm.length; i++){
                    if(dvm[i].name.indexOf("OpenMemory") !== -1){
                        function_name = dvm[i].name;
                        return function_name;
                    }
                }
            }
        }
    }
};
```

[그림 61 함수 이름 획득 스크립트]

코드를 확인해보면 rpc.exports 선언 후 getFunc : function (and_ver) 부분을 확인할 수 있는데 이는 파이썬에서 frida js 스크립트를 사용할 수 있도록 하는 방법으로 다음과 같이 함수의 이름을 파이썬 코드로 얻을 수 있다.

```
def get_frida_session(self):
    self.frida = frida.get_usb_device(1)
    self.pid = self.frida.spawn([self.package_name])
    self.session = frida.get_usb_device(1).attach(self.pid)
    log.info_('Frida attached : %s(%d)' %(self.package_name, self.pid))
```

[그림 62 함수 이름 획득 코드]

6. 덱스 추출

덱스 추출은 이전까지 획득한 안드로이드 버전, 함수 이름, 패키지 이름을 사용하여 진행한다. 추출은 먼저 획득한 함수를 후킹한 후 해당 함수로 넘어오는 값에서 덱스 or 오덱스를 탐지하고, 만약 탐지가 되었다면 해당 값을 추출하는 것으로 진행된다.
가장 먼저 파이썬에서 넘겨주는 값을 변수에 저장하고 후킹할 함수의 저장은 다음과 같이 진행된다.

```
hook: function (func, proc, ver) {
    var moduleFuncName = func;
    var g_processName = proc;
    var g_AndroidOSVersion = parseInt(ver);
    console.log("[*] Dump dex start");

    if(moduleFuncName !== ""){
        var hookFunction;
        if (g_AndroidOSVersion > 4) {
            hookFunction = Module.findExportByName("libart.so", moduleFuncName);
        } else {
            hookFunction = Module.findExportByName("libdvm.so", moduleFuncName);
            if(hookFunction == null) {
                hookFunction = Module.findExportByName("libart.so", moduleFuncName);
            }
        }
```

[그림 63 인자 저장 및 후킹할 함수 저장 부분]

이후 함수를 후킹하고 넘어오는 인자에서 덱스, 오덱스 값이 있는지 확인한다.
후킹 코드는 다음과 같다.

```
Interceptor.attach(hookFunction,{
    onEnter: function(args){
        var begin = 0;
        var dexMagicMatch = false;
        var odexMagicMatch = false;

        dexMagicMatch = checkDexMagic(args[0]);
        if (dexMagicMatch === true){
            begin = args[0];
        }else {
            odexMagicMatch = checkOdexMagic(args[0]);
            if (odexMagicMatch === true) {
                begin = args[0];
            }
        }

        if (begin === 0){
            dexMagicMatch = checkDexMagic(args[1]);
            if(dexMagicMatch === true) {
                begin = args[1];
            }else{
                odexMagicMatch = checkOdexMagic(args[1]);
                if(odexMagicMatch === true) {
                    begin = args[1];
                }
            }
        }
        if (dexMagicMatch === true) {
            dumpDexToFile(dexMagicMatch, begin, g_processName);
        } else if(odexMagicMatch === true) {
            dumpDexToFile(odexMagicMatch, begin, g_processName);
        }
    },
```

[그림 64 함수 후킹 코드]

덱스나 오덱스의 판별은 다음과 같이 이루어진다.

```
function checkDexMagic(dataAddr) {
    var magicMatch = true;
    var magicFlagHex = [0x64, 0x65, 0x78, 0x0a, 0x30, 0x33, 0x35, 0x00];

    for(var i = 0; i < 8; i++){
        if(Memory.readU8(ptr(dataAddr).add(i)) !== magicFlagHex[i]){
            magicMatch = false;
            break;
        }
    }
    return magicMatch;
}

function checkOdexMagic(dataAddr) {
    var magicMatch = true;
    var magicFlagHex = [0x64, 0x65, 0x79, 0x0a, 0x30, 0x33, 0x36, 0x00];

    for(var i = 0; i < 8; i++){
        if(Memory.readU8(ptr(dataAddr).add(i)) !== magicFlagHex[i]){
            magicMatch = false;
            break;
        }
    }
    return magicMatch;
}
```

[그림 65 덱스 및 오덱스 판별 코드]

감지한 덱스 파일은 다음 코드와 같이 에뮬레이터 내부 /data/data에 저장된다.

```
function dumpDexToFile(isDex, begin, g_processName) {
    var dexType;
    isDex ? dexType = "dex" : dexType = "odex";
    var magic = Memory.readUtf8String(begin).replace(/\\n/g, "");
    var address = ptr(begin).add(isDex ? 0x20 : 0x1C);
    var dex_size = Memory.readInt(ptr(address));
    var dex_path = "/data/data/" + g_processName + "/" + idx + "_" + dex_size + "." + dexType;
    var dex_file = new File(dex_path, "wb");
    idx += 1;

    dex_file.write(Memory.readByteArray(begin, dex_size));
    dex_file.flush();
    dex_file.close();

    console.log("[*] magic : " + magic );
    console.log("[*] size  : " + dex_size);
    console.log("[*] dumped " + dexType + " @ " + dex_path + "\\n");
}
```

[그림 66 추출한 덱스 저장 코드]

7. 원본 덱스 파일 추출

후킹하여 획득한 덱스 파일은 에뮬레이터 내부에 저장되어있기 때문에 이를 추출하는 과정이 필요하다. 이때 덱스 파일은 /data/data/[패키지이름] 경로에 저장되어 있는데 추출 시 해당 경로에 모든 파일을 추출한다. 모든 파일을 추출하는 이유는 패킹된 앱이 실행될 때 추가로 사용하는 라이브러리나 파일이 있을 수 있기 때문에 해당 경로에 있는 모든 파일을 추출하는 것이다.

```
def extract(self):
    out_file = self.package_name+'.tar'
    out_path = '/data/local/tmp/'+out_file
    self.adb.shell('tar cvf %s -C /data/data/%s/ .' %(out_path, self.package_name))
    self.adb.pull(out_path, out_file)
    self.adb.shell('rm '+out_path)
    self.adb.uninstall(self.package_name)
```

[그림 67 원본 덱스 파일 추출 코드]

8. json 생성

마지막으로 언패킹하면서 획득한 모든 정보와 추출한 원본 덱스에서 의미있는 문자열, 해시값 등을 모두 포함한 json 파일을 추출한다. 이와 같이 추출하는 이유는 한번에 정보를 확인하기 위해서도 있지만 웹에서 쉽게 정보를 출력하기 위함도 있다.
문자열 추출 시 사용하는 알고리즘은 쿠쿠 샌드박스에서 기본으로 사용하는 알고리즘을 그대로 사용하였다.

```
def json_export(self):
    self.json_log['android_version'] = self.and_ver
    self.json_log['package_name'] = self.package_name
    self.json_log['process_name'] = self.process_name
    self.json_log['function_name'] = self.function_name
    with tarfile.open(self.package_name+'.tar') as tr:
        file_list = tr.getmembers()
        file_list_s = ', '.join(map(str, file_list))
        p = re.compile('[0-9]_[0-9]*\.[a-z]*')
        dex_list = re.findall(p, file_list_s)
        self.json_log['dex'] = {}
        for dex in dex_list:
            self.json_log['dex'][dex] = {}
            obj = re.search('[0-9]_([0-9]*)\.([a-z]*)', dex)
            self.json_log['dex'][dex]['size'] = obj.group(1)
            self.json_log['dex'][dex]['type'] = obj.group(2)
            self.json_log['dex'][dex]['hash'] = {}
            self.json_log['dex'][dex]['strings'] = []
            with tr.extractfile('./'+dex) as f:
                data = f.read()
                self.json_log['dex'][dex]['hash']['md5'] = hashlib.md5(data).hexdigest()
                self.json_log['dex'][dex]['hash']['sha1'] = hashlib.sha1(data).hexdigest()
                self.json_log['dex'][dex]['hash']['sha256'] = hashlib.sha256(data).hexdigest()
                for s in re.findall(b"[\x1f-\x7e]{6,}", data):
                    self.json_log['dex'][dex]['strings'].append(s.decode('utf-8'))
                for s in re.findall(b"(?:[\x1f-\x7e][\x00]){6,}", data):
                    self.json_log['dex'][dex]['strings'].append(s.decode('utf-16le'))
    with open(self.package_name+'_result.json', 'w') as json_file:
        json.dump(self.json_log, json_file, indent='\t')
```

[그림 68 json 생성 코드]

```
{
    "filename": "2048_bangcle.apk",
    "packer": "bangcle",
    "signature": [
        "libsecexe.so",
        "libsecmain.so",
        "bangcle_classes.jar"
    ],
    "hash": {
        "md5": "bbfcc0d68853c5a8b0aa208309d46ff6",
        "sha-1": "4ee91292f6165b4aeacdb946c3cdaa03e1ee6184",
        "sha-256": "0be1815300c4e180f3010a0165287e20ed5c625ee1a81af3d78d85058f2a6a81"
    },
    "android_version": "7.1.2",
    "package_name": "com.uberspot.a2048",
    "process_name": "com.uberspot.a2048",
    "function_name": "_ZN3art7DexFile10OpenMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEE
    "dex": {
        "1_21272.dex": {
            "size": "21272",
            "type": "dex",
            "hash": {
                "md5": "a9222b4ae903c2c4b4f0b08a23653f57",
                "sha1": "f3e3dcbc55d34e6ff571a8543b5750e3875e1313",
                "sha256": "068cbd331ce8b32988124410ffd8a6ebe85db16ab4be50f58ce0c90a4cae0cb4"
            },
            "strings": [
                " classSize",
                "/.cache",
                "/.cache/",
                "/.cache/classes.dex",
                "/.cache/classes.jar",
                "/.sec_version",
                "/data/dalvik-cache/",
                "/data/dalvik-cache/arm/",
                "/data/data/",
```


[그림 69 저장된 json 파일]

이전 과정을 모두 거치면 다음과 같은 출력 화면을 확인할 수 있다.

```
C:\Users\okko2\OneDrive\Desktop\git\Android-Unpacker-Project\unpacker>python main.py 2048_bangcle.apk
=== Unpacking Start ===
[*] Multiple devices detected, please select the device
[*] 1 : 127.0.0.1:62026
[*] 2 : 127.0.0.1:62025
> 2
[*] ADB connected : 127.0.0.1:62025
[*] 2048_bangcle.apk install success
[*] Frida attached : com.uberspot.a2048(5436)
[*] Android version : 7.1.2
[*] Function name : _ZN3art7DexFile10OpenMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEENS3_9allocatorIcE
EEEjPNS_6MemMapEPKNS_10OatDexFileEPS9_
[*] Process name : com.uberspot.a2048
[*] Dump dex start
[*] magic : dex035
[*] size  : 21272
[*] dumped dex @ /data/data/com.uberspot.a2048/1_21272.dex
```

[그림 70 언패킹 출력 결과]

3.2.2 패커 탐지

기존에 개발된 APKiD의 Yara 스크립트를 이용하여 Yara 도구로 분석할 경우, 일부 패커는 결과가 중복으로 나와 어떠한 패커가 사용됐는지 확인할 수 없다. 또한 Yara 도구는 탐지된 시그니처를 출력해주는 옵션이 존재하지만, 중복 제거 옵션이 존재하지 않아 출력 값을 확인할 때 매우 불편함이 존재한다. 따라서 Yara의 Python 모듈을 이용하여 일부 패커의 중복 탐지와 탐지된 시그니처의 중복 제거의 문제점을 개선한 Yara-Rules 개발을 진행하였다.

1. APK 파일 탐지

Yara 모듈을 불러오고 파일 이름으로 APK 파일명을 입력받는다. 이어서 Yara-Rules를 통해 해당 파일이 APK 파일인지 아닌지 확인한다. 여기서 strings는 탐지되는 시그니처를 의미하며, 문자열이나 Hex 값을 입력한다. 그러므로 APK 파일 여부를 확인하기 위해서 파일 내부에 문자열로 "PK"와 "AndroidManifest"가 존재하게 되면 APK 파일이라는 것을 알 수 있다. 그리고 condition은 strings를 통해 참인지 거짓인지 판별하게 되고, 참이면 정상적인 APK 파일을 출력하고, 거짓이면 올바르지 않은 APK 파일로 출력된다.

```
import yara
import hashlib

filename = input("APK 파일명을 입력해주세요.\n")

# apk 파일 여부
is_apk = yara.compile(
    source='rule is_apk \
    {strings:$zip_head = "PK" $manifest = "AndroidManifest.xml" \
    condition: $zip_head at 0 and $manifest and #manifest >= 2}'
)
apk_match = is_apk.match(filename)
if len(apk_match) == 1:
    print("정상적인 apk 파일입니다.\n")
else:
    print("올바르지 않은 apk 파일입니다.\n")
```

[그림 71 APK 파일 탐지 코드]

2. 패커 탐지

다음으로 패커에 대한 Yara-Rules이다. 위 그림은 Alibaba라는 패커를 예시로 하였으며, strings를 통해 해당 문자열 값이 존재하게 되면 condition으로 판별하여 Alibaba 패커라는 것을 확인할 수 있다. 그리고 이외 다른 패커들도 위 그림과 동일하게 strings를 통해 문자열(시그니처)을 탐지하고, 탐지된 문자열(시그니처)을 기반으로 condition에서 판별하여 어떠한 패커가 사용되었는지 출력하므로 다른 패커들의 패커 탐지 코드는 생략한다. 그리고 일부 패커의 경우 중복으로 탐지가 되어 패커 판별이 불가능했던 문제점을 개선하여 모든 패커가 하나의 패커만 탐지되도록 개발하였다.

```
# alibaba packer
alibaba = yara.compile(
    source='rule alibaba \
    {strings:$lib = "libmobisec.so" \
    condition: $lib}'
)
alibaba_match = alibaba.match(filename)
```

[그림 72 패커 탐지 코드]

3. 탐지된 시그니처 중복 제거

모든 패커는 탐지된 시그니처의 출력 값을 확인해보면 중복으로 탐지되어 매우 불편함이 존재하게 된다. 따라서 편의성 제공을 위해 Python 반복문을 사용하여 탐지된 시그니처의 중복을 제거하도록 개발하였다. 위 그림은 Alibaba 패커를 예시로 하여 탐지된 시그니처를 중복 제거하는 코드이며, 이외 다른 패커들의 중복 제거도 동일하게 개발되었으므로 다른 패커들의 중복 제거 코드는 생략한다.

```
if len(apk_match) == 1 and len(alibaba_match) == 1:
    print(f"{filename} 패커는 alibaba 패커입니다.\n")
    print(f"{filename}의 탐지된 시그니처")
    signature = alibaba_match[0].strings
    signature_list = []
    for x, y, z in signature:
        if z not in signature_list:
            signature_list.append(z)
    for i in signature_list:
        result = i.decode(encoding="utf-8")
        print(result)
    print()
```

[그림 73 탐지된 시그니처 중복 제거 코드]

4. 해시값 산출

마지막으로 악성 앱 식별을 위해 해시 값을 산출한다. 해시는 주로 많이 사용되는 MD5와 SHA-1, SHA-256 해시를 선정하여 총 3가지의 해시 값을 출력하게 된다.

```python
# apk 파일 해시 출력
if len(apk_match) == 1:
    apk_file = open(filename, "rb")
    apk_data = apk_file.read()
    apk_file.close()
    print(f"{filename} 파일의 해시 값 출력 (MD5, SHA-1, SHA-256)")
    print("MD5: " + hashlib.md5(apk_data).hexdigest())
    print("SHA-1: " + hashlib.sha1(apk_data).hexdigest())
    print("SHA-256: " + hashlib.sha256(apk_data).hexdigest())
```

[그림 74 APK 파일 해시 출력 코드]

이전 과정을 모두 거치면 다음과 같은 출력 화면을 확인할 수 있다.

```
D:\Yara-Unpacker>python Yara-Rules_APK.py
APK 파일명을 입력해주세요.
2048_ali.apk
정상적인 apk 파일입니다.

2048_ali.apk 패커는 alibaba 패커입니다.

2048_ali.apk의 탐지된 시그니처
libmobisec.so

2048_ali.apk 파일의 해시 값 출력 (MD5, SHA-1, SHA-256)
MD5: 652bda9f7b8155f413577520e0bda6fc
SHA-1: 4ccd3909dba290789d338035b8400d1d4dbe8e9a
SHA-256: 9e529c4e3402b208f955b1fb6fddd493816fccc48ac1981fd0a1aed35d85acb3
```

[그림 75 Yara 출력 결과]

4. 결론

4.1 결론

오픈소스로 공개된 Native Unpacker 와 Frida Unpacker 를 분석하여 원본 Dex 파일을 추출하는 과정 및 언패커 결과를 확인할 수 있었다. 언패커 개발은 결과를 통해 확인할 수 있듯이 성능이 우수했던 Frida Unpacker 를 기준으로 개발을 진행하였다. 또한 패커 식별을 위하여 오픈소스로 공개된 Yara 시그니처를 응용하여 각 패커마다 사용되는 라이브러리나 리소스 파일을 식별하고 어떠한 패커가 사용되었는지 식별할 수 있었다. 최종적으로 악성 앱 파일을 업로드하면 사용된 패커 식별과 탐지된 라이브러리 및 리소스 확인 후 원본 Dex 파일 추출을 진행한다. 마지막으로 악성 앱을 식별할 수 있도록 MD5, SHA-1, SHA-256 해시값을 산출하는 도구를 개발할 수 있었다.

```
C:\Users\okko2\OneDrive\Desktop\git\Android-Unpacker-Project\unpacker>python main.py 2048_bangcle.apk
정상적인 apk 파일입니다.

2048_bangcle.apk 패커는 bangcle 패커입니다.

=== 2048_bangcle.apk의 탐지된 시그니처 ===
libsecexe.so
libsecmain.so
bangcle_classes.jar

2048_bangcle.apk 파일의 해시 값 출력 (MD5, SHA-1, SHA-256)
MD5: bbfcc0d68853c5a8b0aa208309d46ff6
SHA-1: 4ee91292f6165b4aeacdb946c3cdaa03e1ee6184
SHA-256: 0be1815300c4e180f3010a0165287e20ed5c625ee1a81af3d78d85058f2a6a81

=== Unpacking Start ===
[*] Multiple devices detected, please select the device
[*] 1 : 127.0.0.1:62026
[*] 2 : 127.0.0.1:62025
> 2
[*] ADB connected : 127.0.0.1:62025
[*] 2048_bangcle.apk install success
[*] Frida attached : com.uberspot.a2048(5436)
[*] Android version : 7.1.2
[*] Function name : _ZN3art7DexFile10OpenMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEENS3_9allocatorIcE
EEEjPNS_6MemMapEPKNS_10OatDexFileEPS9_
[*] Process name : com.uberspot.a2048
[*] Dump dex start
[*] magic : dex035
[*] size  : 21272
[*] dumped dex @ /data/data/com.uberspot.a2048/1_21272.dex
```

[그림 76 최종 출력 결과]

4.2 기대효과

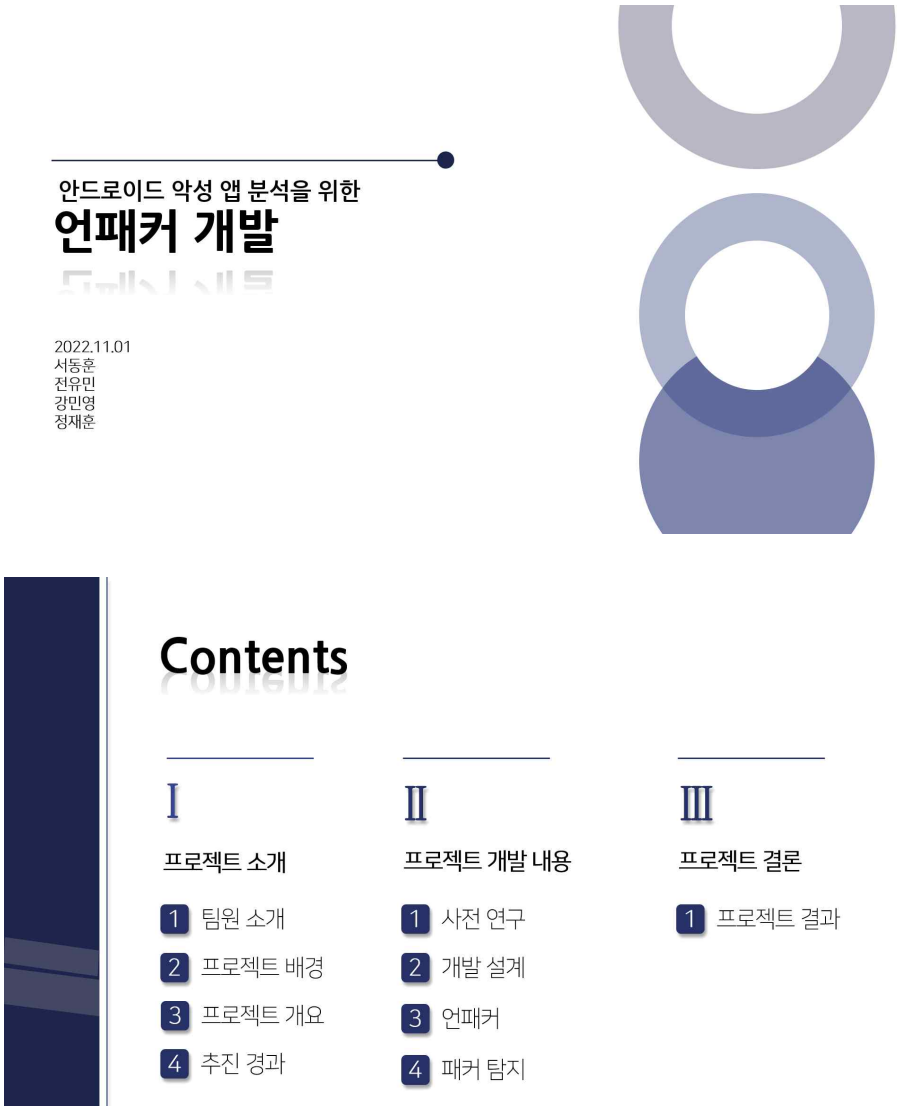
안드로이드 환경에서의 악성 앱은 주로 패킹을 통해 탐지를 우회하거나 분석을 어렵게 한다. 이러한 문점을 해결하기위해 언패킹 도구를 개발하였으며, 사용자는 이 도구를 사용하여 사용한 패커의 종류, 원본 덱스, 의심되는 문자열, 해시 값 등 다양한 정보를 획득할 수 있다. 최종적으로 분석가들이 패킹된 악성 앱을 분석할 때 편리함과 여러 정보 제공을 통해 분석 시 소요하는 시간을 줄일 수 있다.

5. 별첨

5.1 소스 코드

https://github.com/le3mon/Android-Unpacker-Project

5.2 발표 자료



I 프로젝트 소개

1 팀원 소개
2 프로젝트 배경
3 프로젝트 개요
4 추진 경과

팀원 소개

✓ 저희 팀원을 소개합니다!

|서 동 훈
총괄
악성 앱 분석 및 개발

|전 유 민
언패커 분석 및 Yara 개발

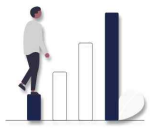
|정 재 훈
언패커 분석 및 웹 개발

|강 민 영
언패커 분석 및 웹 개발

프로젝트 배경

✓ 프로젝트 추진 배경 및 필요성과 목적

프로젝트 추진 배경 및 필요성



스마트폰 사용량 급증에 따른 피해 증가

스마트폰 사용량이 지속적으로 증가함에 따라 이에 따른 피해도 점차 늘어나고 있습니다.

공격자의 안드로이드 악성 APK 파일 전파

공격자는 피싱, 스미싱, 몸캠 피싱 등과 같은 공격을 주로 안드로이드 악성 APK 파일을 이용하여 공격합니다.

1 2
3 목적

안드로이드 악성 APK 파일 탐지 연구 불충분

이와 관련된 도구나 연구가 국내에서는 활발히 이루어지지 않고 있습니다.

안드로이드 언패커 제작

이러한 문제를 해결하기 위해 안드로이드 언패커 개발을 진행합니다.

프로젝트 배경

✓ 프로젝트 추진 배경 및 필요성과 목적

프로젝트 추진 배경 및 필요성

스마트폰 사용량 급증에 따른 피해 증가

스마트폰 사용량이 지속적으로 증가함에 따라 이에 따른 피해도 점차 늘어나고 있습니다.

1

연도	피싱	스미싱	몸캠피싱
2017	545	667	1234
2018	1978	293	1406
2019	2874	207	1824
2020	1519	822	2583

▲피싱, 스미싱, 몸캠피싱 사건 증가량

▲스마트폰 사용률

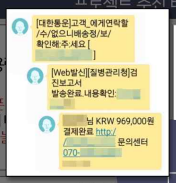
이러한 문제를 해결하기 위해 안드로이드 언패커 개발을 진행합니다.

프로젝트 배경

✓ 프로젝트 추진 배경 및 필요성과 목적

프로젝트 추진 배경 및 필요성

▲질병관리청 사칭



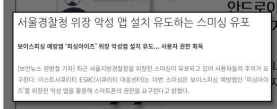
▲택배 사칭

2

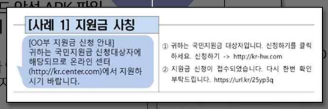
공격자의 안드로이드 악성 APK 파일 전파

공격자는 피싱, 스미싱, 몸캠 피싱 등과 같은 공격을 주로 안드로이드 악성 APK 파일을 이용하여 공격합니다.

3 목적



▲경찰청 위장 앱



▲지원금 사칭

안드로이드 언패커 제작

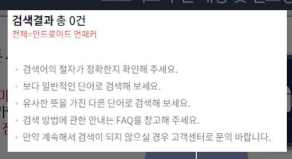
이러한 문제를 해결하기 위해 안드로이드 언패커 개발을 진행합니다.

프로젝트 배경

✓ 프로젝트 추진 배경 및 필요성과 목적

프로젝트 추진 배경 및 필요성

스마트폰 ... 전파



3 목적 ▲DBpia 검색결과

안드로이드 악성 APK 파일 탐지 연구 불충분

이와 관련된 도구나 연구가 국내에서는 활발히 이루어지지 않고 있습니다.

안드로이드 언패커 제작

이러한 문제를 해결하기 위해 안드로이드 언패커 개발을 진행합니다.

프로젝트 배경

✓ 프로젝트 추진 배경 및 필요성과 목적

프로젝트 추진 배경 및 필요성

스마트폰 사용량 급증에 따른 피해 증가

스마트폰 사용량이 지속적으로 증가함에 따라 이에 따른 피해도 점차 늘어나고 있습니다.

공격자의 안드로이드 악성 APK 파일 전파

공격자는 피싱, 스미싱, 몸캠 피싱 등과 같은 공격을 주로 안드로이드 악성 APK 파일을 이용하여 공격합니다.

1 2 3 목적

안드로이드 악성 APK 파일 탐지 연구 불충분

이와 관련된 도구나 연구가 국내에서는 활발히 이루어지지 않고 있습니다.

안드로이드 언패커 제작

이러한 문제를 해결하기 위해 안드로이드 언패커 개발을 진행합니다.

프로젝트 개요

✓ 안드로이드 언패커 제작 개요

안드로이드 언패커 제작

분석가의 실무 능력 향상

국내에서 활발히 이루어지지 않는 패킹 기법에 관한 연구 및 방법론 제시

모바일 금융범죄인 피싱, 스미싱, 몸캠 피싱에 적극적인 대응

추진 경과

✓ 프로젝트 추진 경과

진행 예정
진행중
완료

구분	추진 내용	4월	5월	6월	7월	8월	9월	10월	11월
계획	사전 자료 조사	■							
분석	기존 언패커 도구 분석	■	■	■					
	악성 패킹 앱 분석			■	■				
개발	언패커 도구 개발				■	■	■		
테스트	패킹된 악성 앱 대상 테스트 진행						■	■	
마무리	논문 및 보고서 작성							■	■
	졸업작품 발표회 및 캡스톤 공모전 준비							■	■

Ⅱ 프로젝트 개발 내용

1 사전 연구
2 개발 설계
3 언패커
4 패커 탐지

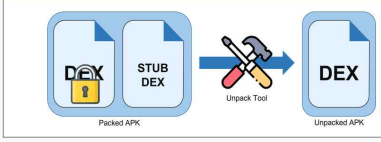
사전 연구

✓ 개발에 앞서 사전 연구 진행

패킹과 언패킹

- 패킹이란 원본 Dex 파일을 보호하기 위해 사용하는 방법으로 주로 Dex 파일 은닉, Dex 파일 덤핑 방지, 안티 리버싱 등 사용
- 언패킹이란 패킹의 반대되는 개념으로, 패킹했던 파일의 패킹을 푸는 것

언패킹 과정



Native 언패커

- Defcon 22에서 공개된 언패커
- 프로세스와 스레드 메모리에 접근 후, 시그니처를 통해 패커의 종류를 식별하고, Dex 파일의 Magic Number를 사용하여 Dex 파일 추출

Frida 언패커

- Frida 도구를 사용하여 제작한 언패커
- 안드로이드에서 Dex 파일을 파싱할 때 사용하는 함수를 후킹한 후 해당 함수로 넘어오는 Dex 파일 추출

15 패커 종류	원본 Dex 추출 여부	16 패커 종류	원본 Dex 추출 여부	18 패커 종류	원본 Dex 추출 여부	19 패커 종류	원본 Dex 추출 여부
15_Ali	O	16_Ali	X				
15_Baidu	X	16_Baidu	O	18_Baidu	O	19_Baidu	X
15_Bangcle	X	16_Bangcle	X	18_Bangcle	X	19_Bangcle	X
15_Ijiami	O	16_Ijiami	X	18_Ijiami	O	19_Ijiami	X
15_Qihoo	O	16_Qihoo	O	18_Qihoo	O	19_Qihoo	X
15_Tencent	X	16_Tencent	X	18_Tencent	X	19_Tencent	X

▲ Native 언패커 결과

- 언패킹이란 패킹의 반대되는 개념으로, 패킹했던 파일의 패킹을 푸는 것

안드로이드 기기에 저장한 후 분석할
앱을 실행하여 실행 파일의 일부로

15 패커 종류	원본 Dex 추출 여부	16 패커 종류	원본 Dex 추출 여부	18 패커 종류	원본 Dex 추출 여부	19 패커 종류	원본 Dex 추출 여부
15_Ali	O	16_Ali	O				
15_Baidu	O	16_Baidu	O	18_Baidu	O	19_Baidu	O
15_Bangcle	O	16_Bangcle	O	18_Bangcle	O	19_Bangcle	O
15_Ijiami	O	16_Ijiami	O	18_Ijiami	X	19_Ijiami	X
15_Qihoo	O	16_Qihoo	O	18_Qihoo	X	19_Qihoo	X
15_Tencent	O	16_Tencent	O	18_Tencent	X	19_Tencent	X

▲ Frida 언패커 결과

Ⅱ 프로젝트 개발 내용

개발 설계

✓ 사전 연구를 바탕으로 언패커 개발 설계

개발 설계

개발 목표

① 원본 Dex File 추출

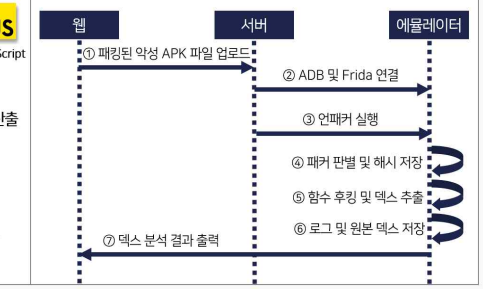
② Yara 시그니처를 응용한 **패커 식별**

③ 악성 앱 식별을 위한 MD5, SHA-1, SHA-256 **해시 값 산출**

Python JavaScript

웹 사이트에 **악성 앱 파일 업로드** 후, 사용된 **패커 식별**과
탐지된 라이브러리 및 리소스 확인 후 **원본 Dex file 추출**

전체 구상도



Ⅱ 프로젝트 개발 내용

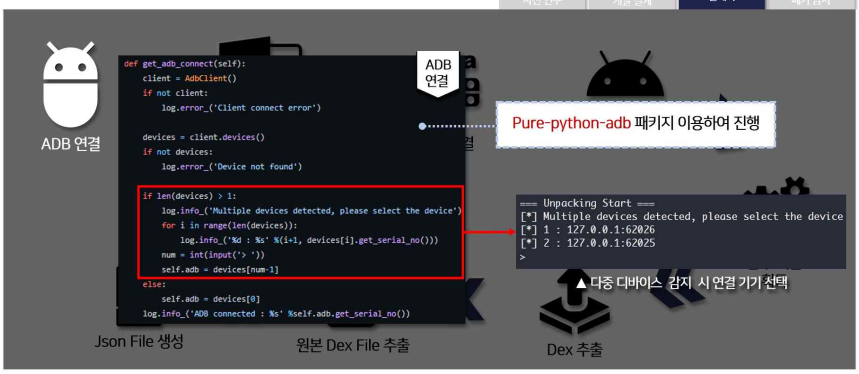
언패커 개발

✓ 기존 언패커 분석 후 개발 진행

Ⅱ 프로젝트 개발 내용

언패커 개발

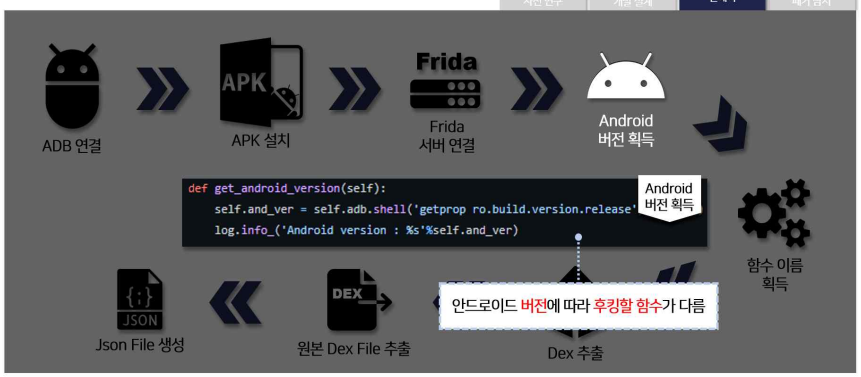
✓ 기존 언패커 분석 후 개발 진행



언패커 개발 ✓ 기존 언패커 분석 후 개발 진행

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행



Ⅱ 프로젝트 개발 내용

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행


```
get_function = """
rpc.exports = {
    getFunc: function (and_ver) {
        var function_name = '';
        var i = 0;
        var android_version = parseInt(and_ver);
        if (android_version > 4) {
            var lib_name = android_version >= 10 ? 'libdexfile.so' : 'libart.so'
            var art = Module.enumerateExportsSync(lib_name);

            for(i = 0; i< art.length; i++){
                if(art[i].name.indexOf("OpenMemory") !== -1){
                    function_name = art[i].name;
                    return function_name;
                }else if(art[i].name.indexOf("OpenCommon") !== -1){
                    if (android_version >= 10 && art[i].name.indexOf("ArtDexFileLoader") !== -1)
                        continue;
                    function_name = art[i].name;
                    return function_name;
```


Ⅱ 프로젝트 개발 내용

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행


```
get_function = """
rpc.exports = {
    getFunc: function (and_ver) {
        var function_name = '';
        var i = 0;
        var android_version = parseInt(and_ver);
        if (android_version > 4) {
            var lib_name
            var art = Mo
                                def get_function_name(self):
                                    script = self.session.create_script(get_function)
                                    script.load()
                                    api = script.exports
            for(i = 0; i        self.function_name = api.get_func(self.and_ver)
                if(art[i        log.info_('Function name : %s' %self.function_name)
                    func
                return function_name;
            }else if(art[i].name.indexOf("OpenCommon") !== -1){
                if (android_version >= 10 && art[i].name.indexOf("ArtDexFileLoader") !== -1)
                    continue;
                function_name = art[i].name;
                return function_name;
```


Ⅱ 프로젝트 개발 내용

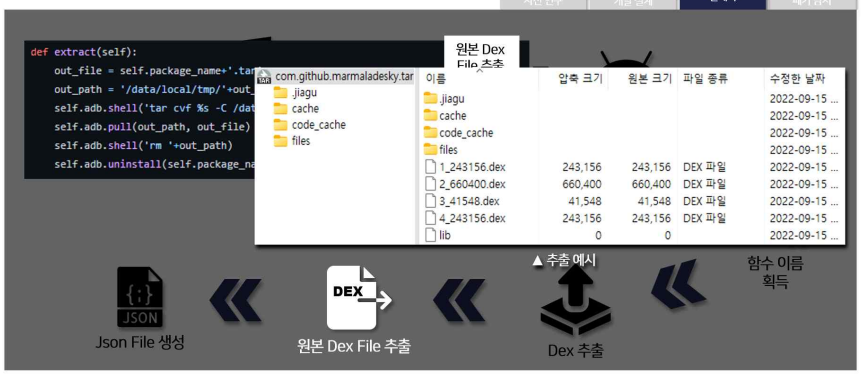
언패커 개발 ✓ 기존 언패커 분석 후 개발 진행

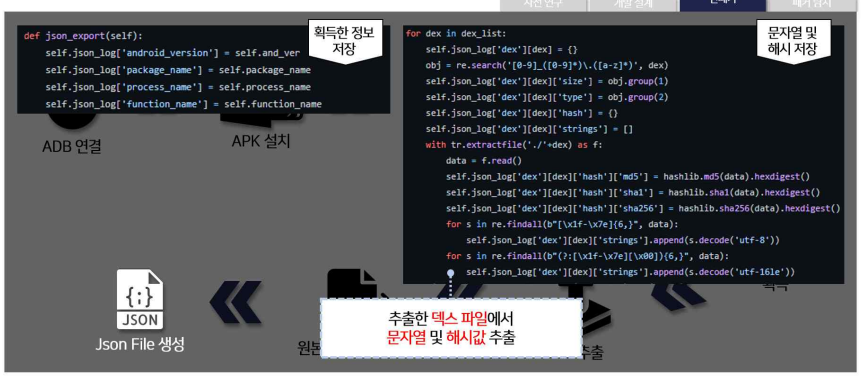

```
rpc.exports = {
    hook: function (func, proc, ver) {
        var moduleFuncName = func;
        var g_processName = proc;
        var g_AndroidOSVersion = parseInt(ver);
        console.log("[*] Dump dex start");

        if(moduleFuncName !== ""){
            var hookFunction;
            if (g_AndroidOSVersion > 4) {
                hookFunction = Module.findExportByName("libart.so", moduleFuncName);
            } else {
                hookFunction = Module.findExportByName("libdvm.so", moduleFuncName);
                if(hookFunction == null) {
                    hookFunction = Module.findExportByName("libart.so", moduleFuncName);
                }
            }
            Interceptor.attach(hookFunction,{
                onEnter: function(args){
                    var begin = 0;
                    var dexMagicMatch = false;
                    var odexMagicMatch = false;
```

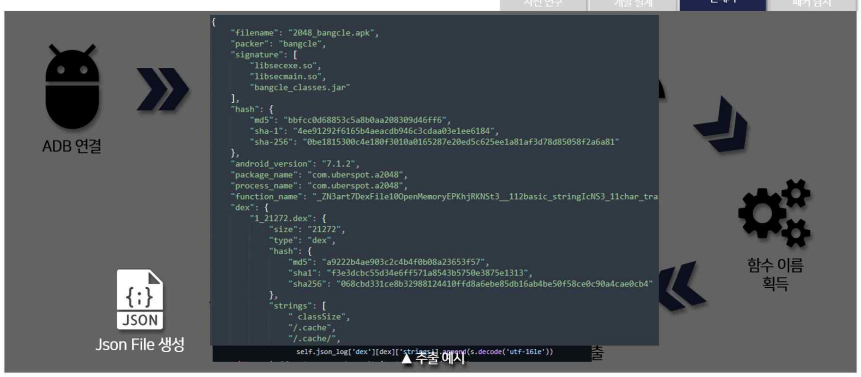

Ⅱ
프로젝트 개발 내용
언패커 개발
기존 언패커 분석 후 개발 진행
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 닫지
Interceptor.attach(hookFunction,{
onEnter: function(args){
var begin = 0;
var dexMagicMatch = false;
var odexMagicMatch = false;
dexMagicMatch = checkDexMagic(args[0]);
if (dexMagicMatch === true){
begin = args[0];
함수 후킹
function checkDexMagic(dataAddr) {
var magicMatch = true;
var magicFlagHex = [0x64, 0x65, 0x78, 0x0a, 0x30, 0x33, 0x35, 0x00];
for(var i = 0; i < 8; i++){
if(Memory.readU8(ptr(dataAddr).add(i)) !== magicFlagHex[i]){
magicMatch = false;
break;
}
}
return magicMatch;
}
덱스 탐색
함수 이름 획득
JSON
Json File 생성
후킹 후 덱스 매직 넘버 값을 통해 덱스 탐색
원본 Dex File 추출
Dex 추출


Ⅱ
프로젝트 개발 내용
언패커 개발
기존 언패커 분석 후 개발 진행
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 닫지
function dumpDexToFile(isDex, begin, g_processName) {
var dexType;
isDex ? dexType = "dex" : dexType = "odex";
var magic = Memory.readUtf8String(begin).replace(/\\n/g, "");
var address = ptr(begin).add(isDex ? 0x20 : 0x1C);
var dex_size = Memory.readInt(ptr(address));
var dex_path = "/data/data/" + g_processName + "/" + idx + "_" + dex_size + "." + dexType;
var dex_file = new File(dex_path, "wb");
idx += 1;
dex_file.write(Memory.readByteArray(begin, dex_size));
dex_file.flush();
dex_file.close();
console.log("[*] magic : " + magic);
console.log("[*] size : " + dex_size);
console.log("[*] dumped " + dexType + " @ " + dex_path + "\\n");
}
Dex 추출
Android 버전 획득
발견한 덱스는 "/data/data/[패키지 이름]" 경로에 저장
함수 이름 획득
Json File 생성
원본 Dex File 추출
Dex 추출

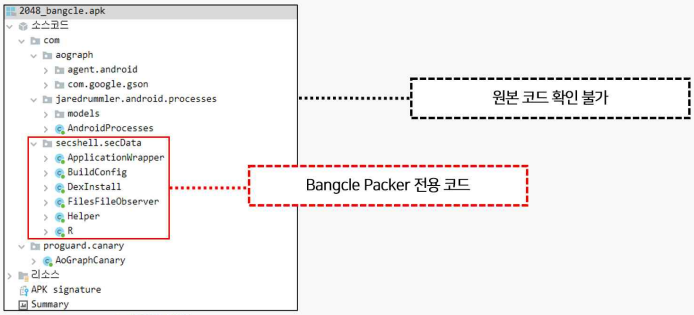

Ⅱ
프로젝트 개발 내용
언패커 개발
기존 언패커 분석 후 개발 진행
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 닫지
def extract(self):
out_file = self.package_name+'.tar'
out_path = '/data/local/tmp/'+out_file
self.adb.shell('tar cvf %s -C /data/data/%s/ .' %(out_path, self.package_name))
self.adb.pull(out_path, out_file)
self.adb.shell('rm '+out_path)
self.adb.uninstall(self.package_name)
원본 Dex File 추출
Android 버전 획득
추출한 덱스를 포함한 "/data/data/[패키지 이름]" 경로에 모든 파일 추출
함수 이름 획득
JSON
Json File 생성
DEX
원본 Dex File 추출
Dex 추출


def extract(self):
out_file = self.package_name+'.tar
out_path = '/data/local/tmp/'+out_
self.adb.shell('tar cvf %s -C /dat
self.adb.pull(out_path, out_file)
self.adb.shell('rm '+out_path)
self.adb.uninstall(self.package_na
원본 Dex File 추출
com.github.marmaladesky.tar
.jiagu
cache
code_cache
files
이름 압축 크기 원본 크기 파일 종류 수정한 날짜
.jiagu 2022-09-15 ...
cache 2022-09-15 ...
code_cache 2022-09-15 ...
files 2022-09-15 ...
1_243156.dex 243,156 243,156 DEX 파일 2022-09-15 ...
2_660400.dex 660,400 660,400 DEX 파일 2022-09-15 ...
3_41548.dex 41,548 41,548 DEX 파일 2022-09-15 ...
4_243156.dex 243,156 243,156 DEX 파일 2022-09-15 ...
lib 0 0 2022-09-15 ...
▲ 추출 예시
함수 이름 획득
JSON
Json File 생성
DEX
원본 Dex File 추출
Dex 추출


def json_export(self):
self.json_log['android_version'] = self.and_ver
self.json_log['package_name'] = self.package_name
self.json_log['process_name'] = self.process_name
self.json_log['function_name'] = self.function_name
획득한 정보 저장
ADB 연결
APK 설치
for dex in dex_list:
self.json_log['dex'][dex] = {}
obj = re.search('[0-9]_([0-9]*)\.([a-z]*)', dex)
self.json_log['dex'][dex]['size'] = obj.group(1)
self.json_log['dex'][dex]['type'] = obj.group(2)
self.json_log['dex'][dex]['hash'] = {}
self.json_log['dex'][dex]['strings'] = []
with tr.extractfile('./'+dex) as f:
data = f.read()
self.json_log['dex'][dex]['hash']['md5'] = hashlib.md5(data).hexdigest()
self.json_log['dex'][dex]['hash']['sha1'] = hashlib.sha1(data).hexdigest()
self.json_log['dex'][dex]['hash']['sha256'] = hashlib.sha256(data).hexdigest()
for s in re.findall(b"[\x1f-\x7e]{6,}", data):
self.json_log['dex'][dex]['strings'].append(s.decode('utf-8'))
for s in re.findall(b"(?:[\x1f-\x7e][\x00]){6,}", data):
self.json_log['dex'][dex]['strings'].append(s.decode('utf-16le'))
문자열 및 헤시 저장
JSON
Json File 생성
추출한 덱스 파일에서
문자열 및 헤시값 추출

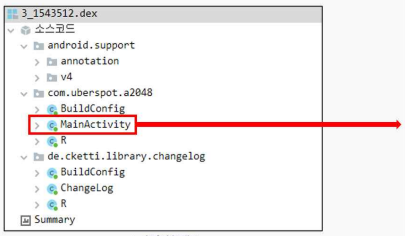

{
"filename": "2048_bangcle.apk",
"packer": "bangcle",
"signature": [
"libsecexe.so",
"libsecmain.so",
"bangcle_classes.jar"
],
"hash": {
"md5": "bbfcc0d68853c5a8b0aa208309d46ff6",
"sha-1": "4ee91292f6165b4aeacdb946c3cdaa03e1ee6184",
"sha-256": "0be1815300c4e180f3010a0165287e20ed5c625ee1a81af3d78d85058f2a6a81"
},
"android_version": "7.1.2",
"package_name": "com.uberspot.a2048",
"process_name": "com.uberspot.a2048",
"function_name": "_ZN3art7DexFile10OpenMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_tra
"dex": {
"1_21272.dex": {
"size": "21272",
"type": "dex",
"hash": {
"md5": "a9222b4ae903c2c4b4f0b08a23653f57",
"sha1": "f3e3dcbc55d34e6ff571a8543b5750e3875e1313",
"sha256": "068cbd331ce8b329081244410ffd8a6e0e85db16ab4be50f58ce0c90a4cae0cb4"
},
"strings": [
" class5ize",
"/.cache",
"/.cache/",
ADB 연결
JSON
Json File 생성
▲ 추출 예시
함수 이름 획득

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행



▲ 패킹된 APK

언패커 개발 ✓ 기존 언패커 분석 후 개발 진행



▲ 추출한 덱스

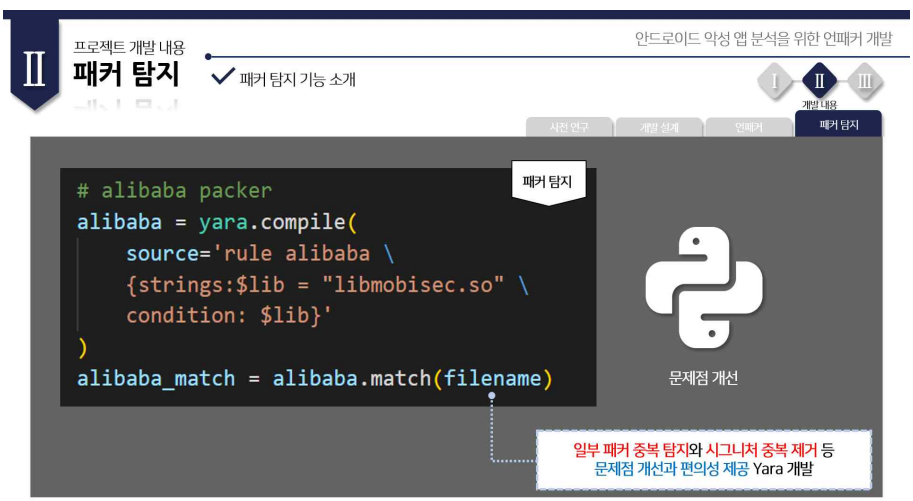
▲ MainActivity

패커 탐지 ✓ 패커 탐지 기능 소개

기존 개발된 Yara 도구 » 문제점 분석 » 문제점 개선


Ⅱ
프로젝트 개발 내용
패커 탐지
패커 탐지 기능 소개
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 탐지
기존 개발된
Yara 도구
rule alibaba : packer
{
meta:
description = "Alibaba"
strings:
$lib = "libmobisec.so"
condition:
is_apk and $lib
}
APKiD
Yara-Rules
meta : APK에 관한 정보, 해시 등 코멘트
strings : 탐지되는 시그니쳐 (문자열 or Hex)
condition : 조건의 참/거짓 판별


Ⅱ
프로젝트 개발 내용
패커 탐지
패커 탐지 기능 소개
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 탐지
is_apk 2048_ali.apk
0x0:$zip_head: PK
0x51e40:$manifest: AndroidManifest.xml
0xb091d:$manifest: AndroidManifest.xml
tencent 2048_ali.apk
0xbb1d9:$zip_lib: lib/armeabi/libmobisecy.so
0xb0746:$zip_lib: lib/armeabi/libmobisecy.so
alibaba 2048_ali.apk
0x56eb8:$lib: libmobisec.so
0xb070b:$lib: libmobisec.so
패커
중복 탐지
문제점 분석
시그니쳐
중복 출력
일부 패커는 중복 탐지로 판별 불가
탐지된 시그니쳐 중복 제거 출력 옵션 없음


Ⅱ
프로젝트 개발 내용
패커 탐지
패커 탐지 기능 소개
안드로이드 악성 앱 분석을 위한 언패커 개발
Ⅰ Ⅱ Ⅲ
개발 내용
사전 연구
개발 설계
언패커
패커 탐지
# alibaba packer
alibaba = yara.compile(
source='rule alibaba \
{strings:$lib = "libmobisec.so" \
condition: $lib}'
)
alibaba_match = alibaba.match(filename)
패커 탐지
문제점 개선
일부 패커 중복 탐지와 시그니쳐 중복 제거 등
문제점 개선과 편의성 제공 Yara 개발

시그니처 중복 제거

```python
if len(apk_match) == 1 and len(alibaba_match) == 1:
    print(f"{filename} 패커는 alibaba 패커입니다.\n")
    print(f"{filename}의 탐지된 시그니쳐")
    signature = alibaba_match[0].strings
    signature_list = []
    for x, y, z in signature:
        if z not in signature_list:
            signature_list.append(z)
    for i in signature_list:
        result = i.decode(encoding="utf-8")
        print(result)
```

문제점 개선

탐지되는 문자열 혹은 Hex의 시그니쳐 중복 제거

해시값 출력

```python
# apk 파일 해시 출력
if len(apk_match) == 1:
    apk_file = open(filename, "rb")
    apk_data = apk_file.read()
    apk_file.close()
    print(f"{filename} 파일의 해시 값 출력 (MD5, SHA-1, SHA-256)")
    print("MD5: " + hashlib.md5(apk_data).hexdigest())
    print("SHA-1: " + hashlib.sha1(apk_data).hexdigest())
    print("SHA-256: " + hashlib.sha256(apk_data).hexdigest())
```

문제점 개선

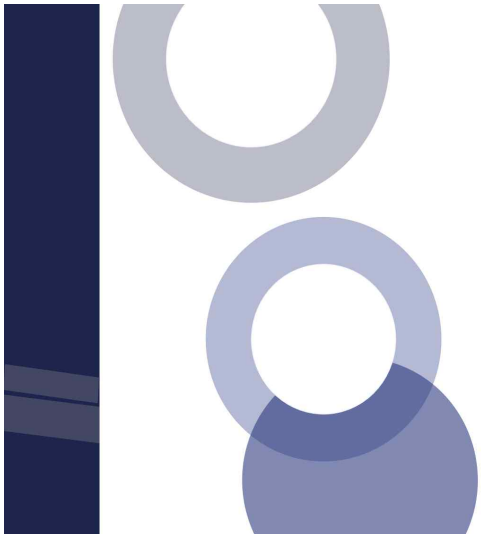
APK 파일의 MD5 / SHA-1 / SHA-256 해시값 출력

최종 출력 결과

```
D:\Yara-Unpacker>python Yara-Rules_APK.py
APK 파일명을 입력해주세요.
2048_ali.apk
정상적인 apk 파일입니다.

2048_ali.apk 패커는 alibaba 패커입니다.

2048_ali.apk의 탐지된 시그니쳐
libmobisec.so

2048_ali.apk 파일의 해시 값 출력 (MD5, SHA-1, SHA-256)
MD5: 652bda9f7b8155f413577520e0bda6fc
SHA-1: 4ccd3909dba290789d338035b8400d1d4dbe8e9a
SHA-256: 9e529c4e3402b208f955b1fb6fddd493816fccc48ac1981fd0a1aed35d85acb3
```

문제점 개선

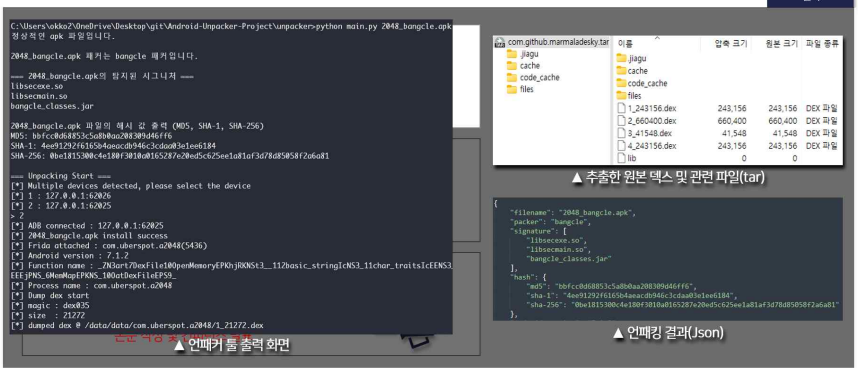
APK 판별 및 탐지된 시그니쳐 기반 패커 판별
악성 앱 분석을 위한 해시 값 산출



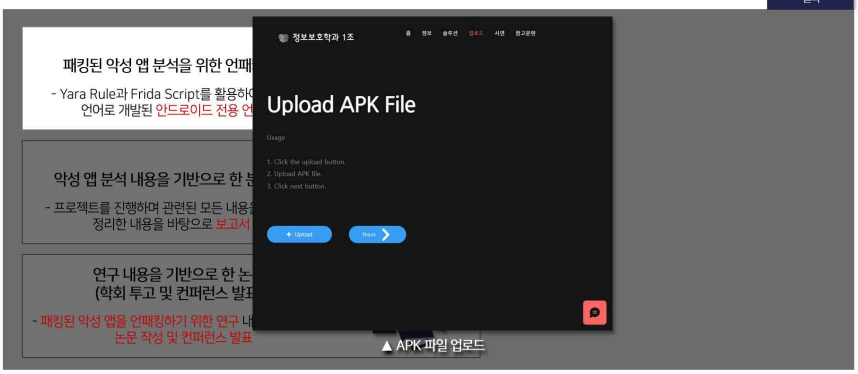
Ⅲ 프로젝트 결론

1 프로젝트 결과

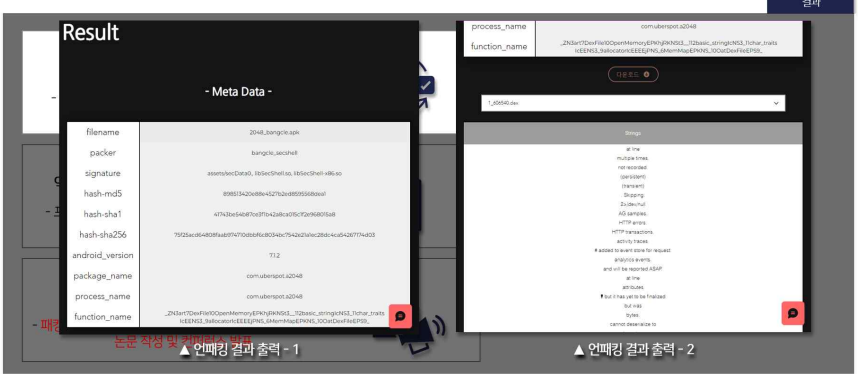
✓ 프로젝트 산출물 및 최종 결론



✓ 프로젝트 산출물 및 최종 결론



✓ 프로젝트 산출물 및 최종 결론



Ⅲ 프로젝트 결론
결과
안드로이드 악성 앱 분석을 위한 언패커 개발

✓ 프로젝트 산출물 및 최종 결론

Ⅰ Ⅱ Ⅲ 결론
결과

패킹된 악성 앱 분석을 위한 언패커 개발
- Yara Rule과 Frida Script를 활용하여 Python 언어로 개발된 안드로이드 전용 언패커

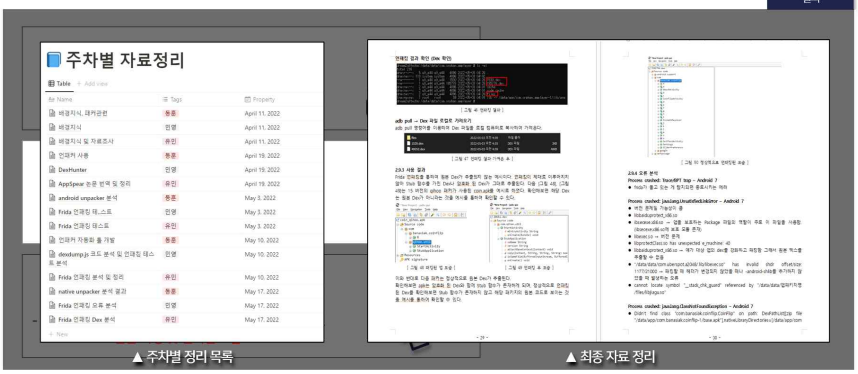
악성 앱 분석 내용을 기반으로 한 분석 보고서
- 프로젝트를 진행하며 관련된 모든 내용을 노션에 정리
정리한 내용을 바탕으로 보고서 제작

연구 내용을 기반으로 한 논문 (학회 투고 및 컨퍼런스 발표)
- 패킹된 악성 앱을 언패킹하기 위한 연구 내용을 바탕으로 논문 작성 및 컨퍼런스 발표

Ⅲ 프로젝트 결론
결과
안드로이드 악성 앱 분석을 위한 언패커 개발

✓ 프로젝트 산출물 및 최종 결론

Ⅰ Ⅱ Ⅲ 결론
결과



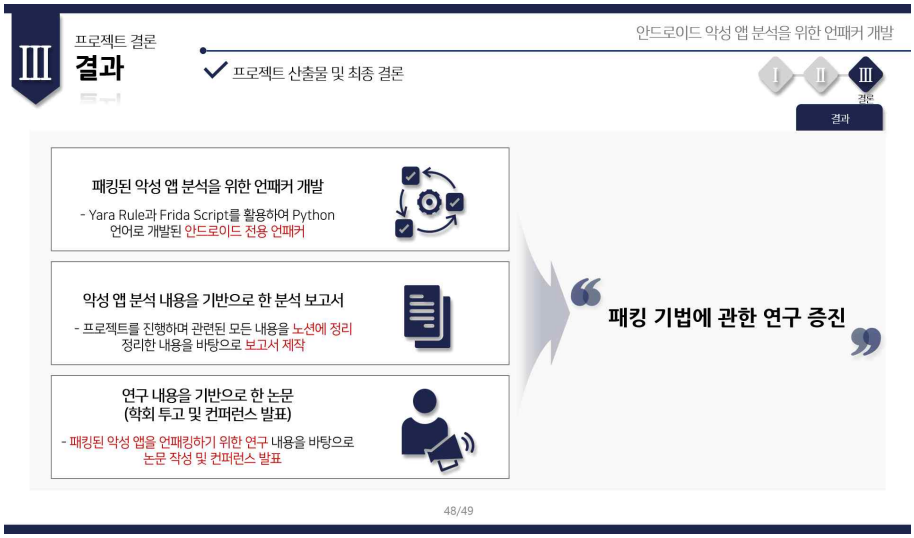
▲ 주차별 정리 목록 ▲ 최종 자료 정리

Ⅲ 프로젝트 결론
결과
안드로이드 악성 앱 분석을 위한 언패커 개발

✓ 프로젝트 산출물 및 최종 결론

Ⅰ Ⅱ Ⅲ 결론
결과

▲ 한이음 ICT 멘토링 진행 및 논문 투고 ▲ 학회 논문 장려상 수여


Ⅲ
프로젝트 결론
결과
안드로이드 악성 앱 분석을 위한 언패커 개발
프로젝트 산출물 및 최종 결론
Ⅰ Ⅱ Ⅲ
결론
결과
패킹된 악성 앱 분석을 위한 언패커 개발
- Yara Rule과 Frida Script를 활용하여 Python 언어로 개발된 안드로이드 전용 언패커
악성 앱 분석 내용을 기반으로 한 분석 보고서
- 프로젝트를 진행하며 관련된 모든 내용을 노션에 정리 정리한 내용을 바탕으로 보고서 제작
연구 내용을 기반으로 한 논문 (학회 투고 및 컨퍼런스 발표)
- 패킹된 악성 앱을 언패킹하기 위한 연구 내용을 바탕으로 논문 작성 및 컨퍼런스 발표
패킹 기법에 관한 연구 증진
48/49


감사합니다.