ModCrypter 기술 요약

1. Portable Executable File 구조

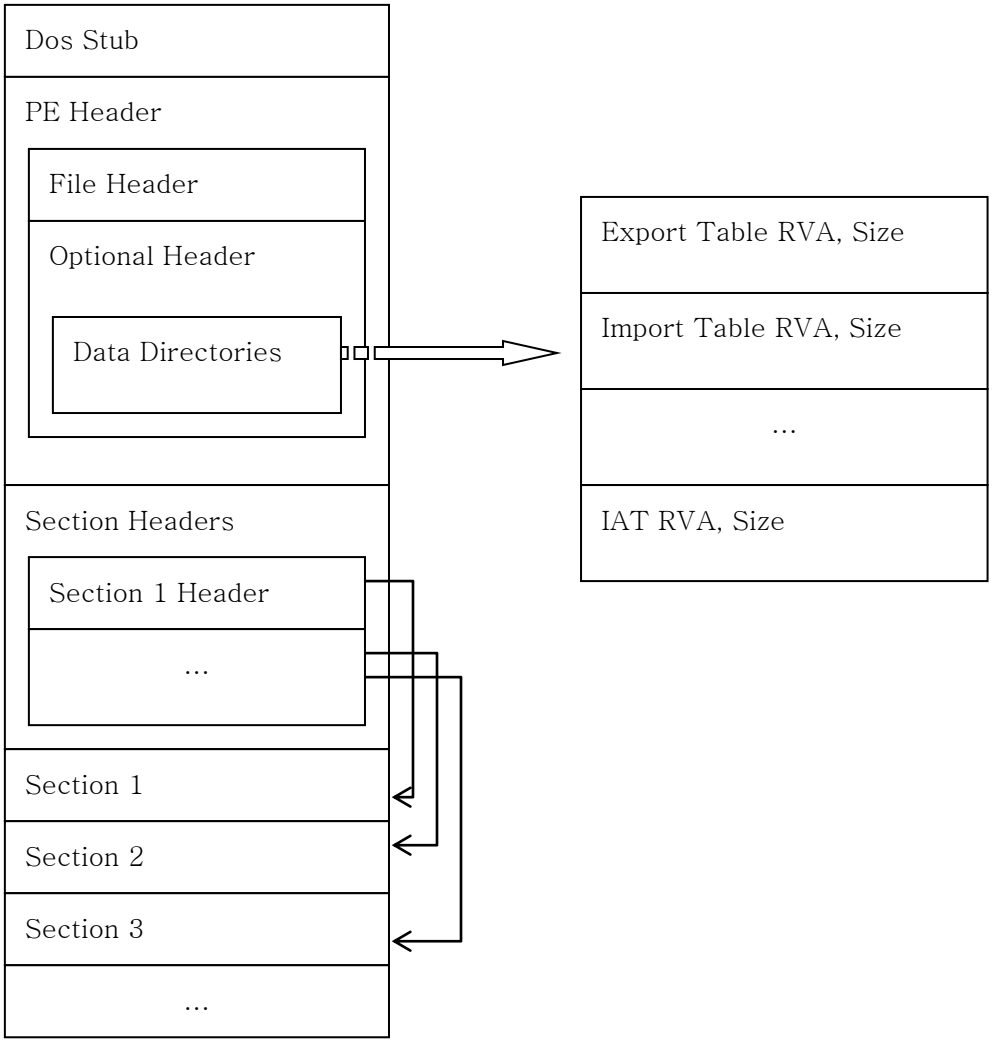


그림 1. PE File 구조

(1) Dos Stub과 Signature

16 비트 윈도우 호환을 위해 사용되는 헤더로써 약 100바이트 정도로 구성되어 있다. IMAGE_DOS_HEADER 구조체에 구조가 정의되어 있으며 처음 2바이트("MZ")를 서명 값으로 사용한다.

(2) PE Header

i. File Header

실행 가능한 시스템 및 섹션의 수와 같은 PE 파일의 전반적인 정보를 제공하는

헤더 이다.

ii. Optional Header

File Header에 기록된 정보보다 좀더 다양한 PE 파일의 정보를 제공하는 헤더로서 PE파일을 나타내는 Magic Number, 링커 버전, 코드 및 데이터의 베이스 RVA, 프로그램 시작 주소(Entry Point) RVA, PE파일 정보 테이블이 기록 되어있다.

(3) Section Header

Section Header는 프로그램 실행에 필요한 코드, 데이터, 리소스 그리고 기타 정보를 담고 있는 Section들의 헤더이다. Section Header에는 Section의 RVA, Virtual Size, Low Data Offset, Low Data Size 등 Section에 관련된 정보를 담고 있다.

(4) Section (Raw Data)

PE 파일에서 프로그램이 실행하는 코드, 데이터, 리소스를 담고 있는 부분이다. 컴파일러에 따라 Section의 이름은 달라질 수 있으며, PE 파일은 적어도 하나의 Section은 가지고 있어야 한다.

ModCrypter에서 암호화할 Section인 코드 영역의 Section의 이름은 “.text”, “code”, “CODE”이며 데이터 Section이름은 “.data”, “data”. “DATA”이다.

주석

[1] RVA(Relative Virtual Address) : PE 파일이 메모리에 사상될 때 각 정보의 조각의 주소를 표현하기 위해 사용되는 값으로 메모리에 사상되는 주소가 일정하지 않으므로 Image Base와의 상대 거리로 표현된 값이다.

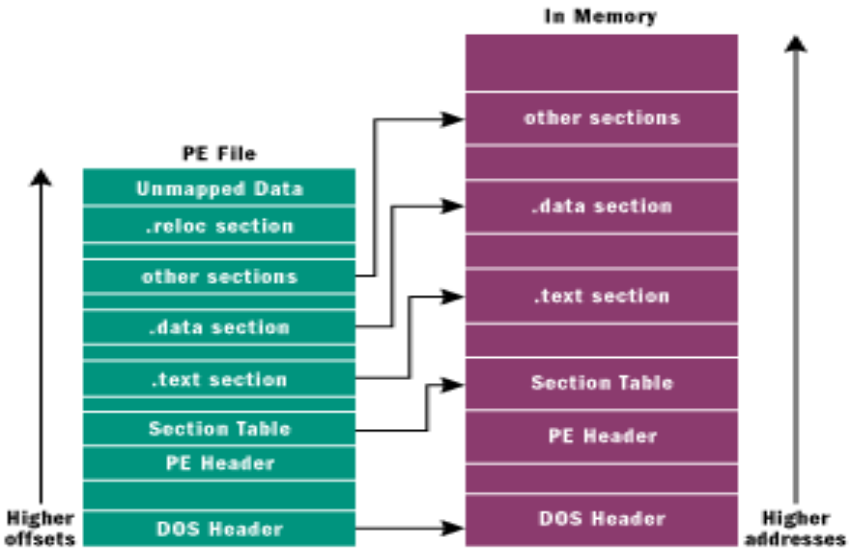


그림 2. 메모리에 사상된(Memory-Mapped) PE 파일

2. ModCrypter가 적용된 PE 파일구조

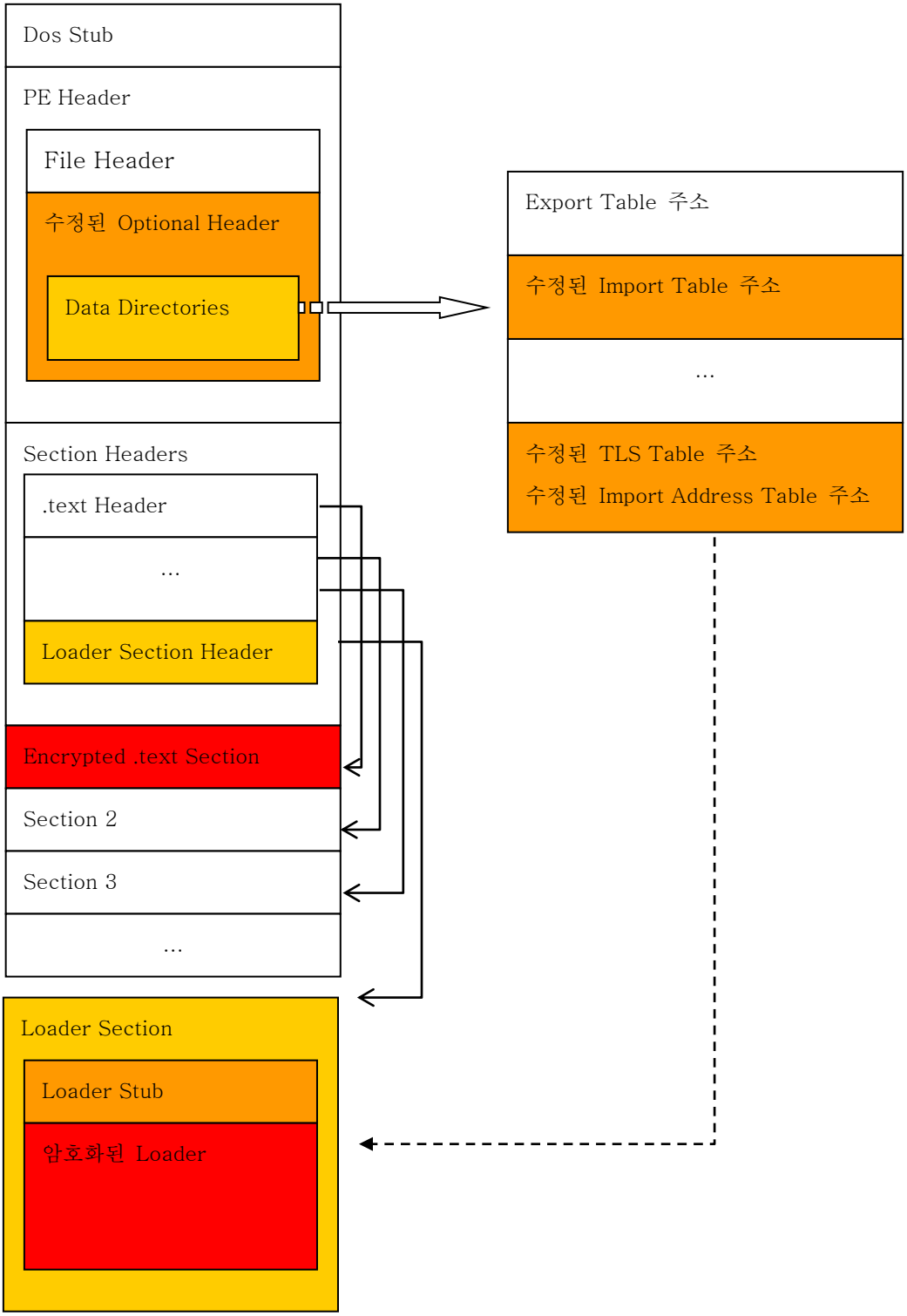


그림3. ModCrypter가 적용된 PE 파일 구조

3. Loader Section 구조

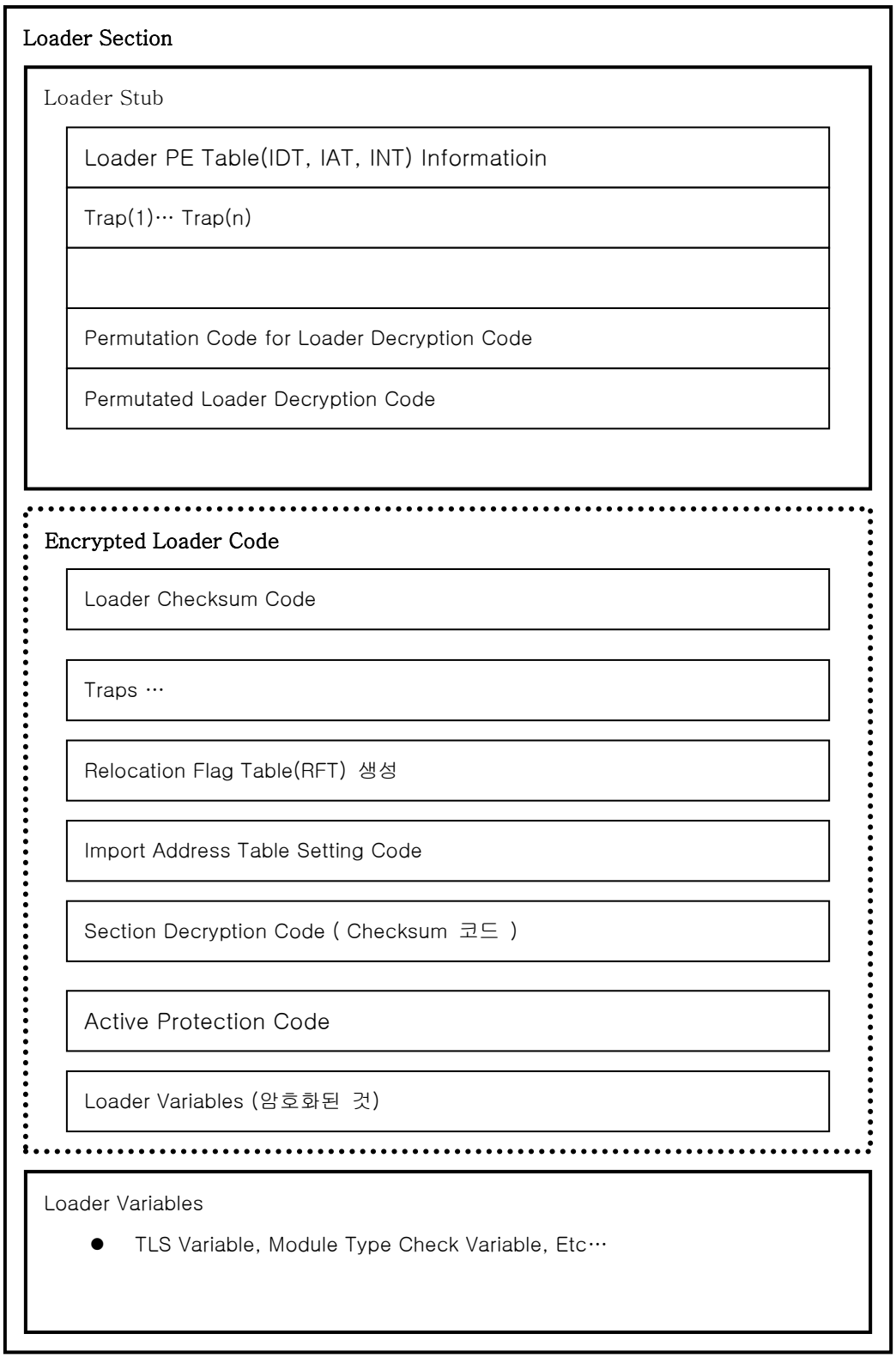


그림4. ModCrypter Loader Section 구조

4. Loader Section 트랩

(1) Anti-Debug, Anti-SoftIce

디버거 동작 유무를 확인할 수 있는 코드(PEB 구조, SHE Handler, Search Service)를 통해 디버거 발견 시 종료 또는 임의의 주소로 분기하도록 함

(2) Partial Checksum

디버거가 Breakpoint를 설정하게 되면 메모리 코드가 변하는 것을 이용하여 Loader의 특정 부분의 올바른 Checksum인 경우만 다음 명령어가 실행할 수 있게 함.

(3) XOR Trap

로더에서 사용되는 옵션을 얻기 위해 암호화된 옵션에서 정상적인 옵션을 구하는 과정

- 현재 옵션 XOR 암호화된 옵션 = 계산된 옵션
- 계산된 옵션 변수에 저장
- 로더 복호화 시에 계산된 옵션과 암호화된 옵션을 계산하여 정상적인 옵션을 구할 수 있음

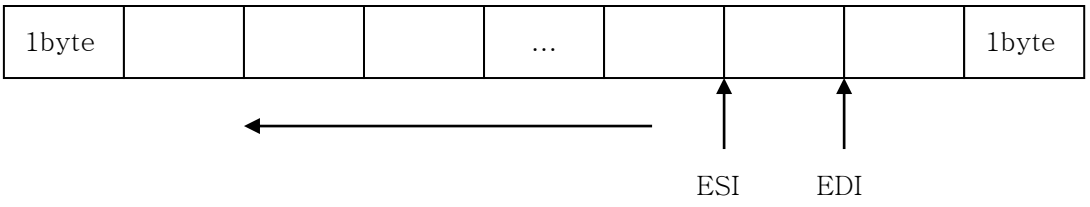
(4) 치환된 Loader 복호화 코드

디버거 또는 디스어셈블러를 사용하여 Loader Section의 Loader 복호화 코드 테이블을 참조하는 것을 방지하기 위해 Loader Decryption Code를 치환(permutation)함

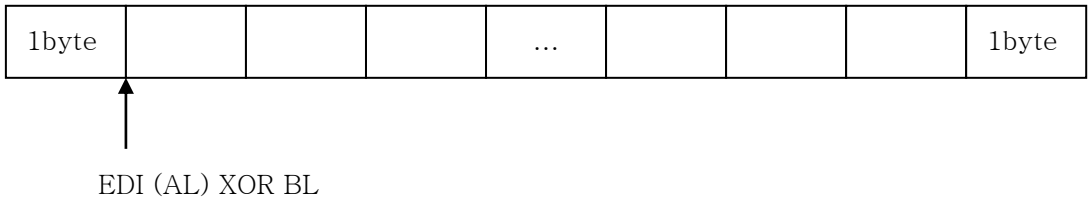
[설명] - Loader Decode Buffer의 OPCODE를
- Mix된 Decode Buffer 방식은 ..
Random Number xor B1 = B1'
B1' xor B2 = B2'
B2' xor B3 = B3' 로 생성되어 있는 상태로

B60' xor B5f' = B60
B5f' xor Bfe' = Bf5
B1' xor Random number = B1으로 얻을 수 있다.

- 루프 한번 실행하지 전



- 루프 마지막 5F번 실행 후



5. Loader Section & Code Section 암.복호화

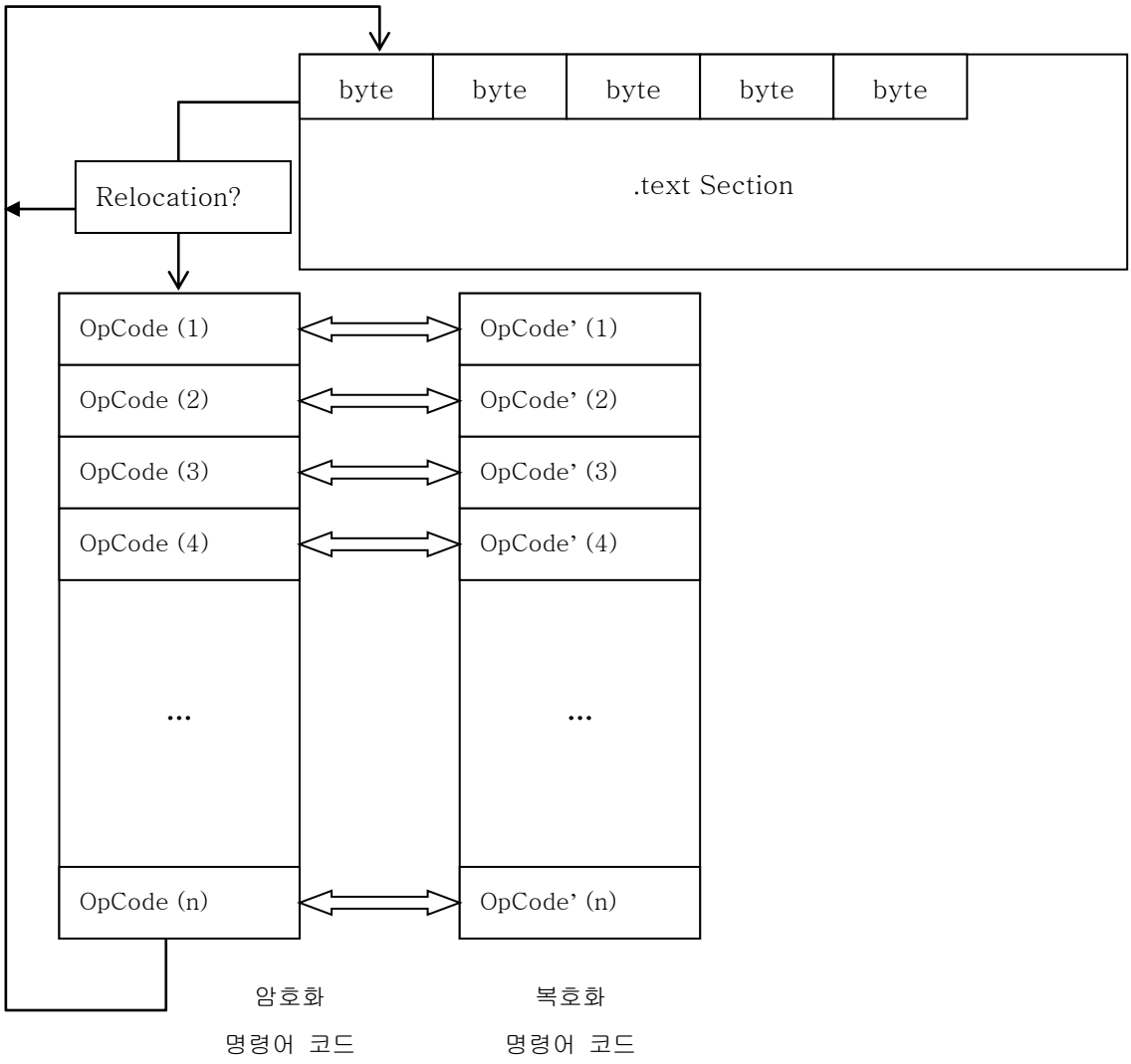


그림5. 코드영역 암호화 복호화 과정

Loader 암복호화 방식은 Section Encryption방식에서 Relocation Flag Table을 참조하는 것을 제외하고 동일하며 복호화는 복호화 명령어 코드를 통해 암호화 방식과 동일하게 동작한다.

암호화 과정

(1) 파일에 위치한 .text Section 영역 찾음

(2) .text Section 영역의 크기 계산

(3) 암호화/복호화 함수 코드 생성과 암호화

A. 섹션 암호화 시 Relocation Flag Table 생성. Dll 또는 OCX 경우 ImageBase의 충돌이 일어날 때 Relocation Table을 참조하여 Relocation Flag Table을 생성한다. (부록 A참고)

B. 23가지의 하드웨어 명령어로 구성된 배열에서 96(0x60) 바이트 크기의 명령어를 임의로 선택한다. (단 23가지의 하드웨어 명령어는 reversible code의 쌍이며 junk code로 구성됨)

C. 하드웨어 명령어 중 피 연산자가 필요한 경우 Pseudo random number 생성기로 값을 함께 생성한다.

D. A와C 과정에서 얻어진 정보로 암호화 명령어 코드 테이블를 생성한다.

E. 암호화 명령어 코드 생성시 복호화를 하기 위해 암호화 명령어 코드에 대응하는 복호화 명령어 코드 테이블을 함께 생성(Relocation Flag Table과 함께 로더 영역에 복사해 둔다.)

F. 그림5와 같이 암호화 대상 섹션을 Relocation Flag Table을 참조하여 바이트 단위로 암호화 한다. (모듈이 Relocation Table을 참조 할 때 Relocation Flag Table의 값이 0x00의 경우 현재 위치의 바이트의 암호화는 SKIP)

복호화 과정

(1) PE 파일을 암호화 할 때 암호화 명령어 테이블 생성 시 함께 생성한 복호화 명령어 테이블이 로더 위치하고 있다.

(2) PE 파일의 코드영역의 위치(RVA)와 크기를 얻어온다. (PE 헤더 분석)

(3) 코드 영역을 복호화 명령어 테이블과 Relocation Flag Table을 참조하여 복호화 한다.

6. Loader Section/ Code Section 체크섬

(1) Section의 무결성을 검증하기 위해 각 Section의 바이트들을 연쇄적으로 더하여 값을 얻고 그 값은 암호화 되는 Loader 변수에 저장된다.

7. .Data Section 암.복호화

data 섹션 암.복호화의 경우 코드 영역의 암호화 방식과 같으나 고객의 PC에서 수집한 하드웨어 정보의 해쉬 값(벤더키)을 암호화 연산의 키 재료로 사용하여 키 재료가 일치하지 않으면 정상 동작하지 않는 차이점이 있다. 다음은 Data 섹션을 벤더키를 가지고 암복호화 하는 과정을 그림으로 나타내었다.

(1) 벤더 키 쌍 생성

- ModCrypter 등록 시 발급된 벤더 키(레지스트리 정보)와 직접 계산된 벤더키를 데이터 섹션을 암복호화의 키 재료로 사용한다.
- 정상적인 등록이라면 레지스트리에 있는 키와 계산된 키 값은 같음

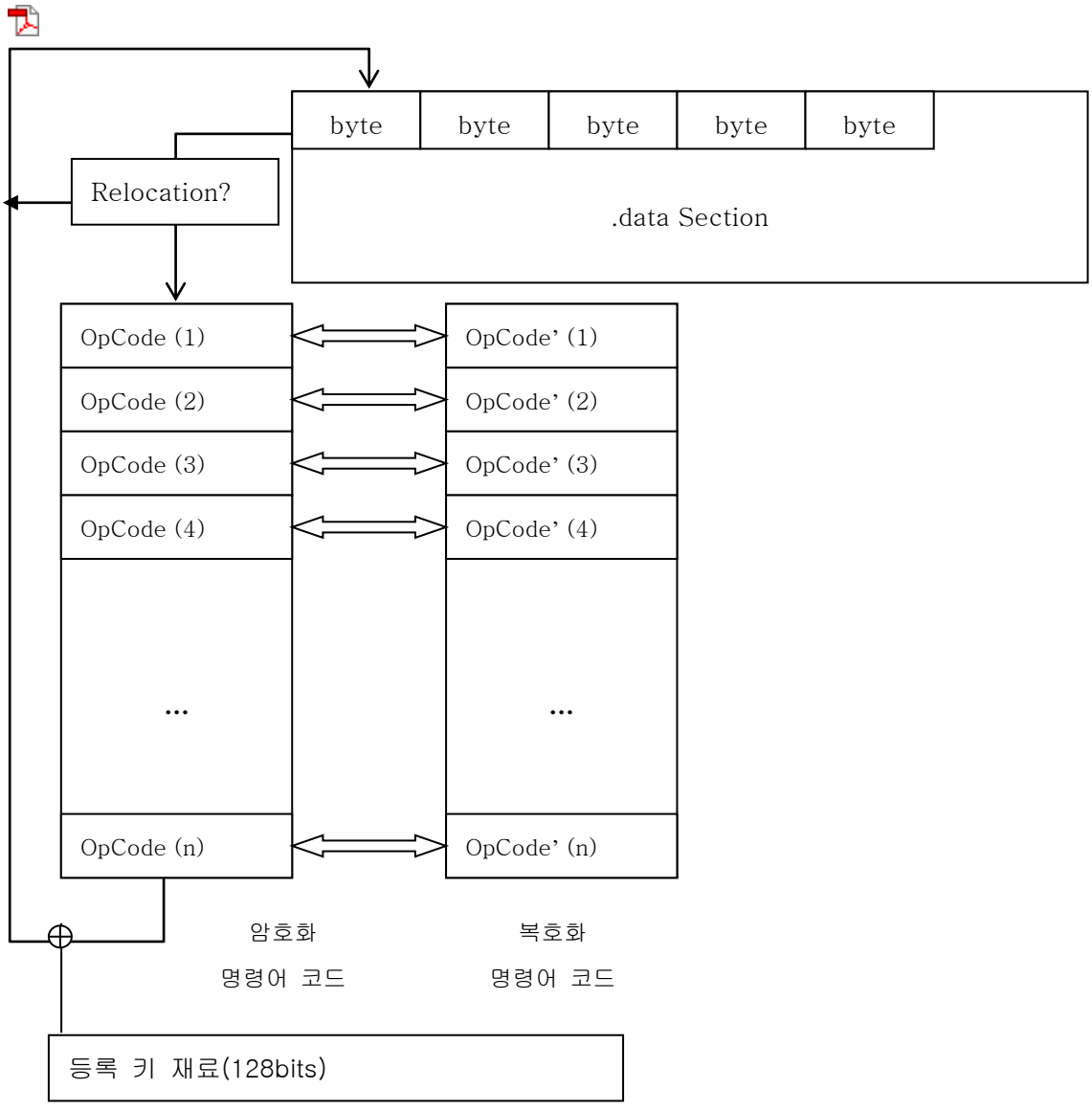


그림 6. 데이터 영역 암/복호화 과정

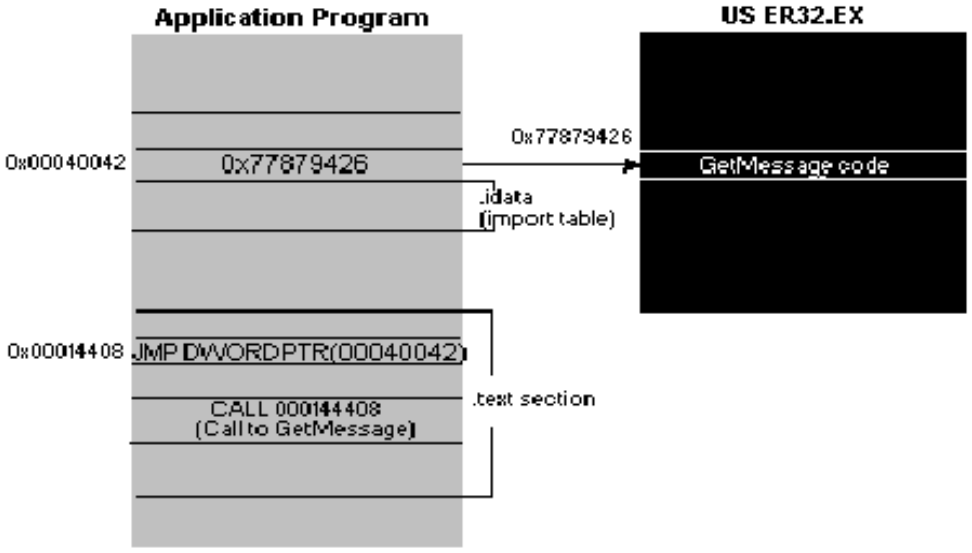
8. Import Address Table(IAT) Setting

(1) Imported Symbol과 IAT

컴파일러는 함수를 호출하는 구문을 발견하게 되면 그 함수가 내부 모듈에 있는지 외부 모듈(dll)에 있는 함수 인지 알지 못하므로 CALL XXXXXXXX(내부) 또는 CALL DWORD PTR[XXXXXXXX]의 콘드를 생성하지 못한다. 결국 컴파일러는 CALL XXXXXXXX 와 같이 임시 Transfer Area를 가리키는 "Trampoline Address"로 주소를 고정하게 되며 링커는 그 함수 호출에 대한 Stub function(JMP XXXXXXXX 형태)을 제공하여 실제 내부 모듈의 호출인지 외부 모듈의 호출인지 구분하게 해준다.

만약 내부 모듈의 함수라면 링커는 "CALL 0xXXXXXXXX"을 호출한 Transfer Area에 "XXXXXXXX: JMP 0xYYYYYYYY" 형태가 될 것이고 외부 모듈 함수라면 Transfer Area에 "XXXXXXXX: JMP DWORD PTR[0xYYYYYYYY]" 형태가 된다.

외부 모듈의 DWORD PTR[0xYYYYYYYY]의 주소 0xZZZZZZZZ(IAT의 주소)은 시스템 로더가 모듈을 올린후 결정 되는 것이므로 링커가 임의의 값을 채워 넣는다.



(2) ModCrypter 복호화 시 Import Address Table Re-Addressing

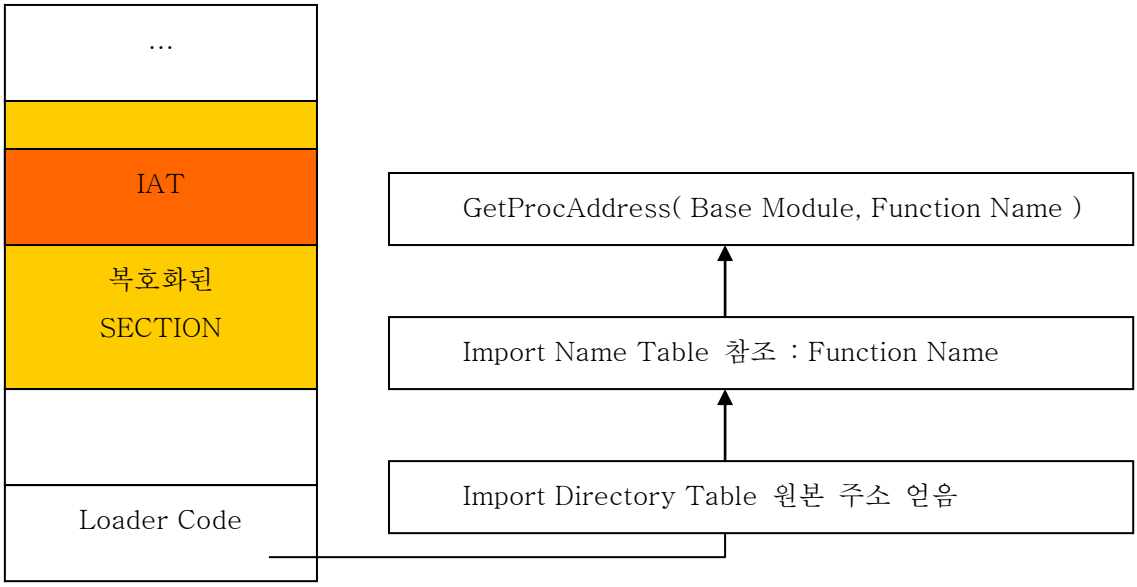
만약 IAT가 ".text" 영역에 위치한다면 암호화 된 모듈이 실행 될 때 시스템 로더가 암호화 된 영역에 IAT 영역에 import function의 실제 주소를 채워 넣게 된다. 따라서 ModCrypter로더가 .text 영역을 복화화 하면서 IAT의 주소를 바꾸게 되며 체크섬도 달라지

게 된다. 결국 복호화 된 후 다시 IAT의 Import Function의 주소를 채워줘야 한다.

(3) Ordinal Passing

IAT를 채워 줄 때 GetProcAdress에 두번째 인자가 Ordinal 이면 Import Name Table의 실제 데이터 값의 WORD Value 만을 인자로 전달해야 한다.

RVA	DATA	VALUE
0xXXXXXXXX	80000C40	0C40



9. Active Protection Code

(1) Active Anti-Debug

디버거가 실행 중에 프로세스에 Attach하여 디버깅하는 것을 방지하기 위해 주기적으로 디버거 존재 유무를 검사한다. 일반적으로 크래커는 프로세스를 디버깅하기 위해 윈도우 제공함수에 Break Point를 설정하여 디버깅 하기 때문에 윈도우 제공함수를 사용하여 디버거를 검사하지 않고 어셈블리 코드를 직접 작성하여 PEB에 디버거 존재 유무를 검사한다.

(2) Active Anti-Dump

ModCrypter가 암호화한 PE 파일은 메모리에 로드되면 복화화 되기 때문에 메로리에 로드된 프로세스를 덤프하게 되면 원본 코드를 얻을수 있다. 따라서 메모리 덤프 유틸로부터 덤프를 방지하기 위해 PE파일의 헤더 정보를 조작한다.

(3) Active Anti-Memory Hack

PE파일을 크래커가 조작하는 것은 ModCrypter가 보호할 수 있지만 메모리상에 특정 코드 영역 패칭하게 된다면 프로그램은 의도되지 않은 방식으로 동작할 수 있다. 따라서 메모리 사상된 코드영역을 주기적으로 검사하여 패칭된 경우 종료한다.

(4) Anti-API Hooking

앞서 PE 파일이 실행시 참조하는 외부 함수는 Import Address Table을 통해 실제 주소를 얻는다고 설명하였다. 만약 크래커가 IAT에 주소를 바꿔치기(IAT Hook)한다면 프로그램은 의도된 데로 동작하지 않을 수 있다. 따라서 ModCrypter는 PE파일에서 IAT를 설정하고 난 다음 Import Descriptor Table의 RVA를 로더의 Import Descriptor Table로 설정해 놓는다.

(5) Original Entry Point Hiding

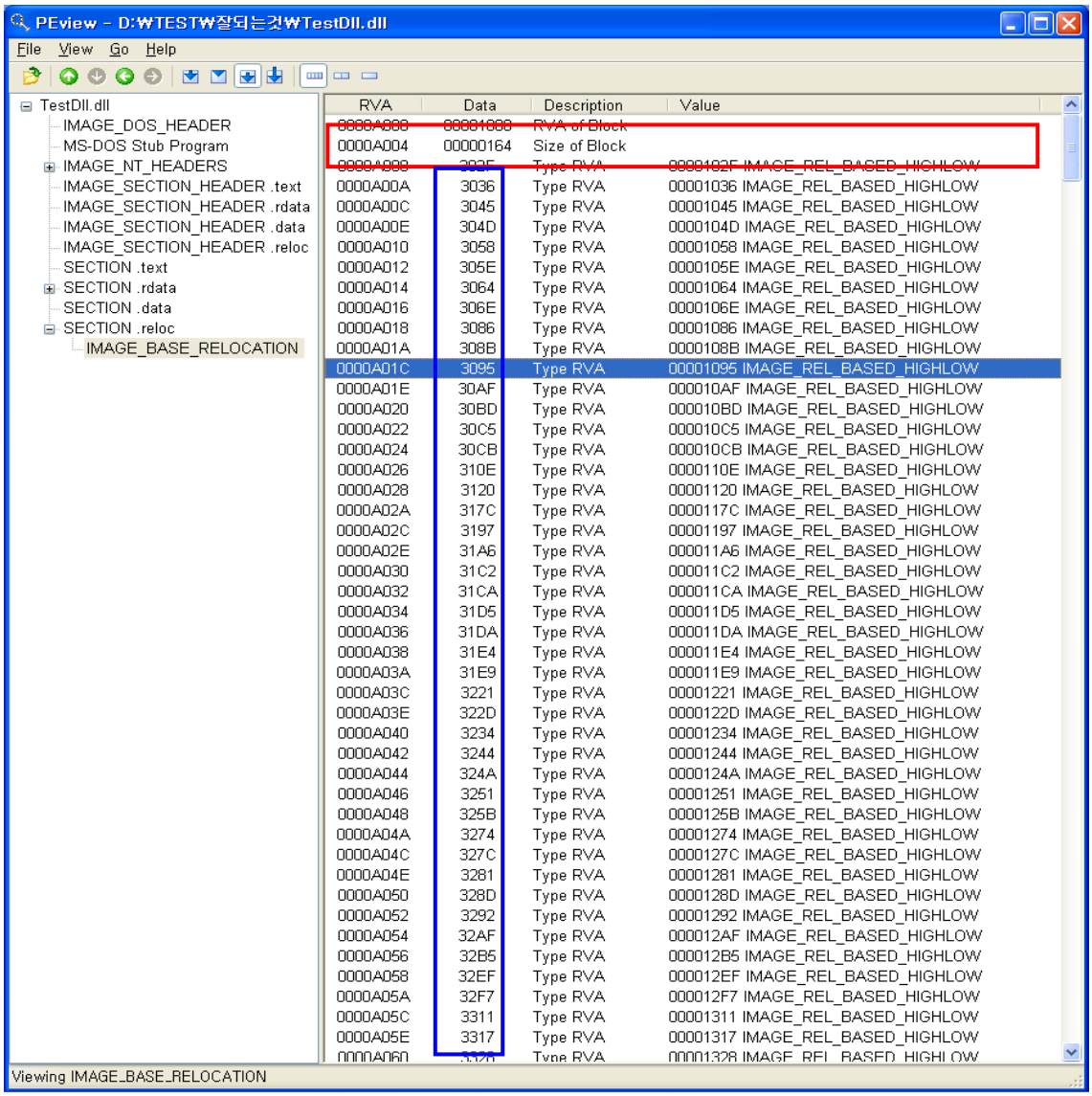
원본 시작 주소의 바이트(0~100)를 의미없는 값으로 조작하여 크래커가 ModCrypter가 적용된 PE파일의 원본을 덤프를 해서 얻더라도 프로그램을 정상 동작하지 않게 한다. (부록 B참조)

부록 A

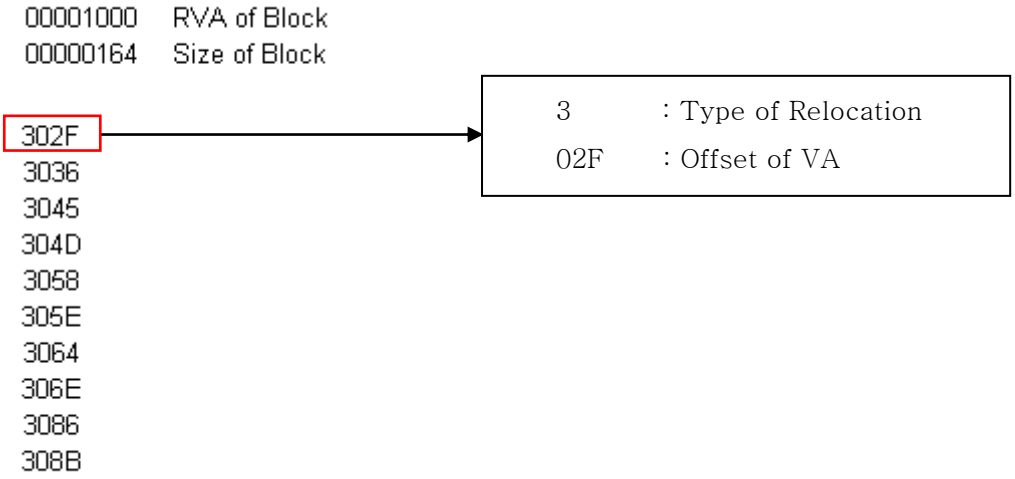
ModCrypter DLL Image Base Relocation 방법

Image Base Relocation 테이블은 다음과 같은 구조체로 구성되어 있다.

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
} IMAGE_BASE_RELOCATION;
```



1. Data 의미



(1) Type of Relocation

0 : IMAGE_REL_BASED_ABSOLUTE

1 : IMAGE_REL_BASED_HIGH

2 : IMAGE_REL_BASED_LOW

3 : IMAGE_REL_BASED_HIGHLOW

4 : IMAGE_REL_BASED_HIGHADJ

5 : IMAGE_REL_BASED_JMPADDR

6 : IMAGE_REL_BASED_SECTION

7 : IMAGE_REL_BASED_REL32

(2) Relocation Offset 구하기

Image Base + RVA(RVA of Block + OffSet of VA)

2. Block의 Entry 수

(SizeOfBlock - IMAGE_SIZEOF_BASE_RELOCATION) / 2

3. DLL Relocation Table을 참조한 ModCrypter 구현

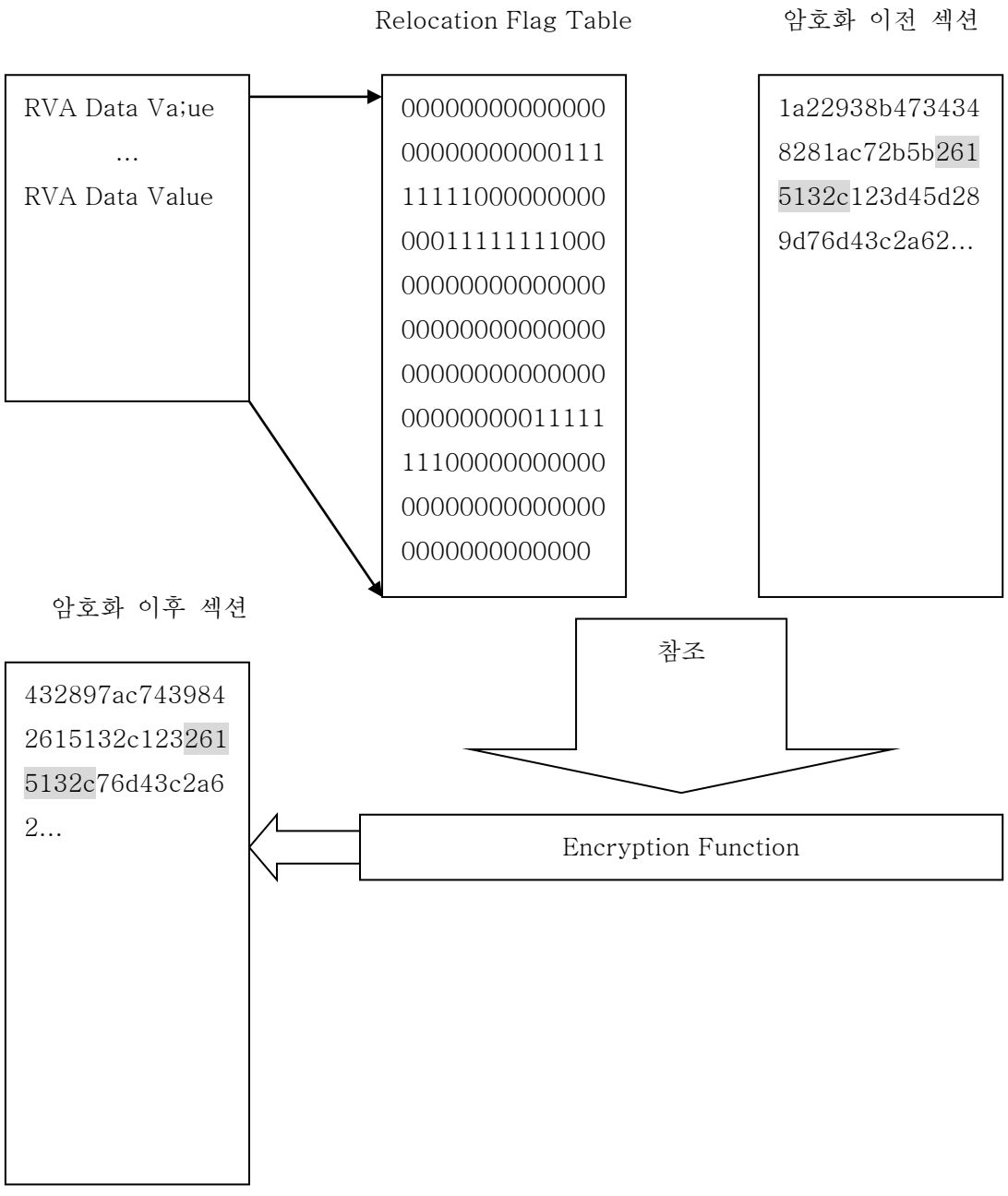
(1) 암호화 단계

- Relocation Section의 참조 여부 결정
- 암호화 루틴 이전에 Relocation Table을 참조하여 Relocation Flag Table (RFT)를 생성

- 섹션 암호화 시 RFT를 참조하여 Relocation 되는 바이트를 제외하고 암호화
- 섹션의 체크섬도 RFT를 참조하여 Relocation 되는 바이트를 제외하고 계산

(2) 복호화 단계

- Relocation Section 참조 여부 결정
- ModCrypter Loader는 복호화 루틴 이전에 Relocation Table을 참조하여 Relocation Flag Table(RFT)를 생성
- 섹션 복호화 시 RFT를 참조하여 Relocation 되는 바이트를 제외하고 복호화
- 섹션의 체크섬도 RFT를 참조하여 Relocation 되는 바이트를 제외하고 계산



부록 B

Entry Point Hiding 기술

- ModCrypter가 PE 파일을 암호화 할 때 PE파일의 시작 주소로부터 어셈블링을 통해 적당한 기계어 코드의 크기를 구한 후 로더의 변수에 저장해 둔다.
- Loader는 프로그램의 시작 주소로 점프하기 전에 코드 섹션의 시작점에서 저장된 길이의 assembly code를 동적으로 할당된 버퍼에 복사한다.
- 복사되는 코드의 크기는 올바른 어셈블링 된 코드로 이미 암호화시 구해져 있다.
- 코드 Sectioin에서 복사된 크기 만큼의 바이트에 임의이 코드를 쓴다.
- Loader 는 복사된 코드로 점프한다.
- 복사된 크기의 어셈블리 코드를 실행하고 조작된 코드 바로 다음 코드로 점프한다.
- 메모리 상에 덤프 유틸로 덤프를 하더라도 올바른 시작 주소의 코드를 얻을 수 없으며 시작 주소를 찾는 유틸(Entry Point Finder)로도 찾을 수 없다.
-