



해킹캠프



페토웍스



발표자료

# 앗!

# 야생의 아키텍처가 나타났다!

TriCore, PPC-VLE 리버스 엔지니어링

# Who am I ?

@d0now\_kim, 김도현, 010111

Vulnerability Researcher

Reverse Engineering Enthusiast

Binary Ninja Fanboy

Python Enjoyer

Working at PetoWorks

Vulnerability Researcher. Reverse Engineering Enthusiast. Binary Ninja Fanboy.  
Python Enjoyer. Working at PetoWorks.



# Introduce the topic



## 발표주제 소개



바이너리  
리버스 엔지니어링



PPC-VLE & TriCore  
바이너리 리버스 엔지니어링  
사례 소개



바이너리의 세계를 탐험할  
여러분을 위한 팁

# of the presentation

# 바이너리 리버스 엔지니어링?



앗! 야생의 아키텍처가 나타났다!

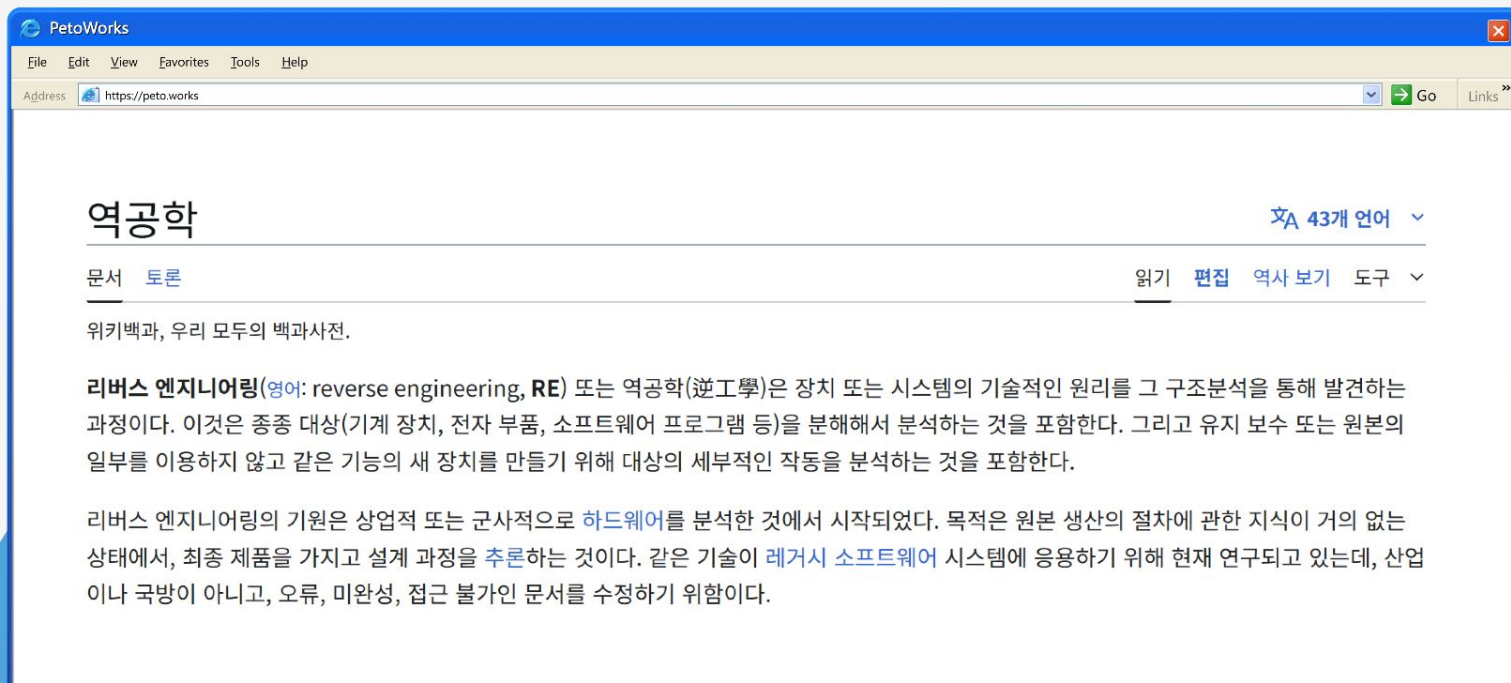
# 바이너리

---

# 리버스 엔지니어링

[Wikipedia](#)

# 리버스 엔지니어링이 뭐예요?



The screenshot shows a web browser window titled 'PetoWorks' with a menu bar (File, Edit, View, Favorites, Tools, Help) and an address bar showing 'https://peto.works'. The main content area displays the Wikipedia page for '역공학' (Reverse Engineering). The page has a blue header with the title '역공학' and a language dropdown menu set to '한글 43개 언어'. Below the title is a navigation bar with links for '문서', '토론', '읽기', '편집', '역사 보기', and '도구'. The main text of the page explains that reverse engineering is the process of analyzing a device or system to understand its internal structure and function, often used for security, forensic, or research purposes. It mentions that reverse engineering is often used to analyze hardware and software systems to understand their internal structure and function, and that it is often used to analyze hardware and software systems to understand their internal structure and function.

역공학 한글 43개 언어

[문서](#) [토론](#) [읽기](#) [편집](#) [역사 보기](#) [도구](#)

위키백과, 우리 모두의 백과사전.

**리버스 엔지니어링**(영어: reverse engineering, **RE**) 또는 역공학(逆工學)은 장치 또는 시스템의 기술적인 원리를 그 구조분석을 통해 발견하는 과정이다. 이것은 종종 대상(기계 장치, 전자 부품, 소프트웨어 프로그램 등)을 분해해서 분석하는 것을 포함한다. 그리고 유지 보수 또는 원본의 일부를 이용하지 않고 같은 기능의 새 장치를 만들기 위해 대상의 세부적인 작동을 분석하는 것을 포함한다.

리버스 엔지니어링의 기원은 상업적 또는 군사적으로 **하드웨어**를 분석한 것에서 시작되었다. 목적은 원본 생산의 절차에 관한 지식이 거의 없는 상태에서, 최종 제품을 가지고 설계 과정을 **추론**하는 것이다. 같은 기술이 **레거시 소프트웨어** 시스템에 응용하기 위해 현재 연구되고 있는데, 산업이나 국방이 아니고, 오류, 미완성, 접근 불가인 문서를 수정하기 위함이다.



# 리버스 엔지니어링 예시 #1

```
def xxxx(l: list[int]) -> int:
    r = -1
    n = 0
    for i, v in enumerate(l):
        if v > n:
            n = v
            r = i
    return r
```

Section title

이 함수의 이름을 추천해주세요!

# 리버스 엔지니어링 예시 #1

```
def xxxx(l: list[int]) -> int:
    r = -1
    n = 0
    for i, v in enumerate(l):
        if v > n:
            n = v
            r = i
    return r
```

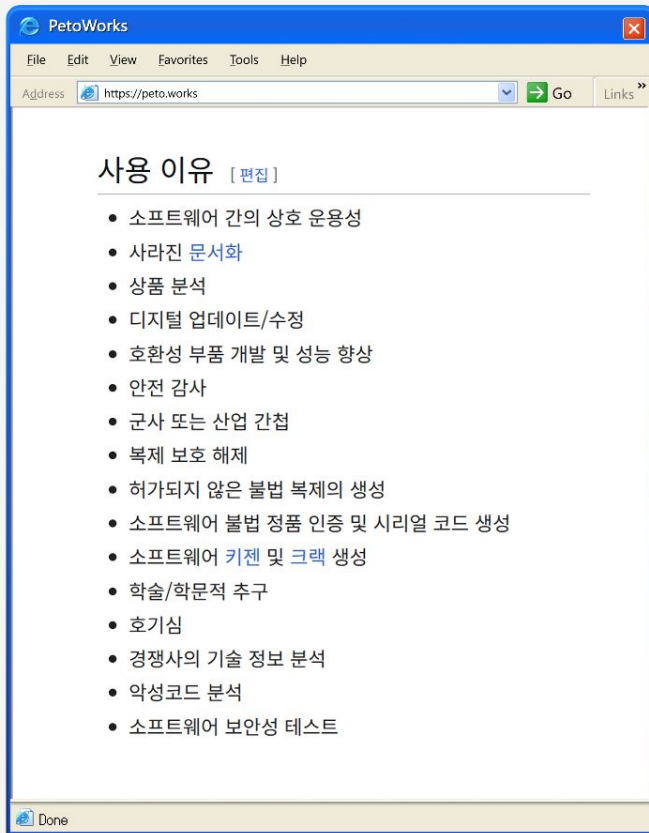
```
def xxxx(l: list[int]) -> int:
    r = -1
    n = 0
    for i, v in enumerate(l):
        if v > n:
            n = v
            r = i
    return r
```

이 함수의 이름을 추천 해 줄래?

이 함수는 리스트에서 최대값의 인덱스를 찾는 함수입니다. 다만 최대값이 아니라 현재까지의 최댓값을 추적해서 인덱스를 반환하는 단순한 방식이죠.

추천할 수 있는 이름 후보는:

- `argmax` — 수학/머신러닝에서 많이 쓰는 관례적인 이름
- `index_of_max` — 의미가 명확하고 직관적임
- `max_index` — 간단히 표현
- `find_max_index` — 함수 행위를 동사형으로 강조



[Wikipedia](#)

# 리버스 엔지니어링 사례



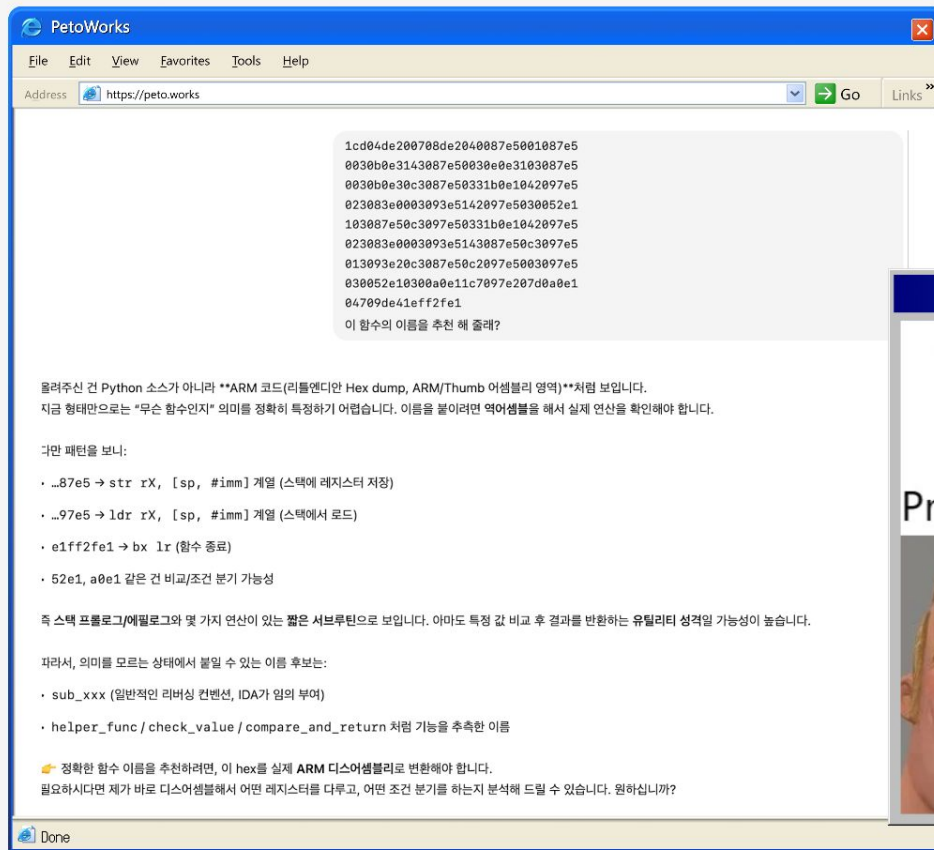
## 리버스 엔지니어링 예시 #2

```
1cd04de200708de2040087e5001087e5  
0030b0e3143087e50030e0e3103087e5  
0030b0e30c3087e50331b0e1042097e5  
023083e0003093e5142097e5030052e1  
103087e50c3097e50331b0e1042097e5  
023083e0003093e5143087e50c3097e5  
013093e20c3087e50c2097e5003097e5  
030052e10300a0e11c7097e207d0a0e1  
04709de41eff2fe1
```

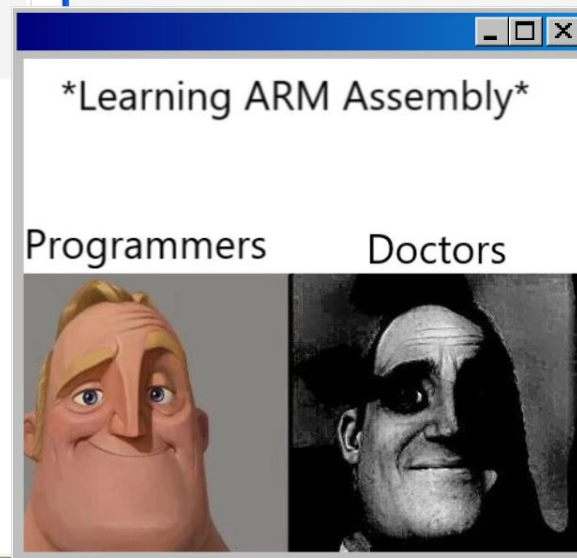
Section title

이 함수의 이름을 추천해주세요!

# 리버스 엔지니어링 예시 #2



그럼에도 놀라운 결과..



# 바이너리가 뭐예요?

- 바이너리 파일(Binary file)은 텍스트 파일과 달리, **사람이 바로 읽고 해석할 수 없는 0과 1의 이진 데이터로 구성된 파일**입니다.
- 컴퓨터만이 이해하고 처리할 수 있는 형식으로, 이미지, 동영상, **실행 파일(프로그램)** 등이 여기에 해당합니다.
- 이러한 파일들은 특정 소프트웨어나 프로그램을 통해서만 열거나 실행할 수 있습니다.



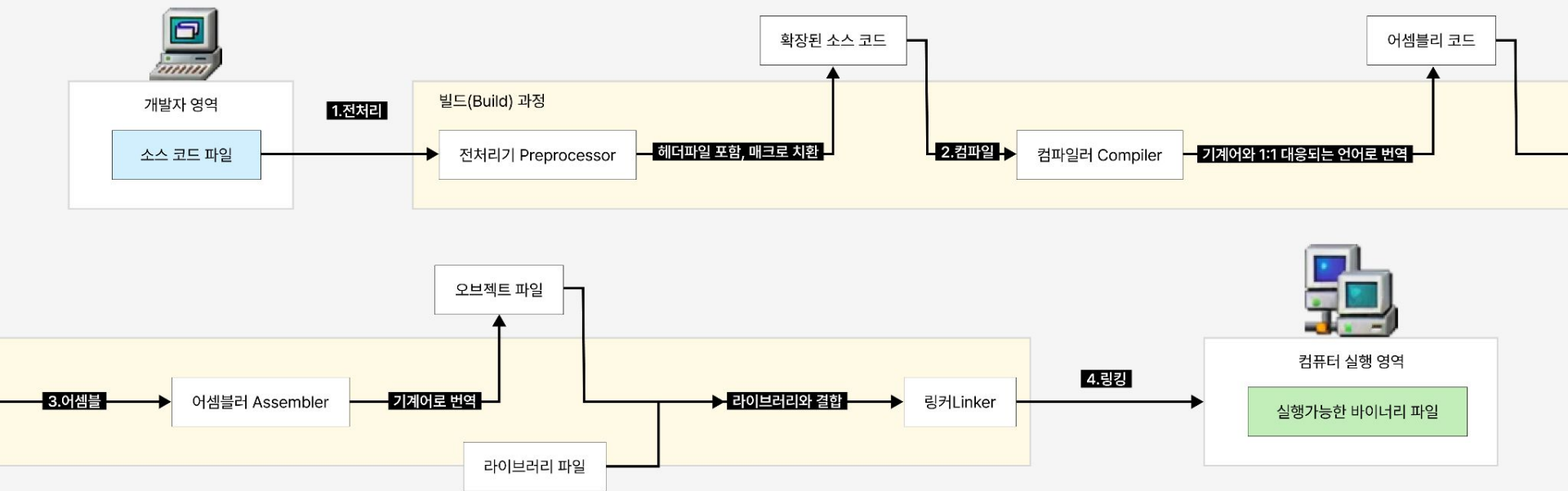
# 바이너리는 어떻게 만들어요?

소스 코드를 바이너리로 만드는 과정은 **컴파일(Compile)**이라고 부르며, 일반적으로 네 가지 주요 단계를 거칩니다.

이 과정은 사람이 이해하는 고급 언어(C, C++, Go 등)를 컴퓨터가 직접 실행할 수 있는 기계어(0과 1의 조합)로 번역하는 작업입니다.



# 바이너리 제작 과정





# 바이너리는 어떻게 읽어야 하나요?

대부분의 경우, 이미 여러분이 사용하고 계시는 IDA Pro, Ghidra, Binary Ninja로 읽을 수 있습니다만...  
이 발표에서는 그렇지 않은 경우에 대해 간략하게 다룹니다.

앗! 야생의 아키텍처가 나타났다!



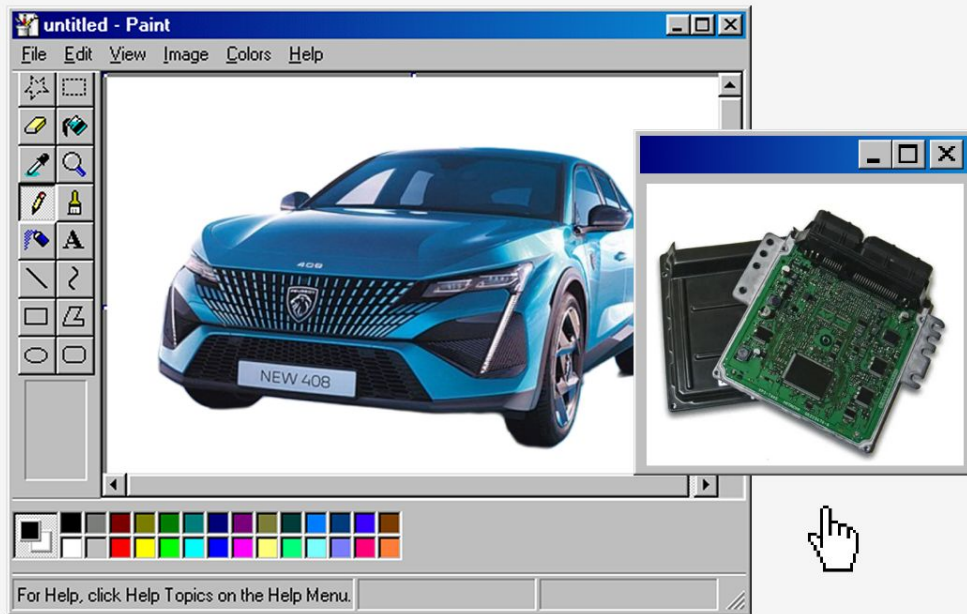
아키텍처



**야생의 아키텍처가 나타났다!**

## MISSION

전기 자동차의 컴퓨터(ECU)  
리버스 엔지니어링!

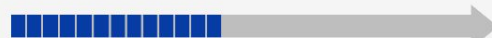
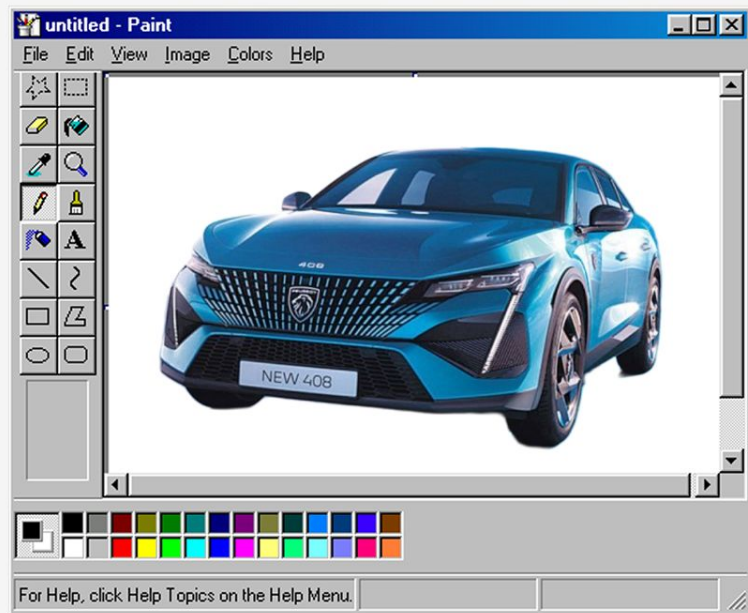


전기자동차의 컴퓨터를 제어하는 소프트웨어 = 펌웨어



**코드를 분석하기 위해서는 펌웨어를 추출해야 한다!**

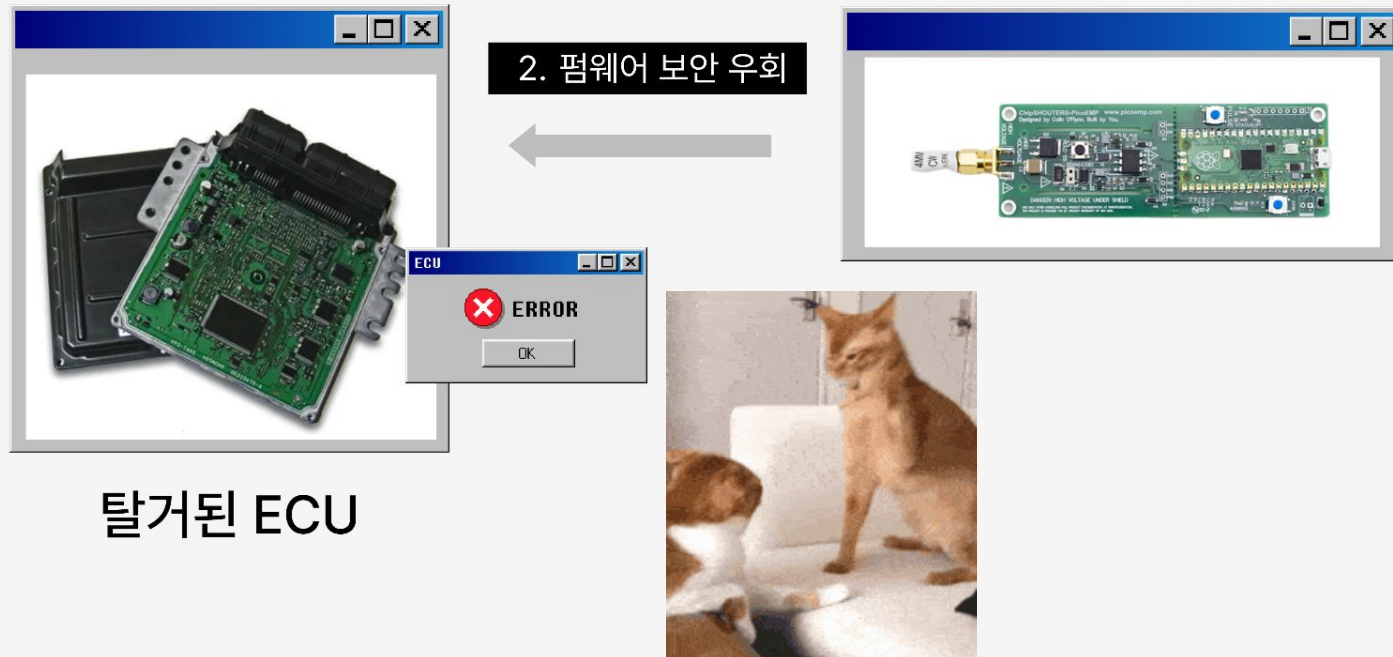
# 펌웨어 획득 과정 - ECU 탈거



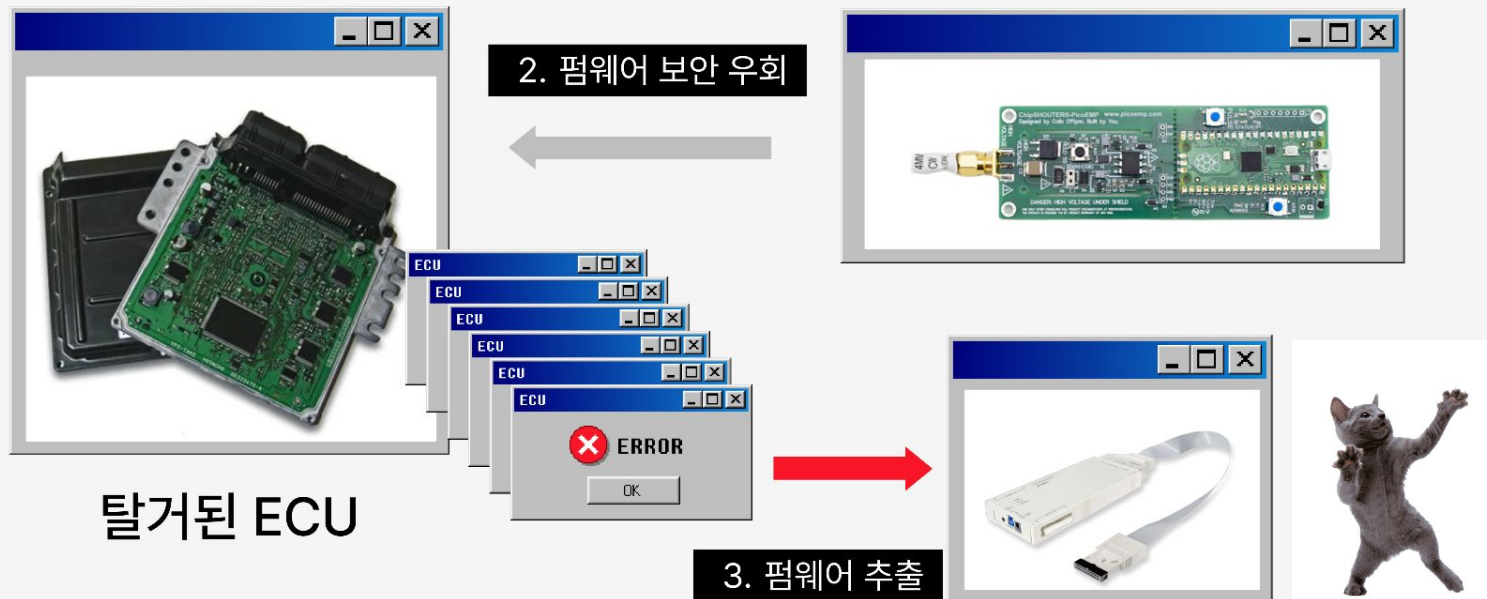
1. ECU 탈거



## 펌웨어 획득 과정 - 펌웨어 보안 우회



# 펌웨어 획득 과정 - 펌웨어 추출

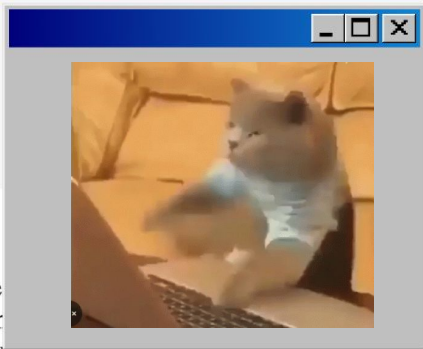


# 펌웨어 해석...?



## 1 Introduction

The MPC5746R family of 32-bit microcontrollers is the latest automotive application controllers. It belongs to an expanding family of focused products designed for flexibility to support a variety of applications. The advanced and cost-efficient host processor core of the MPC5746R automotive controller family complies with the **Power Architecture** embedded category. It operates at speeds as high as 200 MHz and offers high-performance processing optimized for low power consumption. It capitalizes on the available development infrastructure of current Power Architecture devices and is supported with software drivers, operating systems, and configuration code to assist with users' implementations. This section contains detailed information on power considerations, DC/AC electrical characteristics, and AC timing specifications.



**ECU의 프로세서가 Power Architecture(PowerPC)를 사용하는 것을 확인!**

# 펌웨어 해석...?

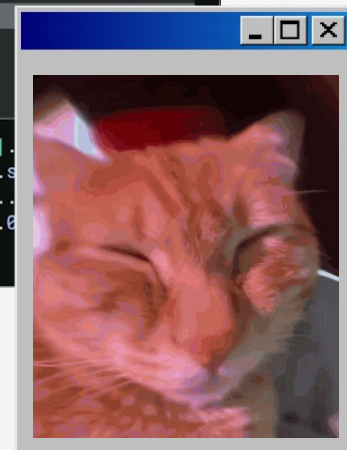


```
0000d0b8                                06 e0 1b 81 09 10 00 80 .....
0000d0c0 d9 01 73 e8 e0 00 1f ff-14 48 cc 0f 1f df 01 34-d2 01 c3 0e 18 7f 80 38-18 a1 80 08 7c 06 00 34 ..s.....H.....4.....8....|..4
0000d0e0 18 9f 80 10 68 56 1b 9f-80 08 79 ff dd 09 2a 23-48 0d e6 12 50 1f 01 30-c2 3f 64 10 ....hV....y...*#H...P..0.?d.

0000d0fc  int32_t sub_d0fc(int32_t arg1 @ r0)

0000d0fc 541f0130  rlwinm r31, r0, 0x0, 0x4, 0x18
0000d100 4810483d  bl      0x11193c
0000d104 341f0078  addic.  r0, r31, 0x78
0000d108 79ffdce1  ??

0000d10c                                c2 4f 18 7c-80 08 79 ff d6 37 90 de-1b 81 08 10 c9 01 00 90 .0.|.
0000d120 21 f1 00 04 18 21 06 f0-1b c1 09 08 00 80 d5 01-73 e8 e0 00 1f ff 14 48-c3 3f 2a 03 1b df 80 08 !....!.....s
0000d140 e6 04 c2 4f 79 ff d6 05-c2 3f 18 9e 80 08 79 ff-d5 fb c2 ff 01 f3 79 ff-d4 ad 48 60 b6 3f d4 0f ...0y....?....y..
0000d160 1b c1 08 08 c5 01 00 90-20 f1 00 04 70 e8 e0 00-30 07 15 92 18 00 e0 01-7c 03 00 34 68 53 00 04 .....p...0
```



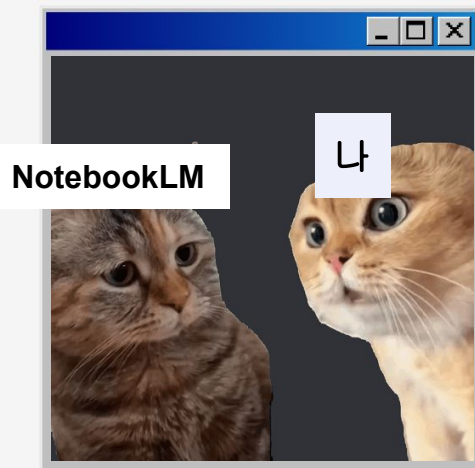
코드 해석 실패

# 펌웨어 해석...?



MPC5746R은 Power Architecture라며... 나를 속인거니 NotebookLM아?

고객님, 걱정하지 마세요. **MPC5746R** 마이크로컨트롤러는 실제로 **Power Architecture**를 기반으로 합니다. 1



# 펌웨어 해석...?

VLE는 일반 Power Architecture와 다르게 해석할 수 있는거야?

네, 고객님의 이해가 정확합니다.

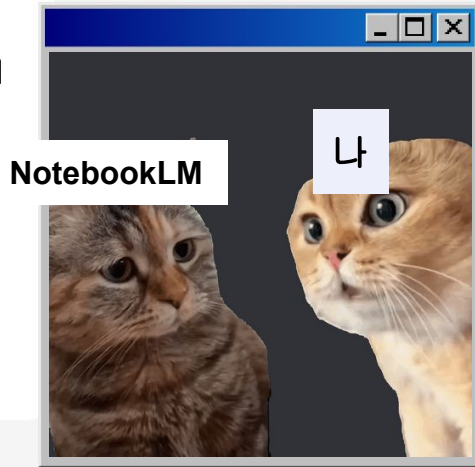
**VLE(Variable-Length Encoding)**는 일반적인 **Power Architecture**(특히 **PowerPC Book E ISA**)와는 다르게 해석해야 할 부분이 있습니다. VLE는 Power Architecture의 "확장(extension)"으로 설계되었으며, 특정 목표(코드 밀도 최적화)를 위해 몇 가지 중요한 차이점을 도입합니다.

다음은 소스에서 제공하는 VLE와 일반 Power Architecture 간의 주요 차이점 및 해석이 필요한 이유입니다:

- **확장으로서의 역할:** VLE는 기존 PowerPC Book E ISA를 완전히 대체하는 것이 아니라, 임베디드 프로세서 공간에서 코드 밀도(code density)를 개선하기 위한 "보완(supplement)"으로 간주될 수 있습니다.

즉, 기본 아키텍처는 유지하되, 특정 목적을 위한 추가적인 규칙과 인코딩을 도입하는 것입니다.

- **명령어 인코딩 및 실행 모드:**
  - VLE 명령어는 16비트 또는 32비트 길이로 인코딩되며, 16비트 경계에 정렬됩니다.
  - 이는 표준 PowerPC Book E 명령어의 고정 길이 인코딩과 다릅니다.

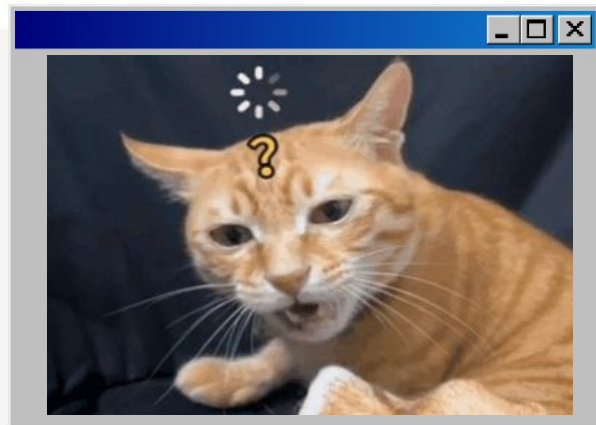


## 펌웨어 해석...?

즉, 기본 아키텍처는 유지하되, 특정 목적을 위한 추가적인 규칙과 인코딩을 도입하는 것입니다.

- **명령어 인코딩 및 실행 모드:**

- VLE 명령어는 16비트 또는 32비트 길이로 인코딩되며, 16비트 경계에 정렬됩니다.
- 이는 표준 PowerPC Book E 명령어의 고정 길이 인코딩과 다릅니다.





일반 instruction set과 VLE instruction set의 핵심 차이점에 대해 알려 줘

일반(Power ISA Books I, II, III-E) instruction set과 VLE(Variable Length Encoded) instruction set의 핵심적인 차이점은 주로 **코드 밀도 최적화**와 관련된 특성에서 나타납니다.

주요 차이점은 다음과 같습니다:

- **Instruction 길이 및 정렬 (Instruction Length & Alignment)**

- **일반 Power ISA:** 모든 instruction은 **4바이트 길이**고 **워드(word)** 단위로 정렬됩니다. 즉, instruction 주소의 하위 2비트는 무시됩니다.
- **VLE Instruction Set:** VLE instruction은 **2바이트 또는 4바이트 길이**며 **하프워드(halfword)** 단위로 정렬됩니다. 따라서 instruction 주소의 최하위 1비트는 0으로 처리됩니다.

**하지만 Binary Ninja는 PPC-VLE를 지원하지 않는다!**

- **주요 목적 (Primary Purpose)**

- **일반 Power ISA:** Book I, II, III-S, III-E, VLE에 정의된 instruction과 기능을 포괄하며, 브랜치, 고정 소수점, 부동 소수점, 벡터 instruction 등 다양한 클래스를 포함합니다.
- **VLE Instruction Set:** Books I, II, III-E에 정의된 instruction set의 **코드 밀도를 최적화한 재인코딩 버전**으로, 특히 **코드 밀도가 시스템 비용 및 성능에 중요한 임베디드 프로세서 환경을 위해** 고안되었습니다. VLE는 Books I-III instruction set에 대한 보완책입니다.

# 리버스 엔지니어링 프레임워크

앗! 야생의 아키텍처가 나타났다!



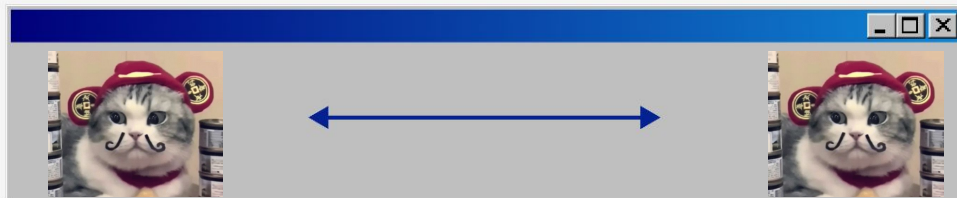
# 리버스 엔지니어링 프레임워크 (explained by cat)

스페인 고객



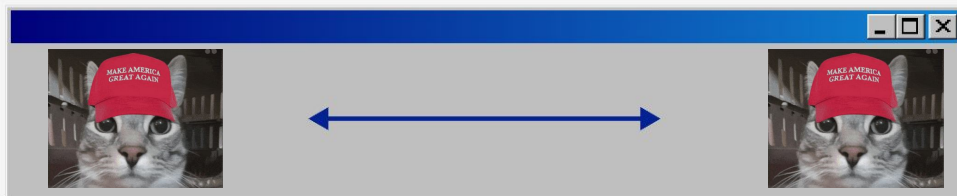
스페인 서버

중국인 고객



중국인 서버

미국인 고객



미국인 서버

고양이 레스토랑 서비스 v 1.0

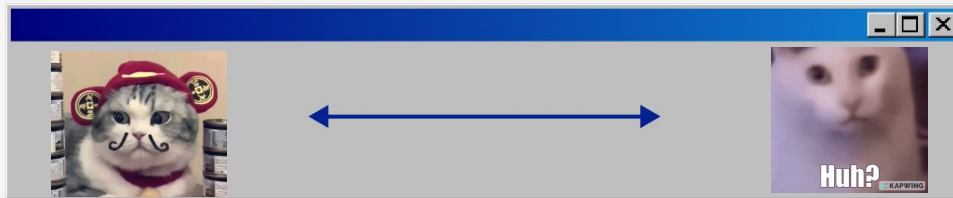
# 리버스 엔지니어링 프레임워크 (explained by cat)

스페인 고객



스페인 서버

중국인 고객



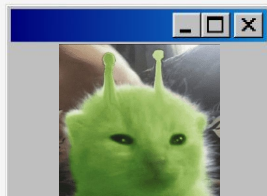
중국인 서버

미국인 고객

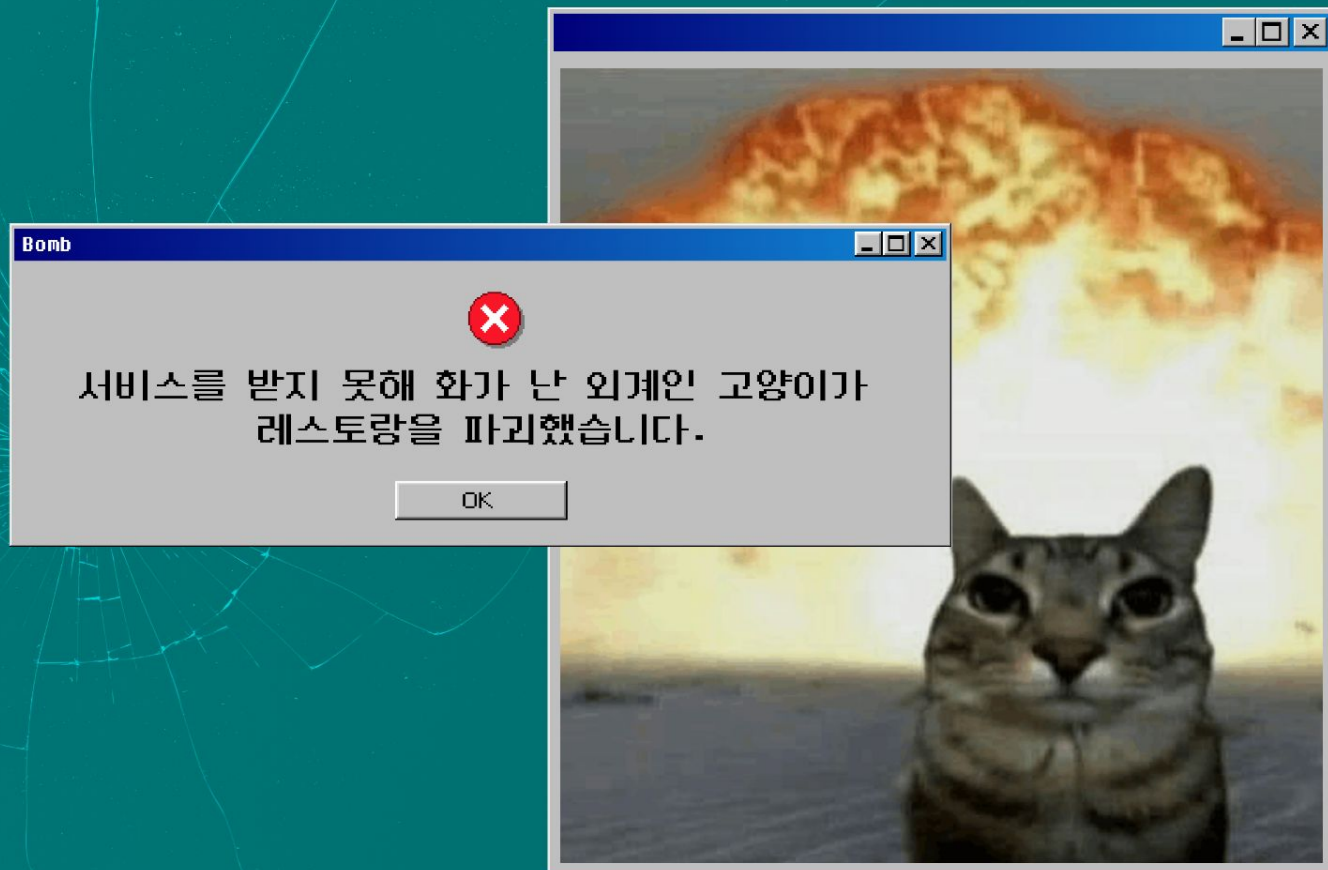


미국인 서버

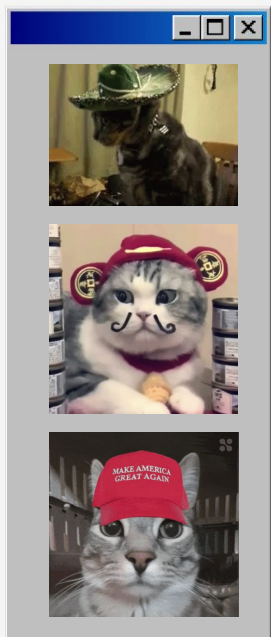
외계인 고객



# 리버스 엔지니어링 프레임워크 (explained by cat)



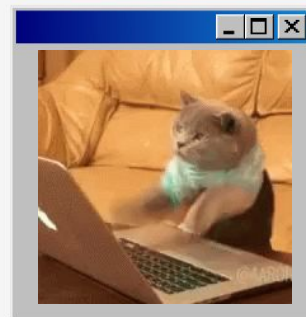
# 리버스 엔지니어링 프레임워크 (explained by cat)



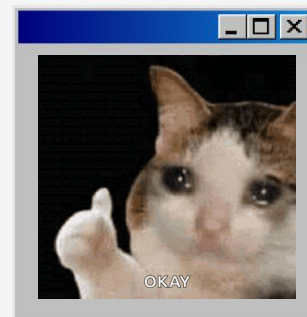
스페인 고객

중국인 고객

미국인 고객



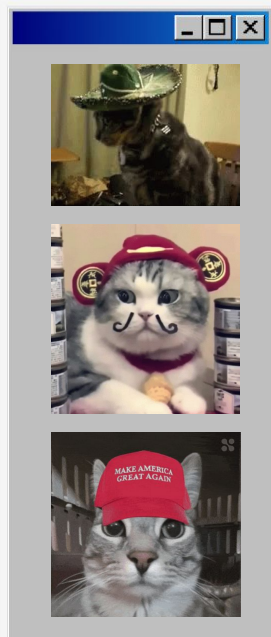
통역가  
(스페인어, 중국어, 영어, 고양이어)



고양이 서버

고양이 레스토랑 서비스 v 2.0

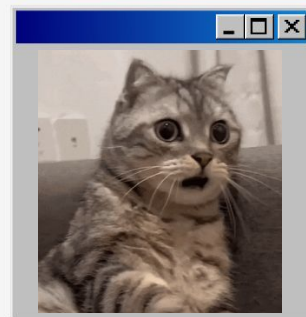
# 리버스 엔지니어링 프레임워크 (explained by cat)



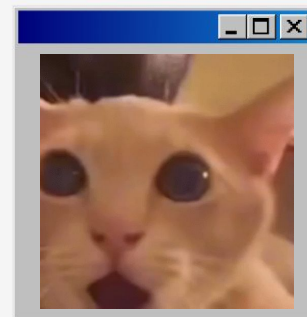
스페인 고객

중국인 고객

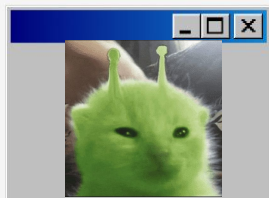
미국인 고객



통역가  
(스페인어, 중국어, 영어, 고양이어)



고양이 서버



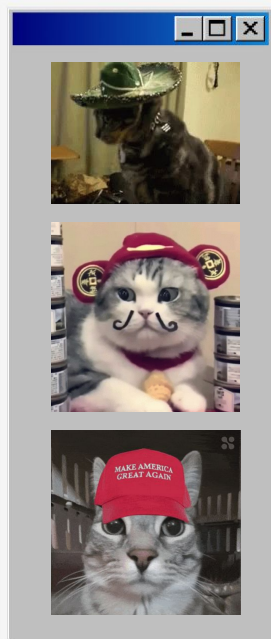
외계인 고객



외계인 고양이가 다시 나타났습니다.



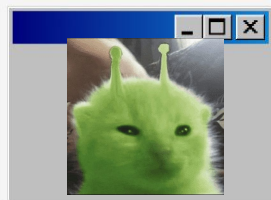
# 리버스 엔지니어링 프레임워크 (explained by cat)



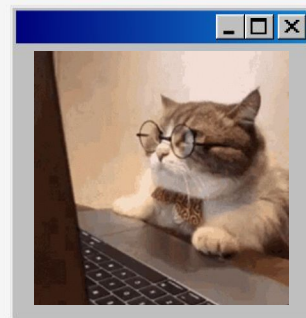
스페인 고객

중국인 고객

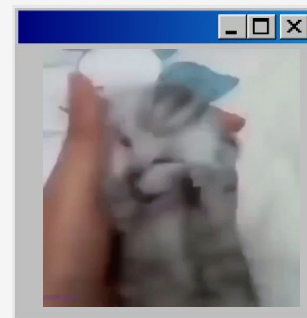
미국인 고객



외계인 고객



통역가  
(스페인어, 중국어, 영어, 고양이어)

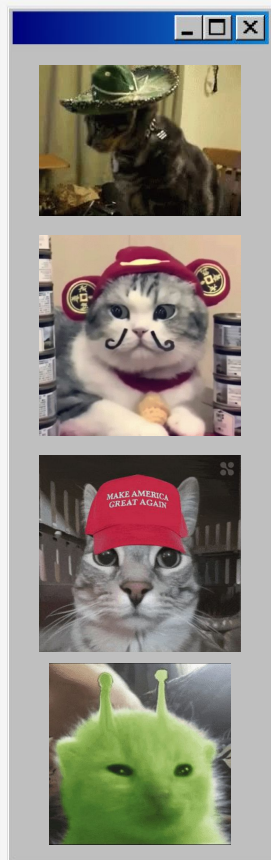


고양이 서버



통역가 고양이는 열심히 외계어를 공부합니다.

# 리버스 엔지니어링 프레임워크 (explained by cat)

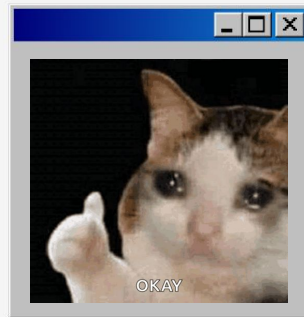


스페인 고객

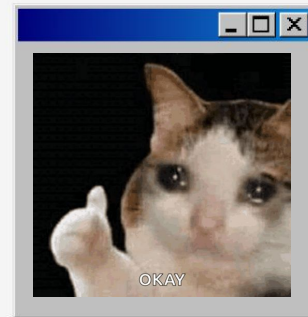
중국인 고객

미국인 고객

외계인 고객



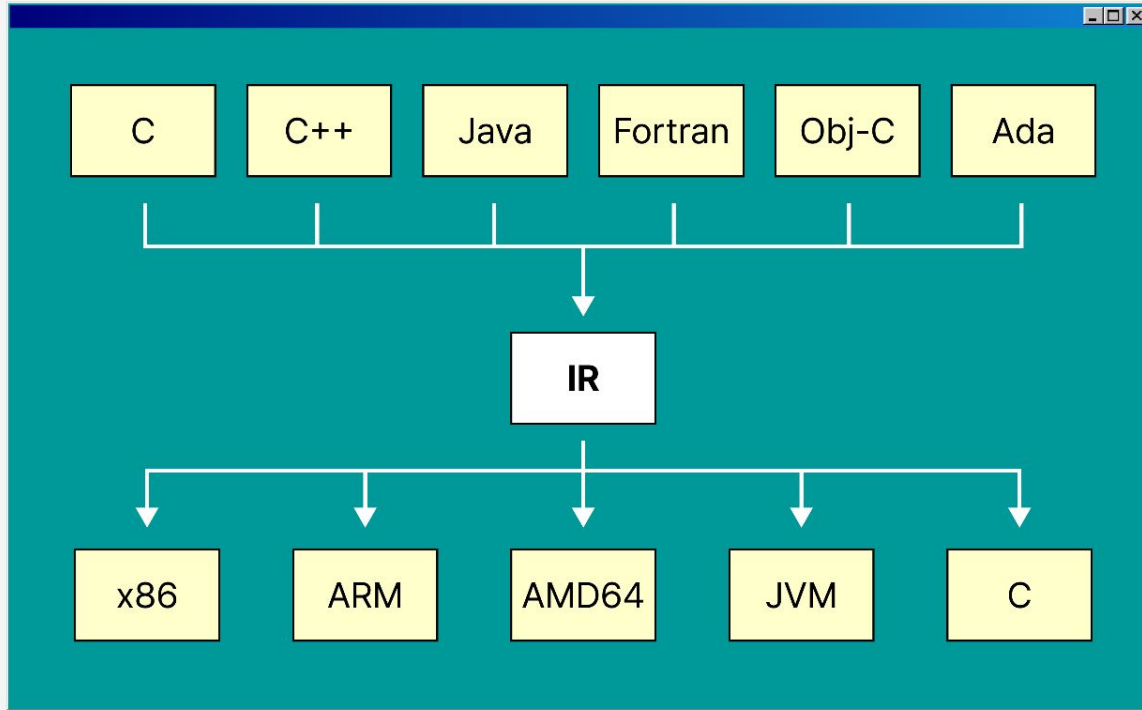
통역가  
(스페인어, 중국어, 영어, **외계어**, 고양이어)



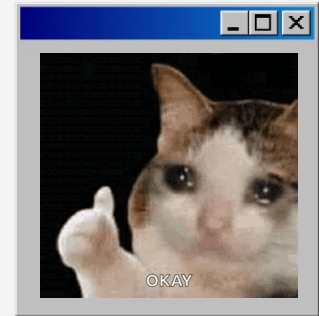
고양이 서버

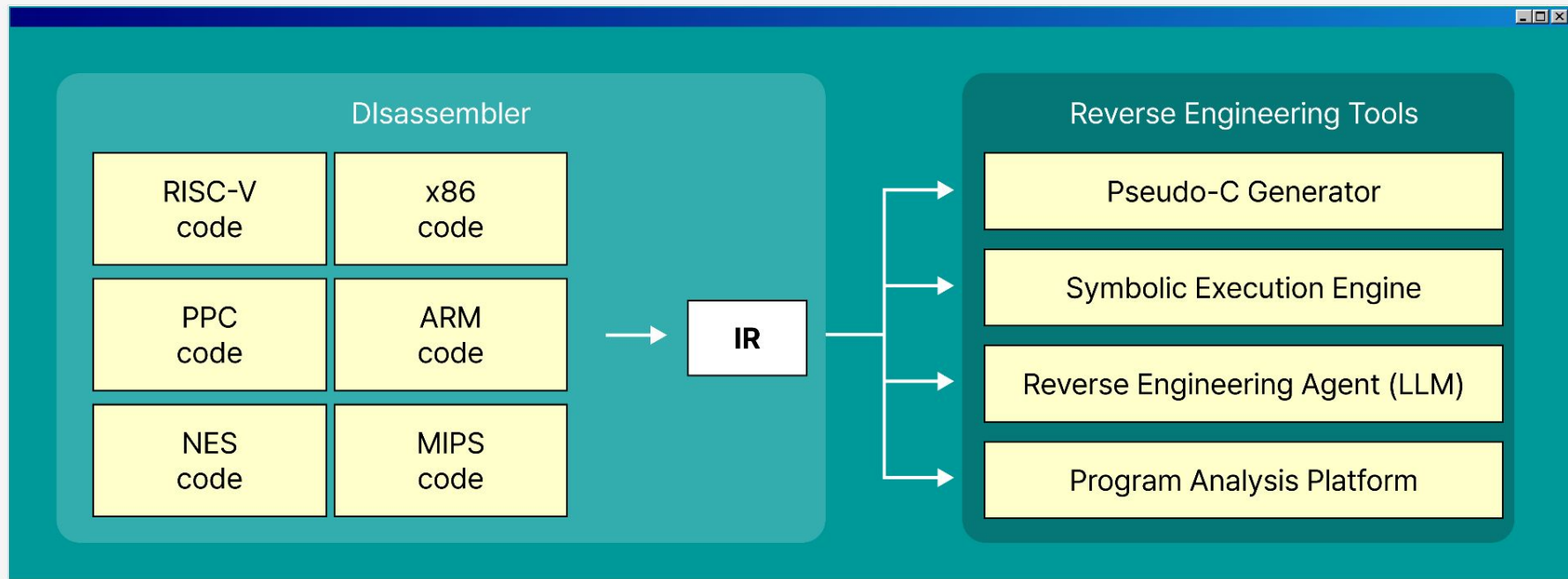
통역가 고양이가 외계어를 공부한 덕에..  
가게가 폭발하지도 않고,  
고양이 서버는 서빙에만 집중하면 됩니다.  
모두가 행복합니다.

## 중간 표현이란?



컴파일러에서 통역가 고양이의 역할을  
'Intermediate Representation'(IR)이  
수행합니다.





이러한 개념의 적용으로, 각 기계어 코드를 해석하는 방법만 디스어셈블러에게 알려주면,  
기존의 모든 리버스 엔지니어링 도구를 재활용할 수 있습니다.

# PPC VLE

# 리버스 엔지니어링

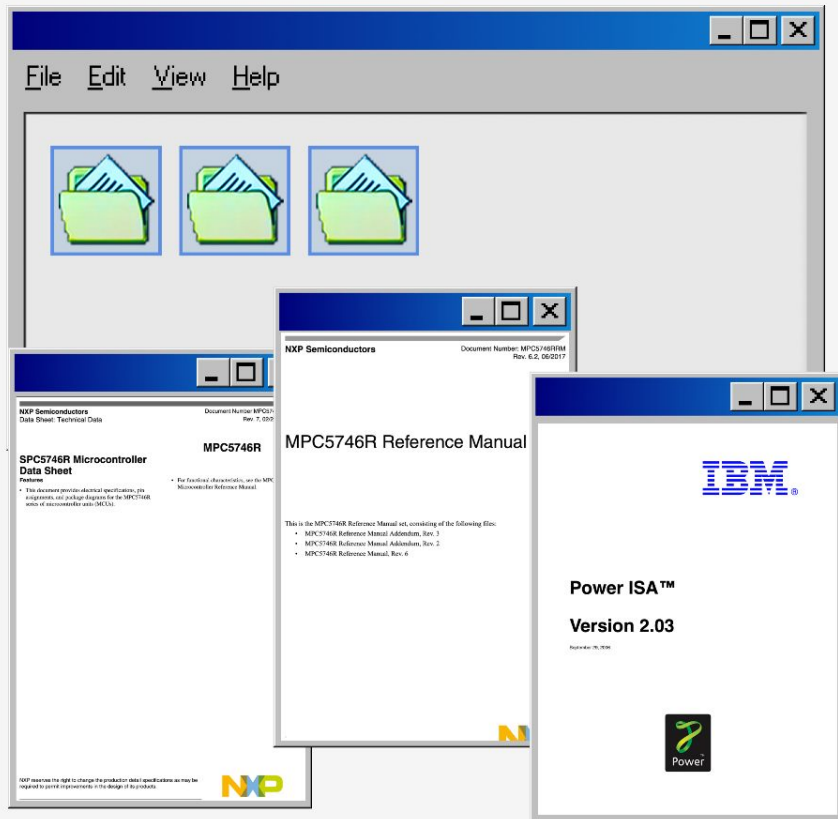
-----

아키텍처 디스어셈블러 구현을 중심으로...

옛 야생의 아키텍처가 나타났다!



# 데이터 시트 또는 아키텍처 명세 획득하기



아키텍처 또는 CPU/MCU/MPU 등에 대한  
데이터 시트, 유저 매뉴얼, 레퍼런스 매뉴얼, 명세 등에는  
아키텍처에 대한 거의 모든 정보가 포함되어 있습니다!

일반적으로 제조사의 웹사이트나 구글링하면 얻을 수 있습니다.

일부 프라이빗 매뉴얼은 서비스 회원 가입 또는 해당 제품을  
직접 구매해야 제공한다.

# 레지스터

컴퓨터 레지스터(Register)는 CPU(중앙처리장치) 내부에 있는 아주 작은 고속의 임시 저장 공간입니다.

컴퓨터의 모든 데이터 처리는 CPU가 담당하는데, 이때 필요한 데이터나 연산 결과를 즉시 사용할 수 있도록 레지스터에 잠시 보관합니다.

## 도움말

쉽게 비유하자면, 레지스터는 요리사의 '작업대'와 같습니다.

모든 재료가 저장된 '냉장고(RAM)'까지 매번 가지 않고,  
지금 당장 필요한 재료와 도구를  
작업대 위에 올려두고 빠르게 요리하는 것과 같은 원리입니다.

확인 (O)



REGISTER

# 레지스터의 종류

## 범용 레지스터 (General Purpose Registers)

이름 그대로 다양한 목적으로 자유롭게 사용할 수 있는 레지스터입니다.  
주로 데이터나 주소를 임시로 저장하는 데 쓰입니다.

---

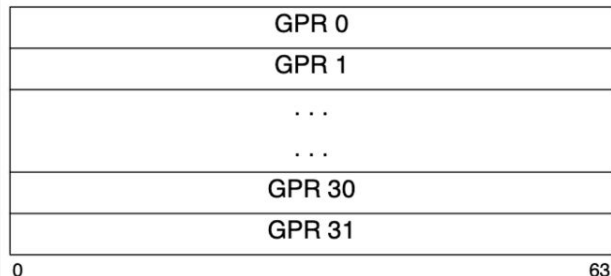
## 특수 목적 레지스터 (Special Purpose Registers)

특별한 기능이 정해져 있어 프로그래머가 직접 값을 바꾸기보다는 시스템에 의해 제어되는 경우가 많습니다.

# 레지스터 분석하기

## 3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Processor. The principal storage internal to the Fixed-Point Processor is a set of 32 General Purpose Registers (GPRs). See Figure 38.



**Figure 38. General Purpose Registers**

Each GPR is a 64-bit register.

## 2.3.2 Link Register

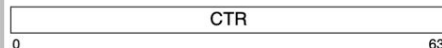
The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it holds the return address after Branch instructions for which LK=1.



**Figure 33. Link Register**

## 2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Register* instruction.



**Figure 34. Count Register**

## 2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).



**Figure 32. Condition Register**

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (*mtcrf*, *mtocrf*).
- A specified field of the CR can be set by a move to the CR from another CR field (*mcrf*), from XER<sub>32:35</sub> (*mcrxr*), or from the FPSCR (*mcrfs*).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a *Compare* instruction.

PowerPC의 여러 레지스터들

## 명령어 분석하기 - 명령어 크기

To reduce code **size**, VLE provides a 16-bit conditional branch instruction that uses the BO16 and BI16 operands. For example, the 32-bit conditional branch `e_bc 1,2,target` can be expressed using a 16-bit instruction format, `se_bc 1,2,target`. In simplified mnemonic form this becomes `se_bt eq,target`. The BO16 operand only allows testing a true or false condition, unlike the BO32 operand that also allows decrementing the CTR. The BI16 operand allows testing of only CR0, unlike the BI32 operand, which allows testing CR0–CR3.

16비트 길이 명령어도 있고, 32비트 길이 명령어도 있다!



# 명령어 분석하기 - 명령어 동작/의미

### 1.4.6 R-form (16-bit Monadic Instructions)

0	6	12	15
OPCD	XO	RX	

Figure 6. R instruction format

### 1.4.7 RR-form (16-bit Dyadic Instructions)

0	6	7	8	12	15
OPCD	XO	RY	RX		
OPCD	2	5	RY	RX	
OPCD	XO	ARY	RX		
OPCD	XO	RY	ARX		

Figure 7. RR instruction format

### 1.4.8 SD4-form (16-bit Load/Store Instructions)

0	4	8	12	15
OPCD	SD4	RZ	RX	

Figure 8. SD4 instruction format

### 1.4.9 BD15-form

0	10	12	15	31
OPCD	BO32	BI32	BD15	LK

Figure 9. BD15 instruction format

### 1.4.10 BD24-form

0	6	7	31
OPCD	0	BD24	LK

Figure 10. BD24 instruction format

### 1.4.11 D8-form

0	6	11	15	24	31
OPCD	RT	RA	XO	D8	
OPCD	RS	RA	XO	D8	

Figure 11. D8 instruction format

### 1.4.12 I16A-form

0	6	11	15	21	31
OPCD	sl	RA	XO	sl	
OPCD	ul	RA	XO	ul	

Figure 12. I16A instruction format

### 1.4.13 I16L-form

0	6	11	15	21	31
OPCD	RT	ul	XO	ul	

Figure 13. I16L instruction format

### 1.4.14 M-form

0	6	11	15	21	26	31
OPCD	RS	RA	SH	MB	ME	
OPCD	RS	RA	SH	MB	ME	

Figure 14. M instruction format

### 1.4.15 SC18-form

0	6	9	11	15	20	21	22	24	31
OPCD	RT	RA	XO	Rc	F	SCL	UI8		
OPCD	RT	RA	XO	F	SCL	UI8			
OPCD	RS	RA	XO	Rc	F	SCL	UI8		
OPCD	RS	RA	XO	F	SCL	UI8			
OPCD	000	BF32	RA	XO	F	SCL	UI8		
OPCD	001	BF32	RA	XO	F	SCL	UI8		
OPCD	XO	RA	XO	F	SCL	UI8			

Figure 15. SC18 instruction format

### 1.4.16 LI20-form

0	6	11	15	17	21	31
OPCD	RT	li20	XO	li20	li20	

Figure 16. LI20 instruction format

### 1.4.17 Instruction Fields

VLE uses instruction fields defined in Section 1.6.21 of Book 1 as well as VLE-defined instruction fields defined below.

ARX (12:15)

Field used to specify an "alternate" General Purpose Register in the range R8:R23 to be used as a destination.

명령어의 각 비트마다 의미가 다릅니다.

바이너리의 각 명령어를 비트 단위로 쪼개는 작업(파싱)을 필요로 합니다.

이런 작업들은 capstone disassembler를 활용할 때도 있습니다.



# 명령어 분석하기 - 명령어 동작/의미

**System Call**
**C-form**

se\_sc

02

0
15

```

SRR1 ←iea MSR
SRR0 ← CIA+2
NIA ←iea IVPR0:47 || IVOR848:59 || 0b0000
MSR ← new_value (see below)

```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. The contents of the MSR are copied into SRR1.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 5.6 of Book III-E.

The interrupt causes the next instruction to be fetched from effective address

IVPR<sub>0:47</sub> || IVOR8<sub>48:59</sub> || 0b0000.

This instruction is context synchronizing.

**Special Registers Altered:**  
SRR0 SRR MSR

명령어의 실제 동작 방식과 피연산자(operands)의 의미가 무엇인지 이해하고, 이를 Binary Ninja가 이해할 수 있는 형태로 의미를 전달해야 합니다.

# 어떤 IR로 바뀌야 할까요?

리버스 엔지니어링에서 사용되는 IR은 대부분 분석 플랫폼에 종속적인 경우가 많습니다.  
각 플랫폼이 고유한 분석 기능을 효과적으로 제공하기 위해 자신만의 IR을 사용하기 때문입니다.

따라서 특정 플랫폼에서 분석 작업을 하려면,  
해당 플랫폼이 제공하는 라이브러리나 API를 통해 대상 코드를 고유의 IR 형태로 변환해야 합니다.

주요 리버스 엔지니어링 플랫폼과 그들이 사용하는 IR은 다음과 같습니다.

- Ghidra: P-code
- Binary Ninja: BNIL (Binary Ninja Intermediate Language)
- angr: VEX (PyVex 라이브러리를 통해 접근)

## 도움말

이번 발표에서는 `Binary Ninja`를 활용하겠습니다.

확인 (O)



# Binary Ninja

- 학생용 라이선스는 \$74 (약 10만원)
- 사용하기 쉬운 GUI
- Binary Ninja의 알파이자 오메가: Python API
- 배우기 쉽다.
- 사용하기 쉽다.
- 확장하기 쉽다.





# Machine Code to IR - Binary Ninja 예시

[https://github.com/PetoWorks/binaryninja-power-vle/blob/a19b1cb8044e8b2cd50ad5499d7ee921ac181020/powervle/lowlevelil/move\\_sysreg.py#L4-L21](https://github.com/PetoWorks/binaryninja-power-vle/blob/a19b1cb8044e8b2cd50ad5499d7ee921ac181020/powervle/lowlevelil/move_sysreg.py#L4-L21)

```
def lift_move_sysreg_instructions(inst: Instruction, il: LowLevelILFunction) -> None:
    for i in range(len(inst.operands)):
        if i == 0: oper_0 = inst.operands[0]
        elif i == 1: oper_1 = inst.operands[1]

    if inst.name == "se_mflr":
        assert len(inst.operands) == 1
        rx = inst.get_operand_value(oper_0)

        ei0 = il.set_reg(4, rx, il.reg(4, "lr"))
        il.append(ei0)

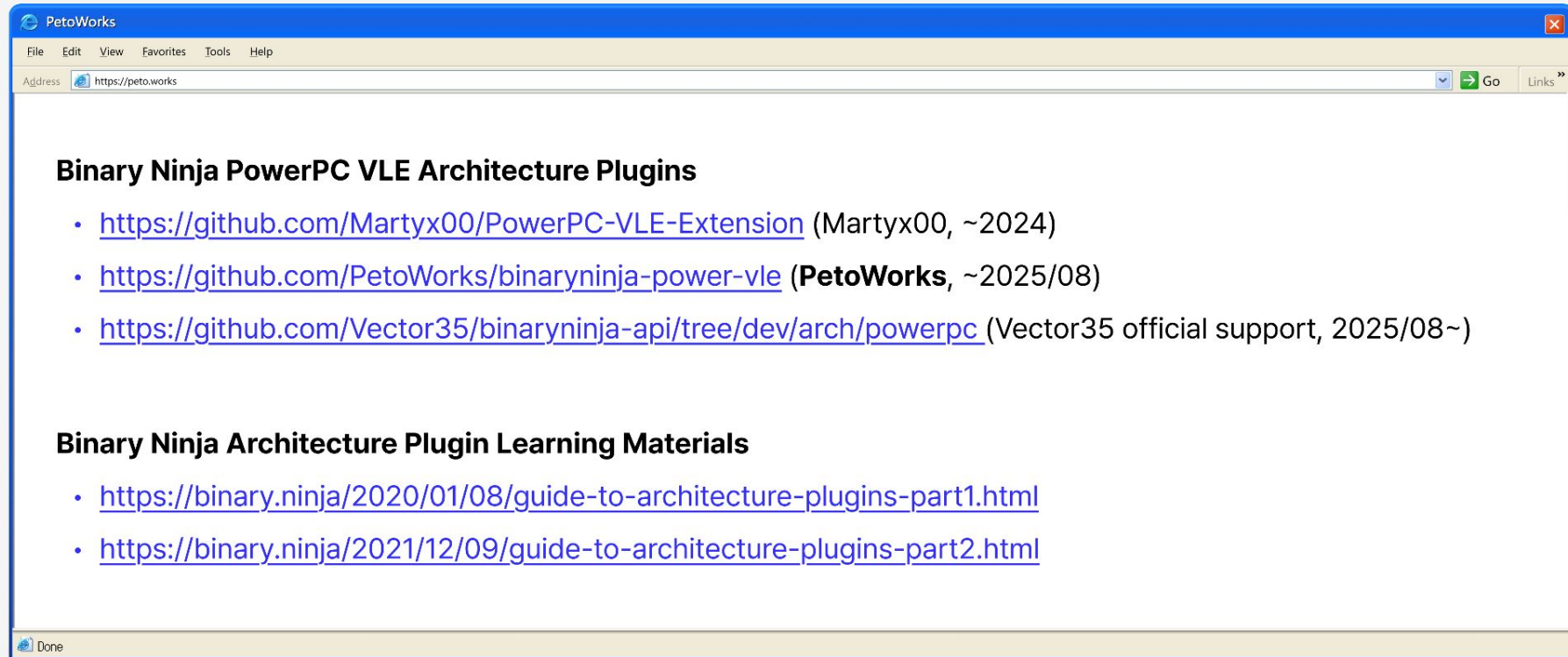
    elif inst.name == "se_mtlr":
        assert len(inst.operands) == 1
        rx = inst.get_operand_value(oper_0)

        ei0 = il.set_reg(4, "lr", il.reg(4, rx))
        il.append(ei0)
```



# Open Source!

Special thanks to 이주영 (PetoWorks), 정종배 (AutoCrypt), 이성민 (한동대학교)



**Binary Ninja PowerPC VLE Architecture Plugins**

- <https://github.com/Martyx00/PowerPC-VLE-Extension> (Martyx00, ~2024)
- <https://github.com/PetoWorks/binaryninja-power-vle> (**PetoWorks**, ~2025/08)
- <https://github.com/Vector35/binaryninja-api/tree/dev/arch/powerpc> (Vector35 official support, 2025/08~)

**Binary Ninja Architecture Plugin Learning Materials**

- <https://binary.ninja/2020/01/08/guide-to-architecture-plugins-part1.html>
- <https://binary.ninja/2021/12/09/guide-to-architecture-plugins-part2.html>

# 결과 훑쳐보기

```

000d0b8                                     06 e0 1b 81 09 10 00 80
000d0c0 d9 01 73 e8 e0 00 1f ff-14 48 cc 0f 1f df 01 34-d2 01 c3 0e 18 7f 80 38-18 a1 80 08 7c 06 00 34 ..s.....H.....4.....8....|..4
000d0e0 18 9f 80 10 68 56 1b 9f-08 00 79 ff dd 09 2a 23-48 0d e6 12 50 1f 01 30-c2 3f 64 10 ....hV....y...*#H...P..0.?.d.

000d0fc      int32_t sub_d0fc(int32_t arg1 @ r0)

000d0fc 541f0130 r1winm r31, r0, 0x0, 0x4, 0x18
000d100 4810483d bl      0x11193c
000d104 341f0078 addic. r0, r31, 0x78
000d108 79ffdce1 ??

000d10c                                     c2 4f 18 7c-80 08 79 ff d6 37 90 de-1b 81 08 10 c9 01 00 90 .0.|..y..7.....
000d120 21 f1 00 04 18 21 06 f0-1b c1 09 08 00 80 d5 01-73 e8 e0 00 1f ff 14 48-c3 3f 2a 03 1b df 80 08 !....!.....s.....H.?*....
000d140 e6 04 c2 4f 79 ff d6 05-c2 3f 18 9e 80 08 79 ff-d5 fb c2 ff 01 f3 79 ff-d4 ad 48 60 b6 3f d4 0f ...0y....?....y.....y..H`.?..
000d160 1b c1 08 08 c5 01 00 90-20 f1 00 04 70 e8 e0 00-30 07 15 92 18 00 e0 01-7c 03 00 34 68 53 00 04 .....p..0.....|..4hS..

```

이전

```

00fa90f4 501f0130 e_lwz r0, 304(r31)
00fa90f8 c23f se_lwz r3, 8(r15)
00fa90fa 6410 se_bseti r0, 0x1
00fa90fc 541f0130 e_stw r0, 304(r31)
00fa9100 4810 se_li r0, 0x1
00fa9102 483d se_li r13, 0x3
00fa9104 341f0078 e_stb r0, 120(r31)
00fa9108 79ffdce1 e_bl sub_fa6de8
00fa910c c24f se_lwz r4, 8(r15)
00fa910e 187c8008 e_addi r3, r28, 0x8
00fa9112 79ffd637 e_bl j_sub_fa653c

00fa9116 90de se_stb r13, 0(r14)
00fa9118 1b810810 e_lmw r28, 16(r1)
00fa911c c901 se_lwz r0, 36(r1)

```

```

00fa8d60      int32_t sub_fa8d60(int32_t arg1 @ r14, void* arg2 @ r15, uint32_t arg3 @ r29)

00fa8d60 {
00fa8d60     uint32_t var_c = arg3;
00fa8d74     *(uint32_t*)0x40001588;
00fa8d78     int32_t r30 = *(uint32_t*)0x40001544;
00fa8d7c     __builtin_clz(arg3);
00fa8d96     int32_t r13;
00fa8d96     *(uint8_t*)((char*)arg2 + 8) =
00fa8d96         sub_fa94cc(sub_fa7098(arg1, 0), 0x40001454, r13 >> 5, arg2, r30);
00fa8da6     int32_t result = j_sub_fa96c6(sub_fa7098(arg1, 1), 0, 0x40001454, 8);
00fa8daa     sub_fa71d2(result);
00fa8db8     return result;
00fa8d60 }

```

이후

(디스어셈블리, Pseudo-C)

# Tricore

# 리버스 엔지니어링

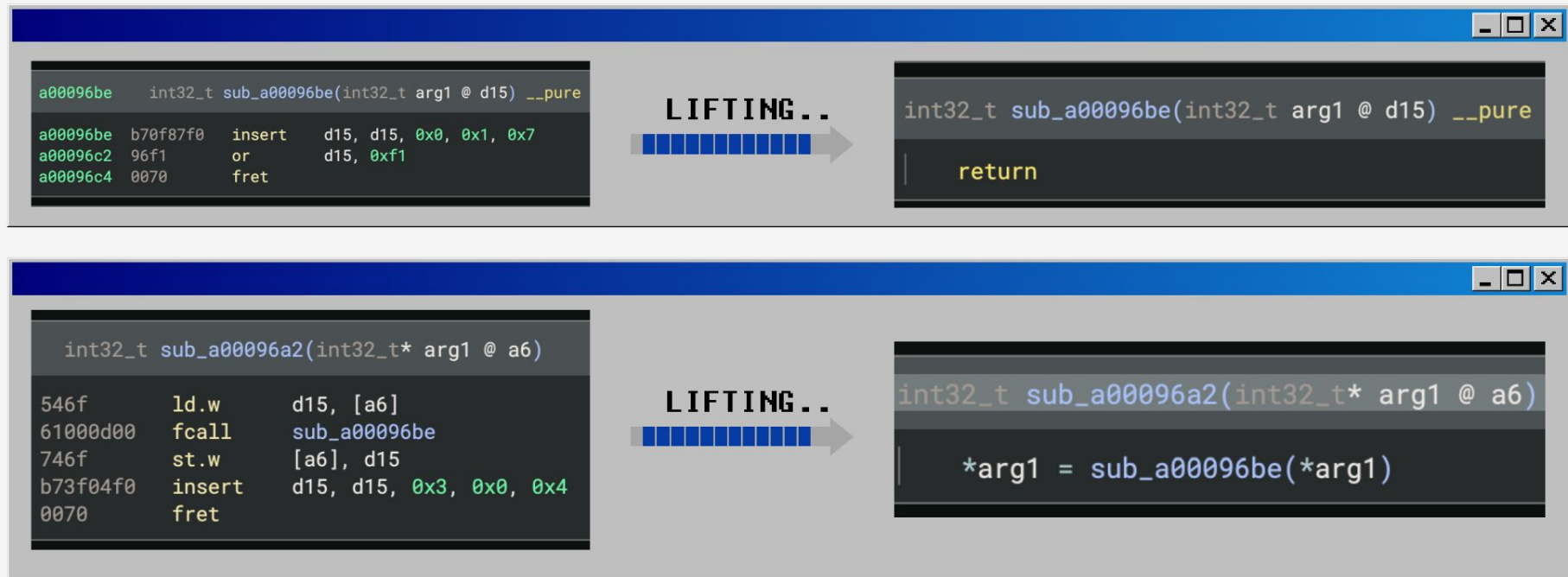
---

리버스 엔지니어링 자동화를 중심으로

옛 마샬의 마키텡처가 나타났다!



# TriCore - Fast Function Calls



어떤 함수들에 대한 해석이 이상하다..!

# TriCore - Fast Function Calls

```

a00096be  int32_t sub_a00096be(int32_t arg1 @ d15) __pure
a00096be  b70f87f0  insert   d15, d15, 0x0, 0x1, 0x7
a00096c2  96f1      or       d15, 0xf1
a00096c4  0070      fret
    
```

PetoWorks
File Edit View Favorites Tools Help
Address <https://peto.works> Go Links

## 4.6 Fast Function Calls with FCALL/FRET

In situations where the saving and restoring of the upper context registers is not required an FCALL instruction may be used in preference to a CALL. The FCALL instruction performs a call jump and in parallel saves the current return address (A11) to the stack. No other state is saved. The called function therefore starts execution with the same context as the caller (with the exception of A10 and A11).

To return from a function called by an FCALL, an FRET instruction is executed. This performs a jump to the current return address (A11) and loads the previous A11 back from the stack. No other state is loaded. The caller function therefore resumes execution with a context modified by the called function. The calling and called functions must co-operate on the use of all registers.

Done

뭔가 새로운 calling convention...!  
그리고 Binary Ninja는  
이를 제대로 해석하지 못하고 있습니다.



# Calling Convention

아키텍처에서 함수(subroutine)를 호출 하는 방법에 대한 정의 (약속!)

예를 들어, ARM 아키텍처는...

- 함수 호출 이전,
  - r0 ~ r3 레지스터에 순서대로 함수의 인자 값을 할당
  - 컨텍스트 관리 (return address, caller/callee saved registers) 등
- 함수 호출 이후, r0, r1 레지스터로 함수 반환 값을 조회
- ...



PROMISE

# Calling Convention 예시

```

1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int ret = callee(0x41, 0x42, 0x43, 0x44);
7      printf("%d\n", ret);
8      return 0;
9  }
10

```

COMPILE

```

movs    r3, #68
movs    r2, #67
movs    r1, #66
movs    r0, #65
bl      callee(int, int, int, int)
mov     r3, r0
str     r3, [r7, #12]

```

[godbolt.org](http://godbolt.org), ARM GCC trunk

# Calling Convention 예시

```

1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int ret = callee(0x41, 0x42, 0x43, 0x44);
7      printf("%d\n", ret);
8      return 0;
9  }
10

```

COMPILE

```

movs    r3, #68
movs    r2, #67
movs    r1, #66
movs    r0, #65
bl       callee(int, int, int, int)
mov     r3, r0
str     r3, [r7, #12]

```

[godbolt.org](http://godbolt.org), ARM GCC trunk

함수 호출 전, 인자 설정

# Calling Convention 예시

```

1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int ret = callee(0x41, 0x42, 0x43, 0x44);
7      printf("%d\n", ret);
8      return 0;
9  }
10

```

COMPILE

```

movs    r3, #68
movs    r2, #67
movs    r1, #66
movs    r0, #65
bl      callee(int, int, int, int)
mov     r3, r0
str     r3, [r7, #12]

```

[godbolt.org](http://godbolt.org), ARM GCC trunk

함수 호출 후, 반환 값 저장



# 컨텍스트

```
1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int variable = 0xdeadbeef;
7      printf("variable = %d\n", variable);
8      int ret = callee(0x41, 0x42, 0x43, 0x44);
9      printf("variable is still %d\n", variable);
10     return ret;
11 }
```

## QUESTION

Q. 'variable'은 'callee()' 호출 후에도  
그 값이 0xdeadbeef일까요?



# 컨텍스트

```
1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int variable = 0xdeadbeef;
7      printf("variable = %d\n", variable);
8      int ret = callee(0x41, 0x42, 0x43, 0x44);
9      printf("variable is still %d\n", variable);
10     return ret;
11 }
```

## QUESTION

Q. 'variable'은 'callee()' 호출 후에도  
그 값이 0xdeadbeef일까요?

YES

NO



# 컨텍스트

```
1  #include <stdio.h>
2
3  extern int callee(int, int, int, int);
4
5  int caller(char* str) {
6      int variable = 0xdeadbeef;
7      printf("variable = %d\n", variable);
8      int ret = callee(0x41, 0x42, 0x43, 0x44);
9      printf("variable is still %d\n", variable);
10     return ret;
11 }
```

## 해설

caller() 함수는 어떤 함수를 호출하고 코드 흐름을 callee에게 넘기게 되더라도 callee의 반환 후, 다시 caller의 코드를 실행해야 합니다.

그리고 callee 호출 이전의 프로그램 상태를 일부 저장해야만 합니다.

확인 (O)



# TriCore - Fast Function Calls

```
a00096be  int32_t sub_a00096be(int32_t arg1 @ d15) __pure
a00096be  b70f87f0  insert  d15, d15, 0x0, 0x1, 0x7
a00096c2  96f1      or      d15, 0xf1
a00096c4  0070      fret
```

PetoWorks

File Edit View Favorites Tools Help

Address <https://peto.works> Go Links »

## 4.6 Fast Function Calls with FCALL/FRET

In situations where the saving and restoring of the upper context registers is not required an FCALL instruction may be used in preference to a CALL. The FCALL instruction performs a call jump and in parallel saves the current return address (A11) to the stack. No other state is saved. The called function therefore starts execution with the same context as the caller (with the exception of A10 and A11).

To return from a function called by an FCALL, an FRET instruction is executed. This performs a jump to the current return address (A11) and loads the previous A11 back from the stack. No other state is loaded. The caller function therefore resumes execution with a context modified by the called function. The calling and called functions must co-operate on the use of all registers.

Done

하지만 TriCore의 Fast Call Function은...

이전 함수의 컨텍스트를  
어딘가에 백업하는 것이 아니라,  
이전 함수의 컨텍스트와 동일한 컨텍스트에서  
실행되는 함수입니다.

# TriCore - Fast Function Calls 해석하기

Binary Ninja가 상위 컨텍스트에서 정상적으로 fast call을 해석하게 만들고

Fast Calling Function 함수들의 전체 코드에 대한 data-flow analysis를 정상적으로 수행하게 만들어야 합니다.



# TriCore - Fast Function Calls 해석하기

```

a00096be  int32_t sub_a00096be(int32_t arg1 @ d15) __pure
a00096be  b70f87f0  insert  d15, d15, 0x0, 0x1, 0x7
a00096c2  96f1      or      d15, d15, 0xf1
a00096c4  0070      fret
  
```

```

int32_t sub_a00096a2(int32_t* arg1 @ a6)
546f      ld.w      d15, [a6]
61000d00  fcall     sub_a00096be
746f      st.w      [a6], d15
b73f04f0  insert   d15, d15, 0x3, 0x0, 0x4
0070      fret
  
```

SET | USE

결론적으로, 상위 컨텍스트에 어떤 레지스터가 사용되고, 어떤 레지스터의 값이 바뀌게 되는지를 전달해야 합니다.

# TriCore - Fast Function Calls 해석하기

SET

USE

```

a00096be  int32_t sub_a00096be(int32_t arg1 @ d15) __pure
a00096be  b70f87f0  insert d15, d15, 0x0, 0x1, 0x7
a00096c2  96f1      or d15, 0xf1
a00096c4  0070
    
```

'Clobbered Registers', 'Return Registers' 프로퍼티를 각 set / use 레지스터로 수정하면...

# TriCore - Fast Function Calls 해석하기

```
int32_t sub_a00096be(int32_t arg1 @ d15) __pure
|
return
```



```
int32_t sub_a00096be(int32_t arg1 @ d15) __pure
|
return (arg1 & 0xffffffff01) | 0xf1
```

```
int32_t sub_a00096a2(int32_t* arg1 @ a6)
|
*arg1 = sub_a00096be(*arg1)
```



```
a00096a2  int32_t sub_a00096a2(int32_t* arg1 @ a6)
a00096a4      int32_t d15 = sub_a00096be(*arg1)
a00096a8      *arg1 = d15
a00096ae      return (d15 & 0xffffffff0) | 3
```

Binary Ninja의 data-flow analysis engine이 반환 값과 관련된 코드를  
dead-code로 처리하지 않아, 위와 같이 정상적으로 해석이 됩니다.

# Fast Function Call 패턴

```
>>> for func in bv.functions:
...     for block in func:
...         for instruction in block:
...             toks, size = instruction
...             if toks[0].text == "fret":
...                 func.name = f"fastcall_{func.start:08x}"
```

펌웨어의 모든 fastcall 함수들의 이름을 알아보기 쉽게 바꿔주는 Binary Ninja 코드입니다.

이를 활용해 fastcall 함수의 패턴을 식별 해 보겠습니다.

# Fast Function Call 패턴

fastcall_a0086e20	0x0a0086e20
fastcall_a0086e34	0x0a0086e34
fastcall_a0086e36	0x0a0086e36
fastcall_a0086e42	0x0a0086e42
fastcall_a0086e4c	0x0a0086e4c
fastcall_a0086e5a	0x0a0086e5a
sub_a0086e68	0x0a0086e68
sub_a008795a	0x0a008795a
sub_a0087a08	0x0a0087a08
sub_a0087a16	0x0a0087a16
sub_a0087a20	0x0a0087a20
sub_a0087a2c	0x0a0087a2c
sub_a0087a36	0x0a0087a36
sub_a0087a4c	0x0a0087a4c
sub_a0087a56	0x0a0087a56
sub_a0087a76	0x0a0087a76
j_sub_a0059156	0x0a0087ae4
sub_a0087ae8	0x0a0087ae8
sub_a0087af4	0x0a0087af4
sub_a0087b02	0x0a0087b02
sub_a0087b0e	0x0a0087b0e
fastcall_a0087b1a	0x0a0087b1a
fastcall_a0087b28	0x0a0087b28
fastcall_a0087b32	0x0a0087b32
fastcall_a0087b3c	0x0a0087b3c
sub_a0087b4e	0x0a0087b4e
sub_a0087b7a	0x0a0087b7a
sub_a0087b8a	0x0a0087b8a
sub_a0087ba8	0x0a0087ba8
sub_a0087bc0	0x0a0087bc0

선형의 주소상에 연속적인 fastcall 함수가 식별된 후,  
일반 함수가 식별됩니다.

이를 해석 해 보겠습니다.

# Fast Function Call 패턴

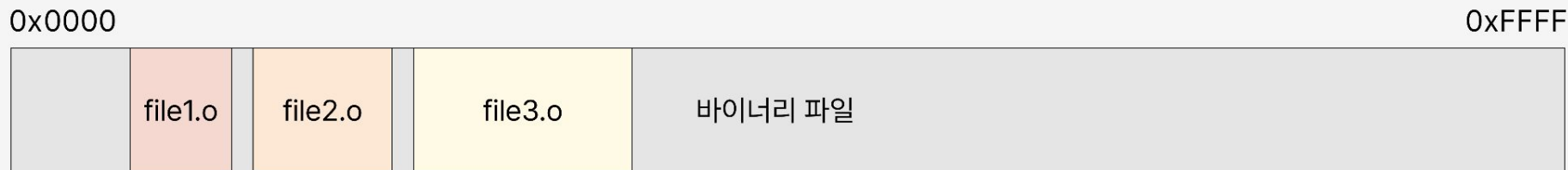


```
$ meson build -v -C builddir
[1/163] cc -o libflashrom.1.dylib.p/en29lv640b.c.o -c ../en29lv640b.c
[2/163] cc -o libflashrom.1.dylib.p/printlock.c.o -c ../printlock.c
[3/163] cc -o libflashrom.1.dylib.p/helpers.c.o -c ../helpers.c
...
```

여러 소스코드를 컴파일하고 빌드하는 과정에서, 컴파일 과정은 파일 단위로 이루어집니다.

그리고 이는 바이너리에서 몇가지 특징을 가지게 됩니다.

# Fast Function Call 패턴



대부분의 경우 컴파일된 여러 오브젝트 파일이 한개의 바이너리로 합쳐질 때  
 그 단위가 바뀌지 않기 때문에, 연속적인 주소에 같은 파일 또는 기능 단위의 코드가 포함 될 가능성이 높습니다.

# Fast Function Call 패턴

```
int32_t fastcall_a0089c8e(int32_t arg1 @ d15, void* arg2 @ a15)
```

```
    int32_t d0 = *(arg2 + 1)
    return d0, d0 & 0x80, arg1
```

```
int32_t fastcall_a0089c98()
```

```
    return 0x70007f1c
```

```
int32_t fastcall_a008cf9a(int32_t arg1 @ d4, int32_t arg2 @ d5)
```

```
    int32_t d15 = arg2 * 0x14
    return 0x7000846c + d15 + (arg1 << 1), d15
```

또한 fastcall 함수들은 컴파일 단계에서,  
반복적인 연산에 대한 저장 공간 최적화에 사용되는 것으로 보입니다.

# Fast Function Call 패턴

fastcall_a0086e20	0x0a0086e20
fastcall_a0086e34	0x0a0086e34
fastcall_a0086e36	0x0a0086e36
fastcall_a0086e42	0x0a0086e42
fastcall_a0086e4c	0x0a0086e4c
fastcall_a0086e5a	0x0a0086e5a
sub_a0086e68	0x0a0086e68
sub_a008795a	0x0a008795a
sub_a0087a08	0x0a0087a08
sub_a0087a16	0x0a0087a16
sub_a0087a20	0x0a0087a20
sub_a0087a2c	0x0a0087a2c
sub_a0087a36	0x0a0087a36
sub_a0087a4c	0x0a0087a4c
sub_a0087a56	0x0a0087a56
sub_a0087a76	0x0a0087a76
j_sub_a0059156	0x0a0087ae4
sub_a0087ae8	0x0a0087ae8
sub_a0087af4	0x0a0087af4
sub_a0087b02	0x0a0087b02
sub_a0087b0e	0x0a0087b0e
fastcall_a0087b1a	0x0a0087b1a
fastcall_a0087b28	0x0a0087b28
fastcall_a0087b32	0x0a0087b32
fastcall_a0087b3c	0x0a0087b3c
sub_a0087b4e	0x0a0087b4e
sub_a0087b7a	0x0a0087b7a
sub_a0087b8a	0x0a0087b8a
sub_a0087ba8	0x0a0087ba8
sub_a0087bc0	0x0a0087bc0

이런 정보들을 바탕으로 우리는 이 패턴을...

# Fast Function Call 패턴

Section (or File)  
#1

Section (or File)  
#2

```
fastcall_a0086e20      0x0a0086e20
fastcall_a0086e34      0x0a0086e34
fastcall_a0086e36      0x0a0086e36
fastcall_a0086e42      0x0a0086e42
fastcall_a0086e4c      0x0a0086e4c
fastcall_a0086e5a      0x0a0086e5a
sub_a0086e68           0x0a0086e68
sub_a008795a           0x0a008795a
sub_a0087a08           0x0a0087a08
sub_a0087a16           0x0a0087a16
sub_a0087a20           0x0a0087a20
sub_a0087a2c           0x0a0087a2c
sub_a0087a36           0x0a0087a36
sub_a0087a4c           0x0a0087a4c
sub_a0087a56           0x0a0087a56
sub_a0087a76           0x0a0087a76
j_sub_a0059156          0x0a0087ae4
sub_a0087ae8           0x0a0087ae8
sub_a0087af4           0x0a0087af4
sub_a0087b02           0x0a0087b02
sub_a0087b0e           0x0a0087b0e
fastcall_a0087b1a      0x0a0087b1a
fastcall_a0087b28      0x0a0087b28
fastcall_a0087b32      0x0a0087b32
fastcall_a0087b3c      0x0a0087b3c
sub_a0087b4e           0x0a0087b4e
sub_a0087b7a           0x0a0087b7a
sub_a0087b8a           0x0a0087b8a
sub_a0087ba8           0x0a0087ba8
sub_a0087bc0           0x0a0087bc0
```

이런 방식으로 해석할 수 있었습니다.

# Fast Function Call 패턴

Section (or File)  
#1

```
fastcall_a0086e20      0x0a0086e20
fastcall_a0086e34      0x0a0086e34
fastcall_a0086e36      0x0a0086e36
fastcall_a0086e42      0x0a0086e42
fastcall_a0086e4c      0x0a0086e4c
fastcall_a0086e5a      0x0a0086e5a
sub_a0086e68           0x0a0086e68
sub_a008795a           0x0a008795a
sub_a0087a08           0x0a0087a08
sub_a0087a16           0x0a0087a16
sub_a0087a20           0x0a0087a20
sub_a0087a2c           0x0a0087a2c
sub_a0087a36           0x0a0087a36
sub_a0087a4c           0x0a0087a4c
sub_a0087a56           0x0a0087a56
sub_a0087a76           0x0a0087a76
j_sub_a0059156          0x0a0087ae4
sub_a0087ae8           0x0a0087ae8
sub_a0087af4           0x0a0087af4
sub_a0087b02           0x0a0087b02
sub_a0087b0e           0x0a0087b0e
```

fastcall 함수가 섹션 단위 초반에 위치하고,  
그 뒤에 위치하는 일반 함수들은 fastcall을 호출합니다.  
이 분석을 바탕으로 실제로 저희 프로젝트를  
다방면으로 확장 시킬 수 있었습니다 :)

(이해하면 무서운 이야기 #1: 이 펌웨어에는 문자열도, 심볼도 없고 오직 코드뿐입니다...👹)

(이해하면 무서운 이야기 #2: 이 프로젝트를 7개월 넘게 진행하고 있습니다...👹)

# 끝내며..

앗! 야생의 아키텍처가 나타났다!





# Reverse Engineering Tips

## 1. 만들고 해체하기:

직접 코드를 만들고 컴파일하고 리버싱 해 보는것도 공부가 됩니다.



# Reverse Engineering Tips

1. 만들고 해체하기

**2. 변수의 자료형 명시하기:**

변수의 자료형을 명시할 수 있을 때 해 놓으세요.

이것으로 리버싱 도구들이 프로그램을 더 잘 이해하게 만들고 여러분에게 도움을 줄 확률을 높입니다.

# Reverse Engineering Tips

1. 만들고 해체하기

2. 변수의 자료형 명시하기

**3. 목표 설정 - 정보 수집 - 가설 설정 - 관측/증명 사이클:**

분석을 구체화하고 내가 아는 것과 모르는 것을 명확하게 만들어 줍니다.

# Reverse Engineering Tips

1. 만들고 해체하기
2. 변수의 자료형 명시하기
3. 목표 설정 - 정보 수집 - 가설 설정 - 관측/증명 사이클

## 4. 문서화하기:

새로 알게 된 사실에 대해서, 특정 기능 단위 분석 완료 시 등 여러분이 편하다고 느낄 때 언제든지 틈틈히 문서화 하는 것으로 중복되는 작업을 피하고 시간을 아낄 수 있습니다.

# Thank you!

PetoWorks, @d0now\_kim

앗! 야생의 아키텍처가 나타났다!