black hat
EUROPE 2020

DECEMBER 9-10
BRIEFINGS

Jack-in-the-Cache: a new code injection technique through modifying X86-to-ARM translation cache

Ko Nakagawa @ FFRI Security, Inc.
Hiromitsu Oshiba @ FFRI Security, Inc.

About us

Ko Nakagawa

- Security researcher at FFRI Security, Inc.
  - https://github.com/kohnakagawa
  - https://linkedin.com/in/koh-nakagawa

Maxwell's
Equations
$\nabla \cdot B = 0$
$\nabla \cdot D = \rho$
$\nabla \times E = -\frac{\partial B}{\partial t}$
$\nabla \times H = \frac{\partial D}{\partial t} + J$

Hiromitsu Oshiba

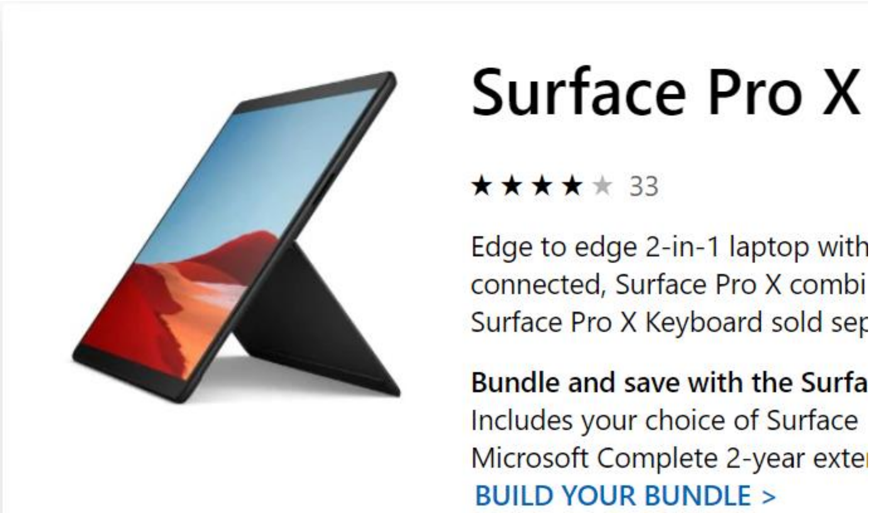
- Research director at FFRI Security, Inc.
  - https://github.com/0x75960
  - https://linkedin.com/in/hiromitsu-oshiba-072576ab

Agenda

- Introduction to Windows 10 on ARM
- Binary translation cache file
- New code injection technique
- Use-cases
- Conclusion

ARM-based laptops

Windows 10 on ARM



Surface Pro X

★★★★☆ 33

Edge to edge 2-in-1 laptop with
connected, Surface Pro X combi
Surface Pro X Keyboard sold sep

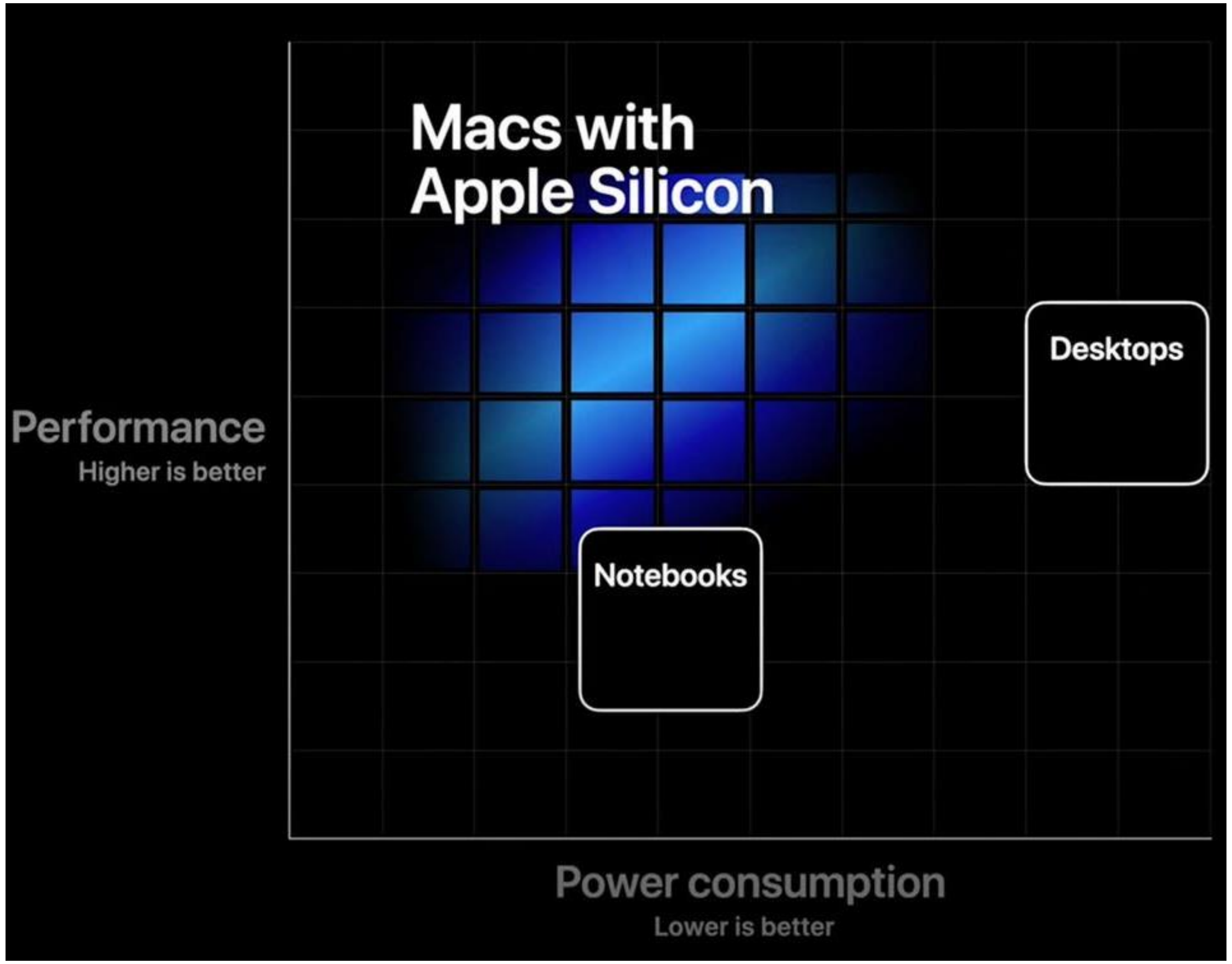
Bundle and save with the Surfa
Includes your choice of Surface
Microsoft Complete 2-year exte

BUILD YOUR BUNDLE >

https://www.microsoft.com/en-us/p/surface-pro-x/8vdnrp2m6hhc?activetab=overview

https://www8.hp.com/us/en/campaigns/hp-envy-x2/overview.html

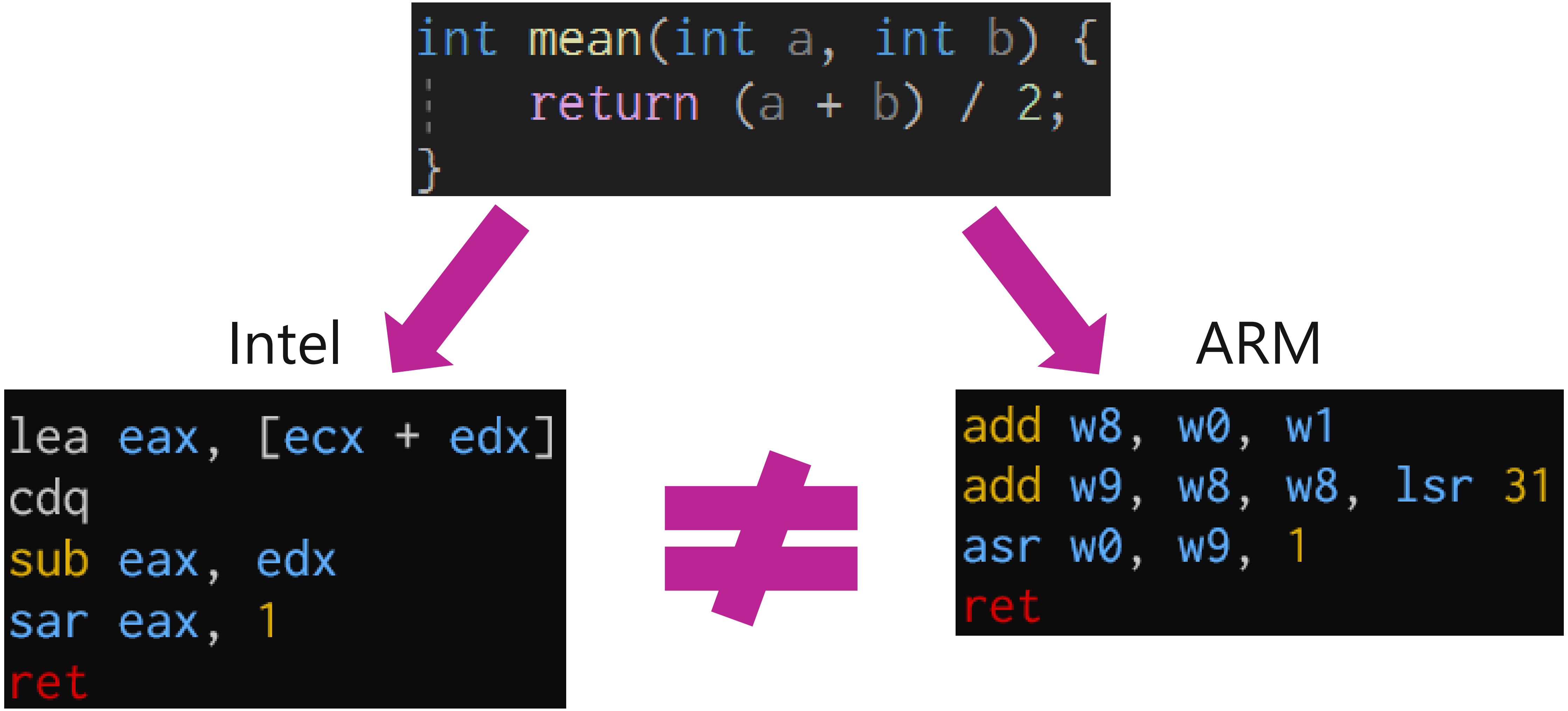
macOS on ARM-based Apple Silicon



https://www.youtube.com/watch?v=GEZhD3J89ZE

Difficulty in transition from Intel to ARM

We cannot use existing software for Intel on ARM-based laptops



```c
int mean(int a, int b) {
    return (a + b) / 2;
}
```

Intel

```
lea eax, [ecx + edx]
cdq
sub eax, edx
sar eax, 1
ret
```

≠

ARM

```
add w8, w0, w1
add w9, w8, w8, lsr 31
asr w0, w9, 1
ret
```

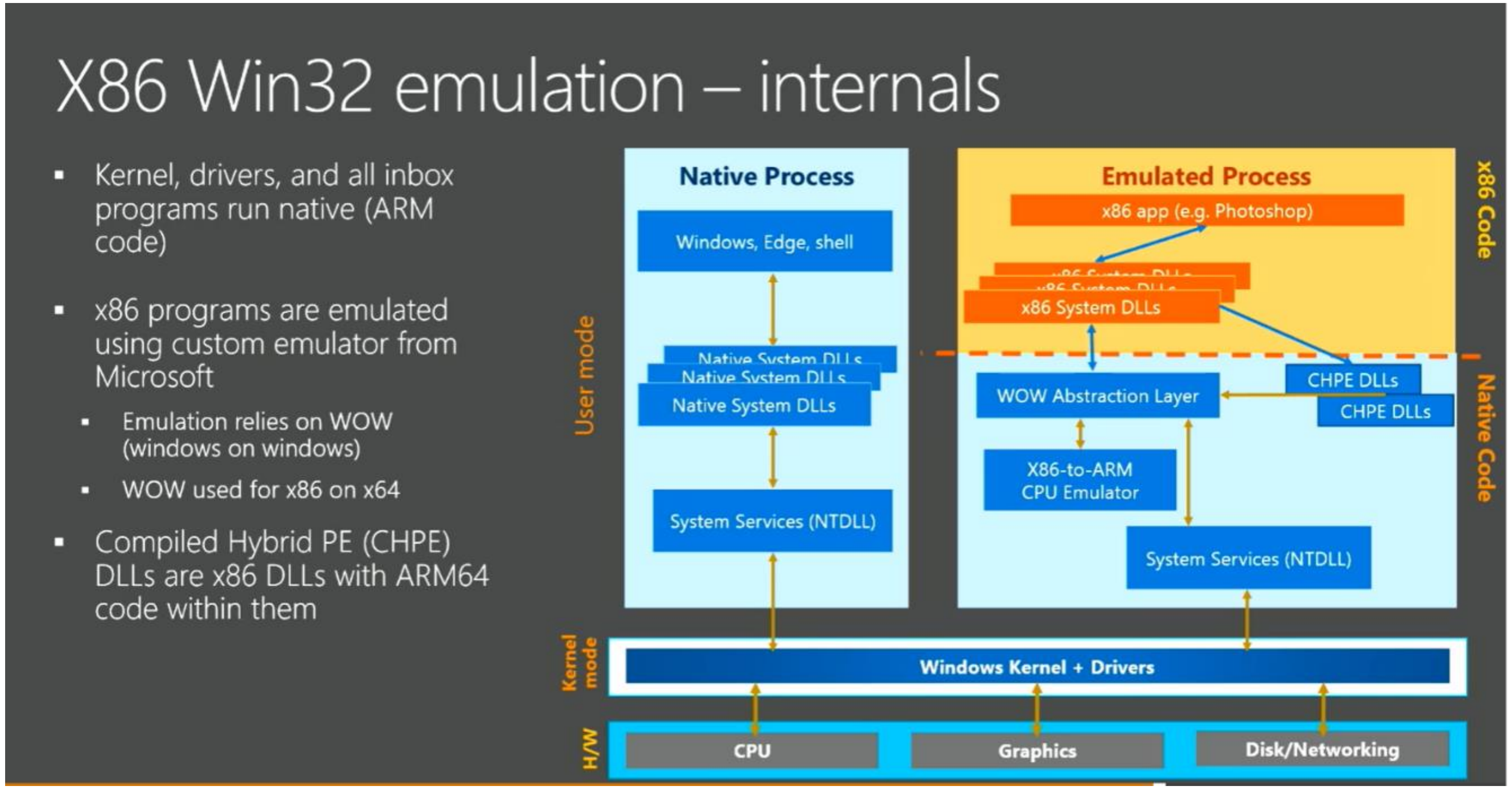
Solutions

Windows 10 on ARM

- x86 Win32 emulation
  - JIT binary translation

macOS Big Sur

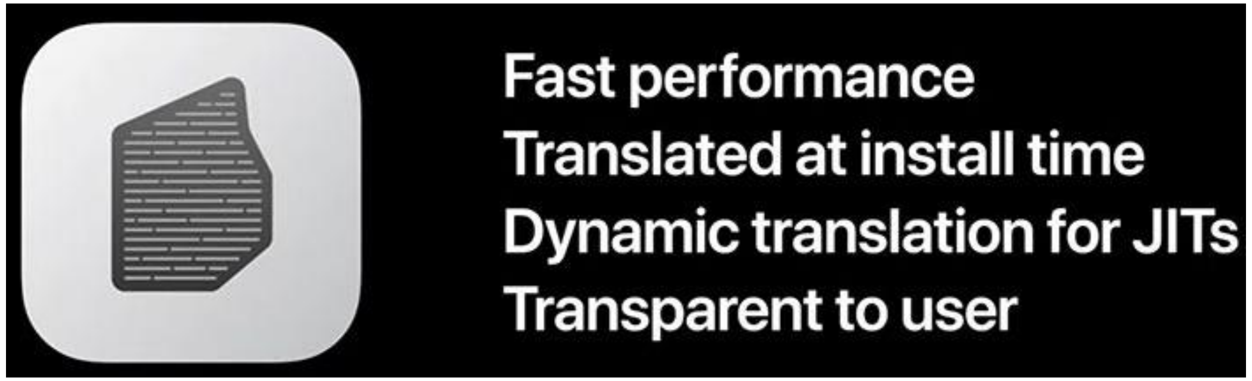
- Rosetta 2
  - binary translation at install time
  - JIT binary translation



X86 Win32 emulation – internals

- Kernel, drivers, and all inbox programs run native (ARM code)
- x86 programs are emulated using custom emulator from Microsoft
  - Emulation relies on WOW (windows on windows)
  - WOW used for x86 on x64
- Compiled Hybrid PE (CHPE) DLLs are x86 DLLs with ARM64 code within them

https://channel9.msdn.com/Events/Build/2017/P4171



Fast performance
Translated at install time
Dynamic translation for JITs
Transparent to user

https://www.youtube.com/watch?v=GEZhD3J89ZE

Hmm? Binary translation? It seems to be very slow.

🤔

Solution in Windows 10 on ARM

x86 emulation works by **compiling blocks of x86 instructions into ARM64 instructions** with optimizations to improve performance. **A service caches these translated blocks of code to reduce the overhead of instruction translation** and allow for optimization when the code runs again.

https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation

Translated blocks of code are cached as a file

X86-To-ARM64 (XTA) cache file

Reduces much of JIT binary translation overhead

- JIT binary translation is not performed when the translation result exists in an XTA cache file

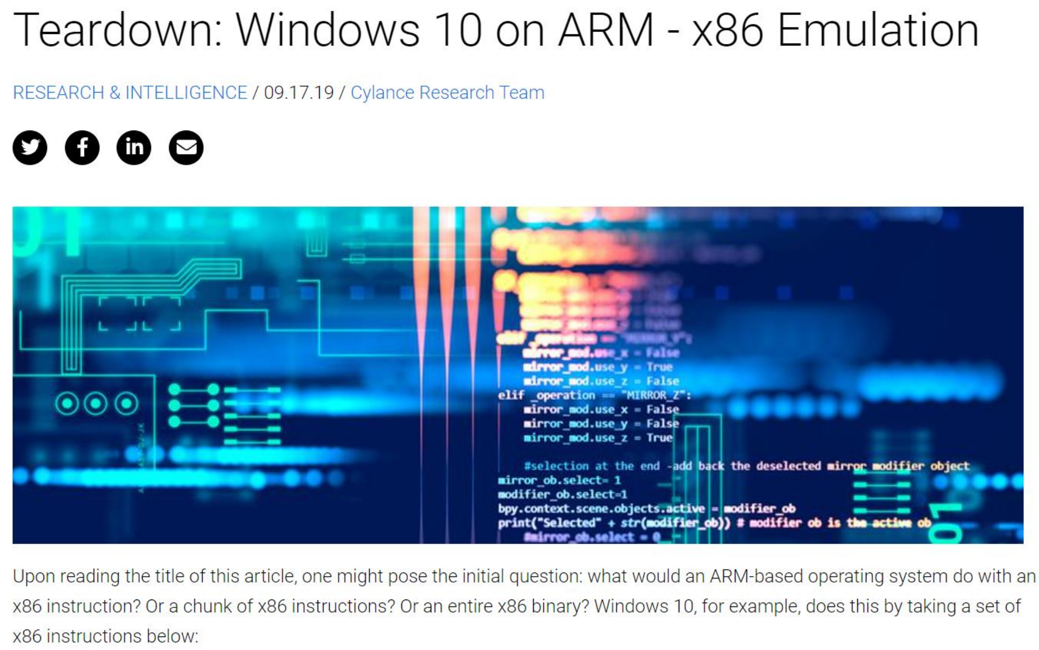
⇒ Improves the performance of x86 emulation

x86 emulation internals

Three components of x86 emulation

- xtajit.dll
  - x86 emulator DLL loaded by WOW64 layer
- xtac.exe
  - Compiler that creates/modifies XTA cache files
- XtaCache.exe
  - Service managing XTA cache files
  - It creates/modifies XTA cache files by running xtac.exe

Related work: Cylance Research team blog



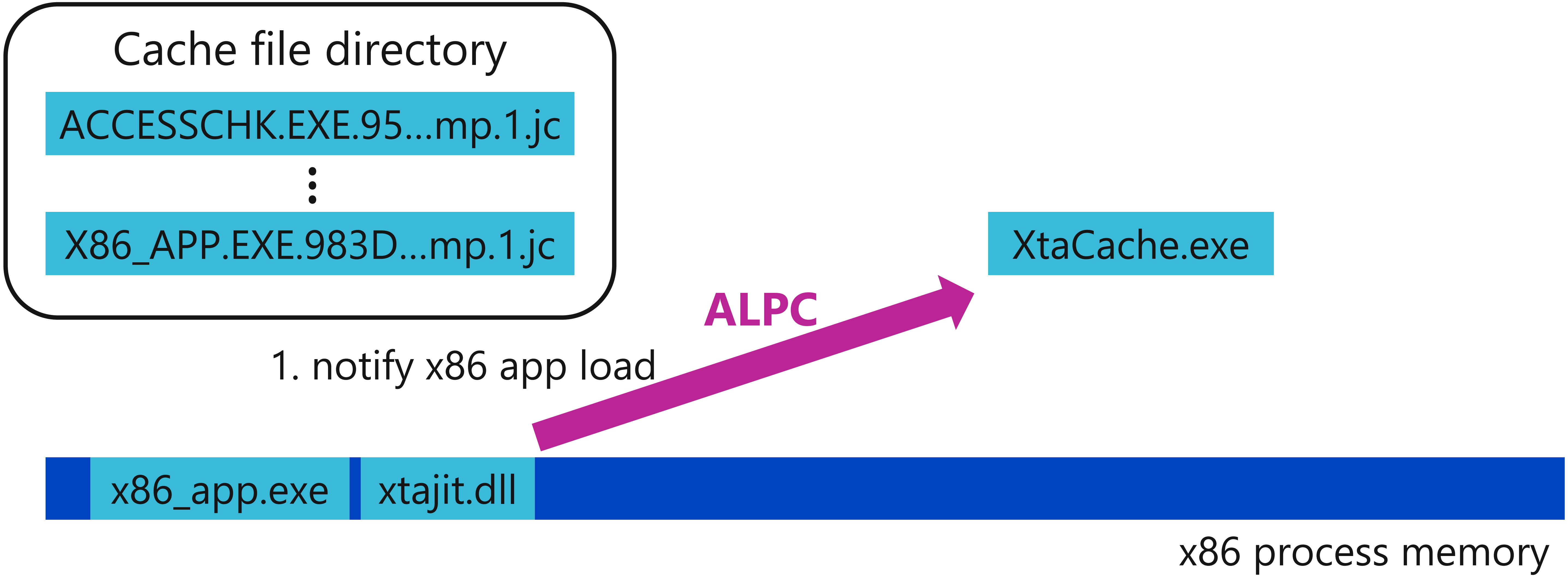
Teardown: Windows 10 on ARM - x86 Emulation

RESEARCH & INTELLIGENCE / 09.17.19 / Cylance Research Team

Upon reading the title of this article, one might pose the initial question: what would an ARM-based operating system do with an x86 instruction? Or a chunk of x86 instructions? Or an entire x86 binary? Windows 10, for example, does this by taking a set of x86 instructions below:

https://blogs.blackberry.com/en/2019/09/teardown-windows-10-on-arm-x86-emulation

Flow of execution using XTA cache file



Cache file directory

ACCESSCHK.EXE.95…mp.1.jc

⋮

X86_APP.EXE.983D…mp.1.jc

XtaCache.exe

ALPC

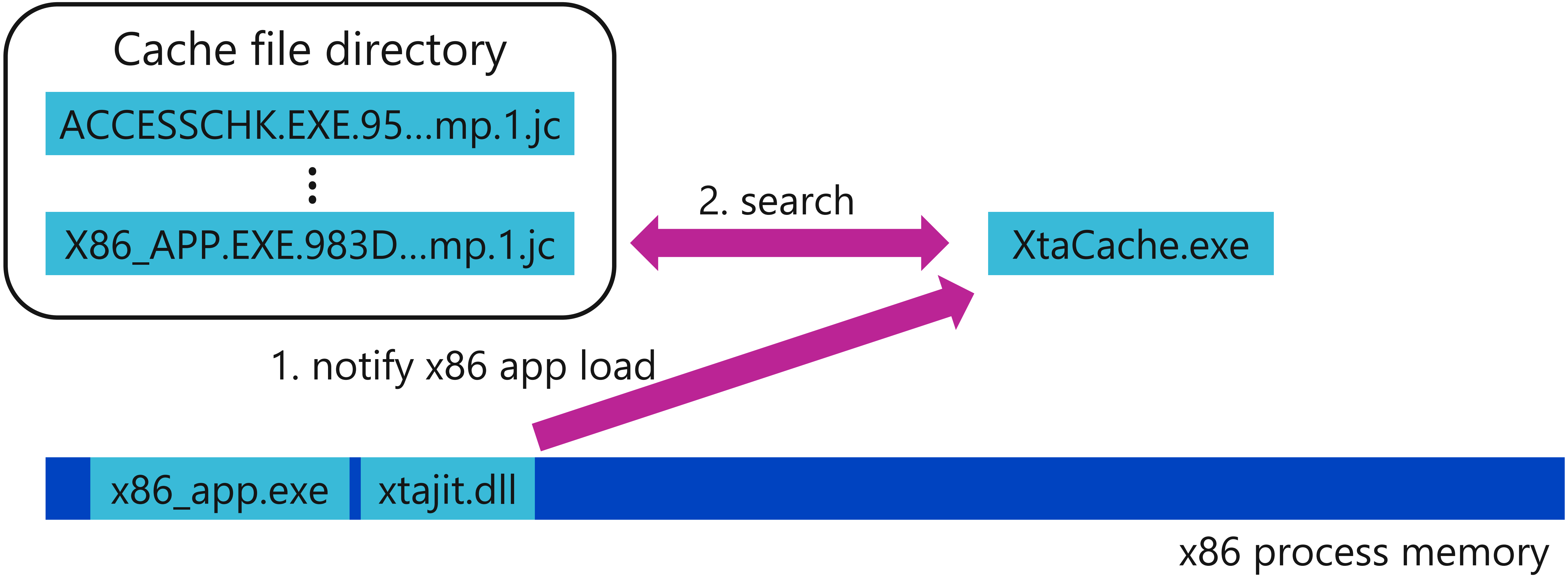
1. notify x86 app load

x86_app.exe

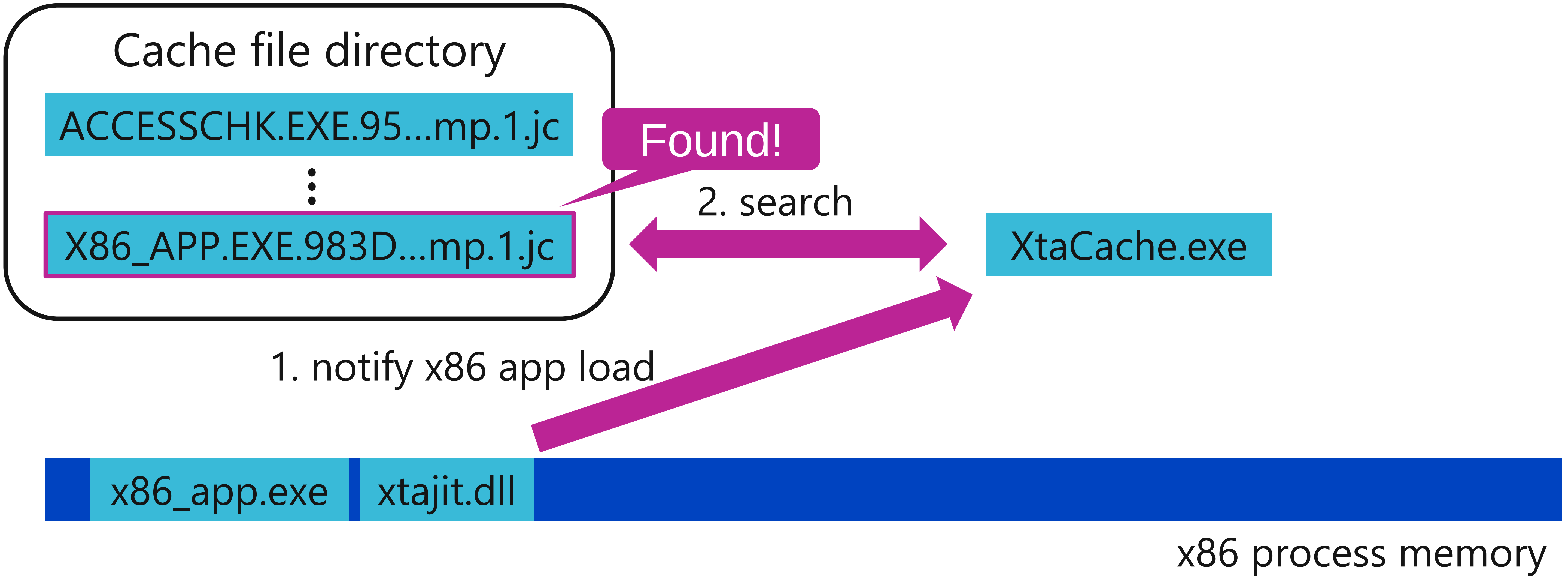
xtajit.dll

x86 process memory

Flow of execution using XTA cache file



Flow of execution using XTA cache file



Cache file directory

ACCESSCHK.EXE.95…mp.1.jc

⋮

X86_APP.EXE.983D…mp.1.jc

Found!

2. search

XtaCache.exe

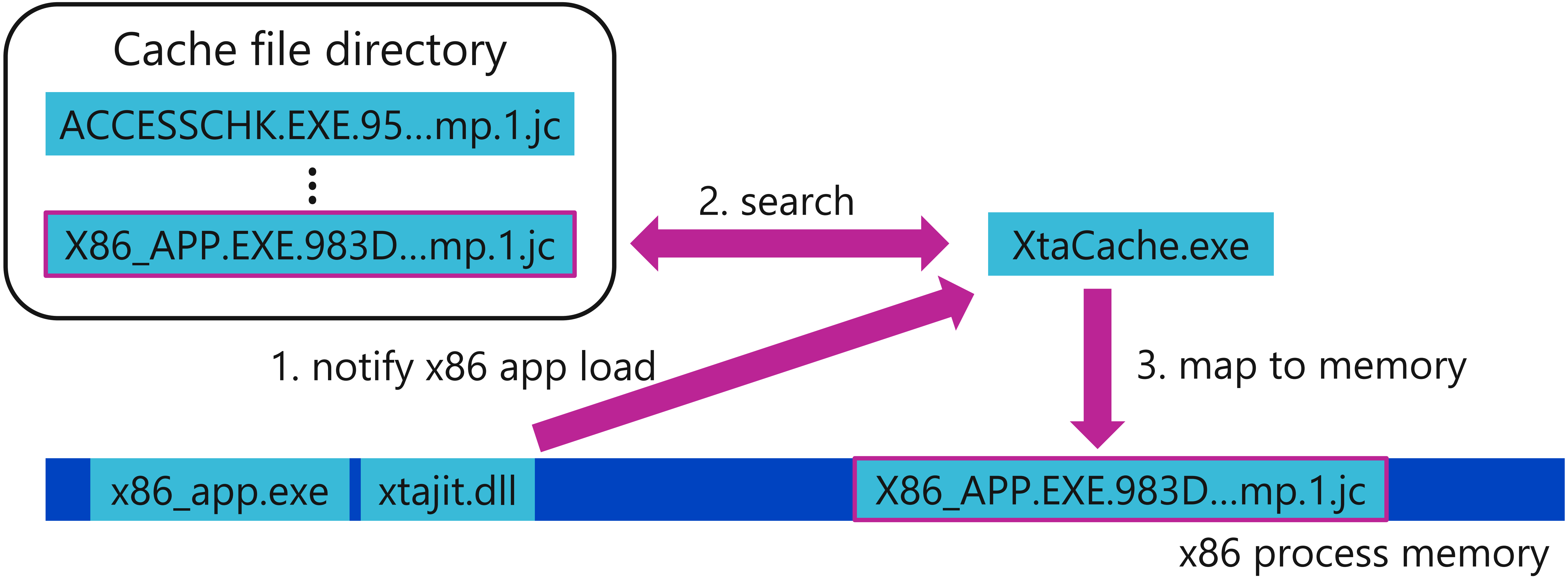
1. notify x86 app load

x86_app.exe

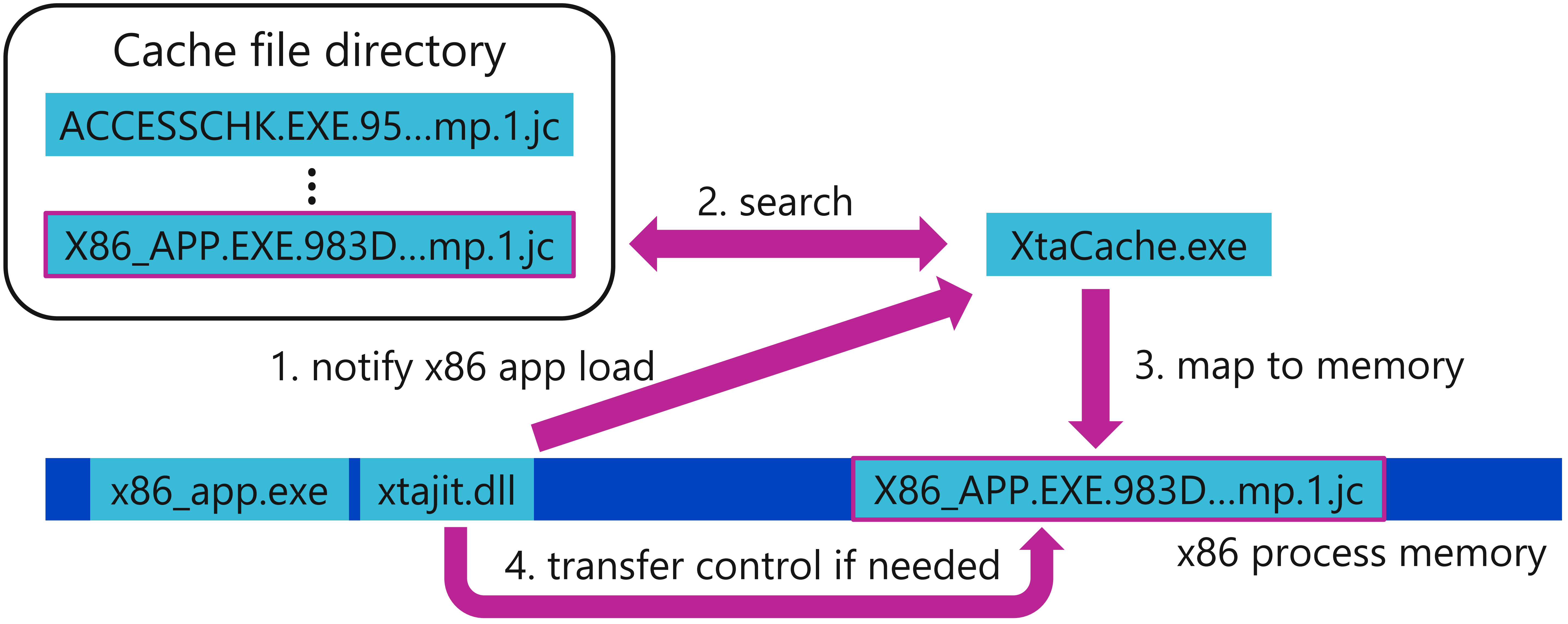
xtajit.dll

x86 process memory

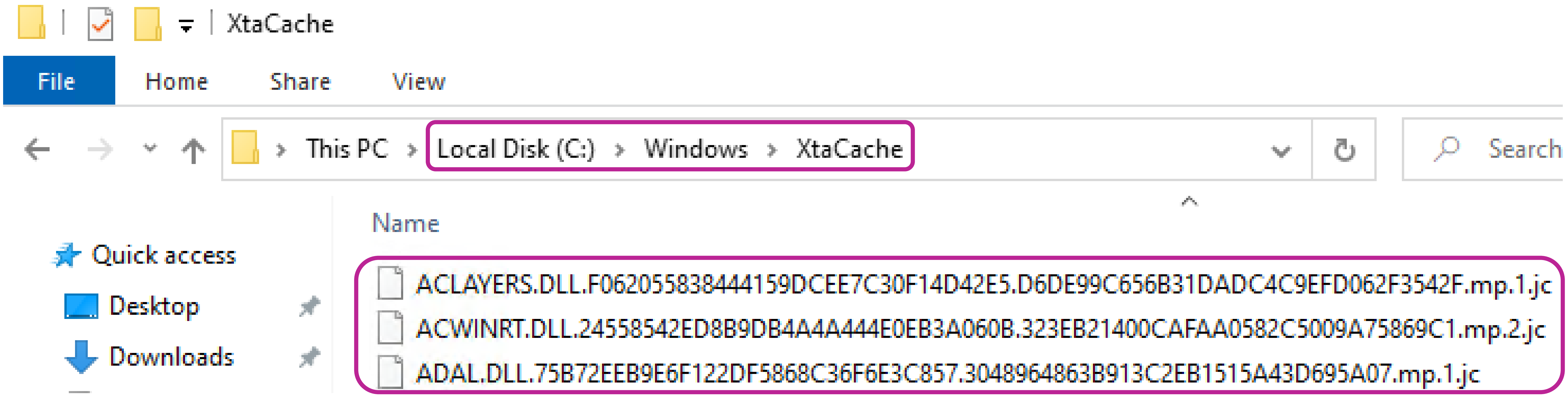
Flow of execution using XTA cache file



Flow of execution using XTA cache file



Where are XTA cache files?



Name

ACLAYERS.DLL.F062055838444159DCEE7C30F14D42E5.D6DE99C656B31DADC4C9EFD062F3542F.mp.1.jc
ACWINRT.DLL.24558542ED8B9DB4A4A444E0EB3A060B.323EB21400CAFAA0582C5009A75869C1.mp.2.jc
ADAL.DLL.75B72EEB9E6F122DF5868C36F6E3C857.3048964863B913C2EB1515A43D695A07.mp.1.jc

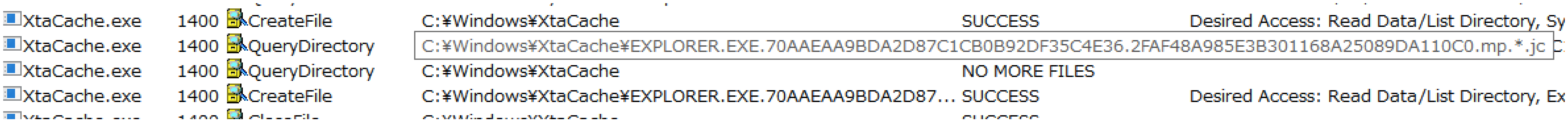
By default, full permission is granted only to XtaCache.exe
However, it can be changed with admin-equivalent privilege

Name of XTA cache file (SysWOW64¥explorer.exe)

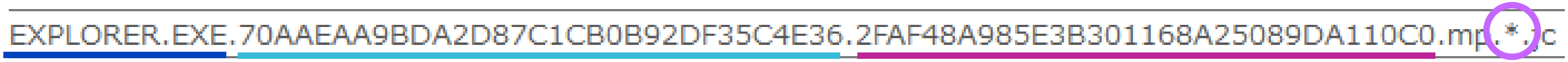
EXPLORER.EXE.70AAEAA9BDA2D87C1CB0B92DF35C4E36.2FAF48A985E3B301168A25089DA110C0.mp.**1**.jc

- **Name of x86 exe or dll (“explorer.exe” in this case)**
- **Hash value determined by file content**
- **Hash value determined by file path**
- **Number of updates of this XTA cache file**
  - xtac.exe updates an XTA cache file during/after emulation to add newly translated blocks of code (explained later)

How does XtaCache.exe search XTA cache files?



```
XtaCache.exe    1400 CreateFile       C:¥Windows¥XtaCache                                                                             SUCCESS          Desired Access: Read Data/List Directory, Sy
XtaCache.exe    1400 QueryDirectory   C:¥Windows¥XtaCache¥EXPLORER.EXE.70AAEAA9BDA2D87C1CB0B92DF35C4E36.2FAF48A985E3B301168A25089DA110C0.mp.*.jc C:
XtaCache.exe    1400 QueryDirectory   C:¥Windows¥XtaCache                                                                             NO MORE FILES
XtaCache.exe    1400 CreateFile       C:¥Windows¥XtaCache¥EXPLORER.EXE.70AAEAA9BDA2D87...                                             SUCCESS          Desired Access: Read Data/List Directory, Ex
```



EXPLORER.EXE.70AAEAA9BDA2D87C1CB0B92DF35C4E36.2FAF48A985E3B301168A25089DA110C0.mp.*.jc

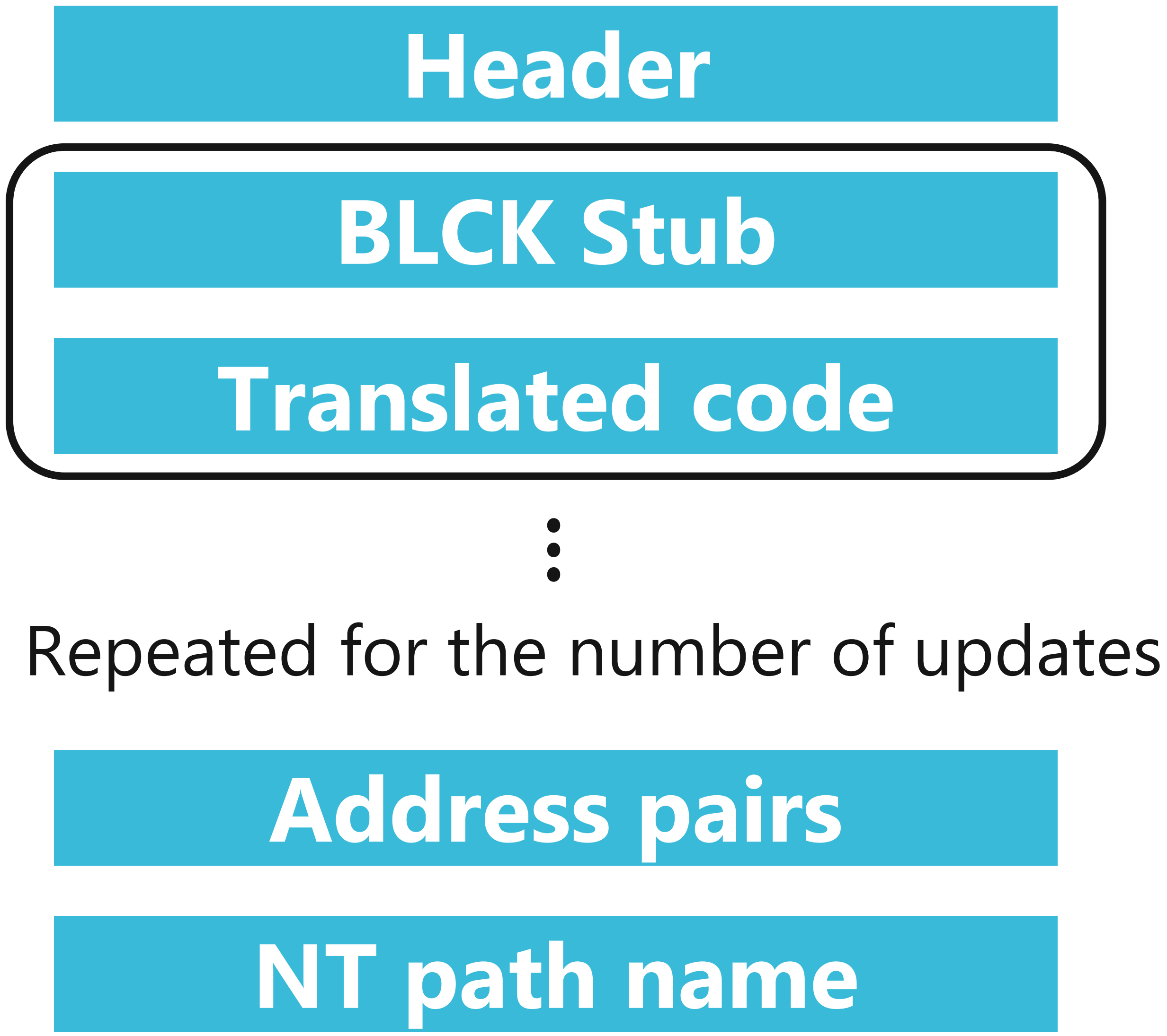
Searches cache files by **file name** · **file content** · **file path**

- **Number of updates** is specified as wildcard

Uses cache file whose number of updates is largest

- Does not use the cache files whose number of updates is smaller
  - These files are removed later

Structure of XTA cache file



Header

BLCK Stub

Translated code

⋮

Repeated for the number of updates

Address pairs

NT path name

Header holding offsets to the following blocks

Code for (1) bridging between XTA cache and xtajit.dll, (2) address lookup operation, and so on

Translated ARM64 code

Address pairs holding the relation between the RVAs of x86 app and the offsets of translated code

NT path name of x86 app

Structure of XTA cache file

Header

BLCK Stub

Translated code

Repeated for the number of updates

Address pairs

NT path name

Header

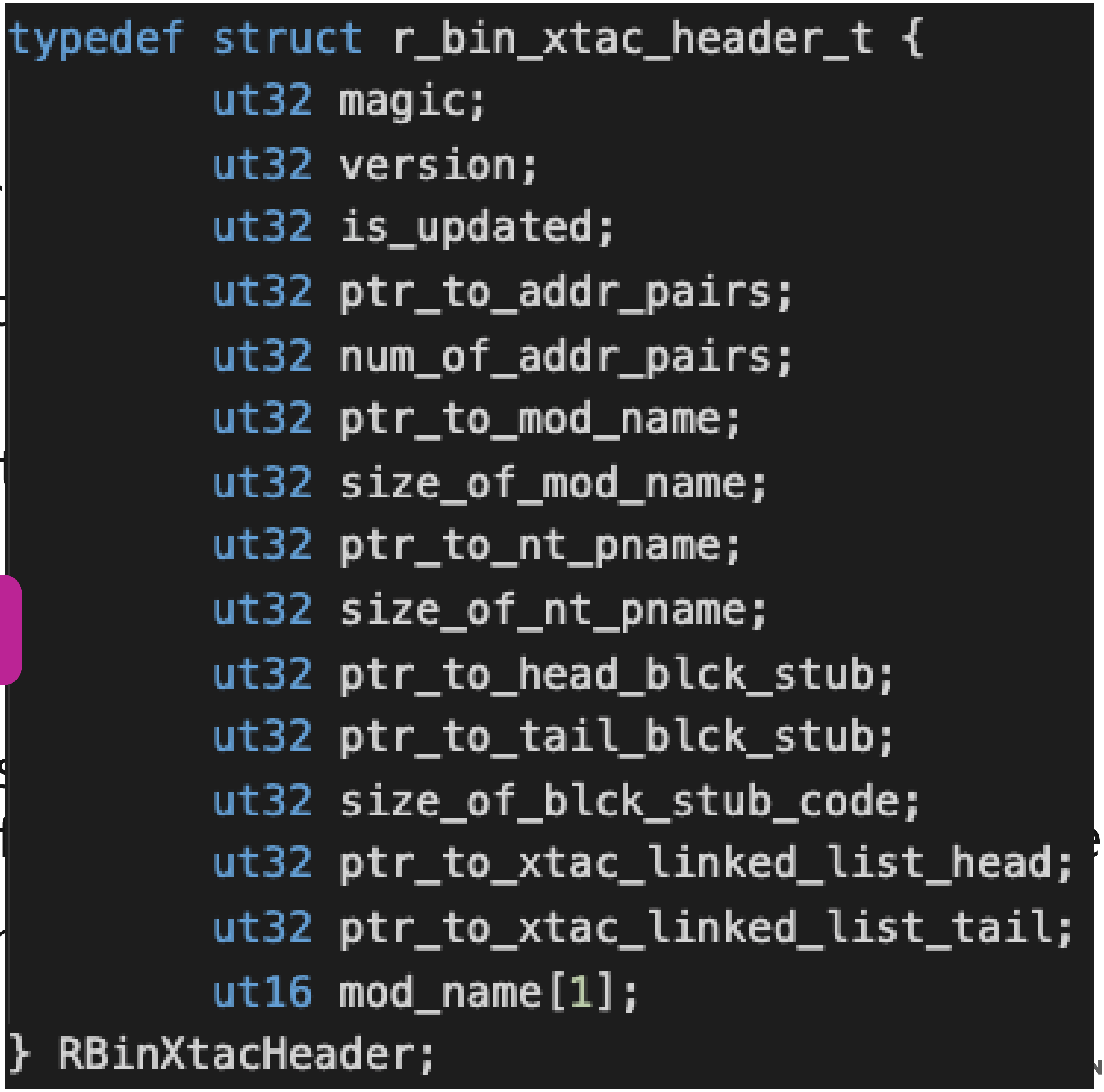
Code fo
xtajit.dl

Translat

Address
RVAs of

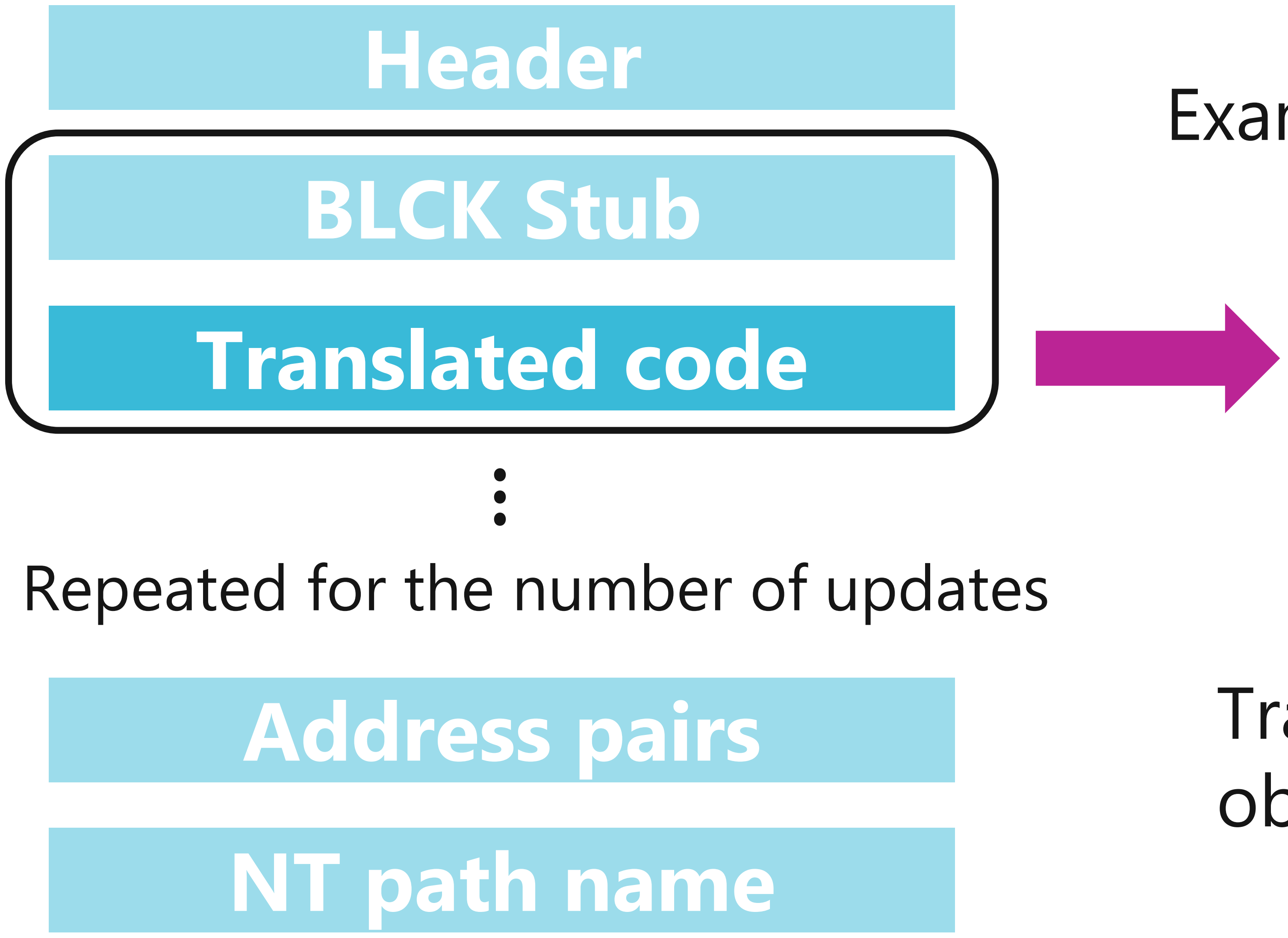
NT path

See appendix for more details

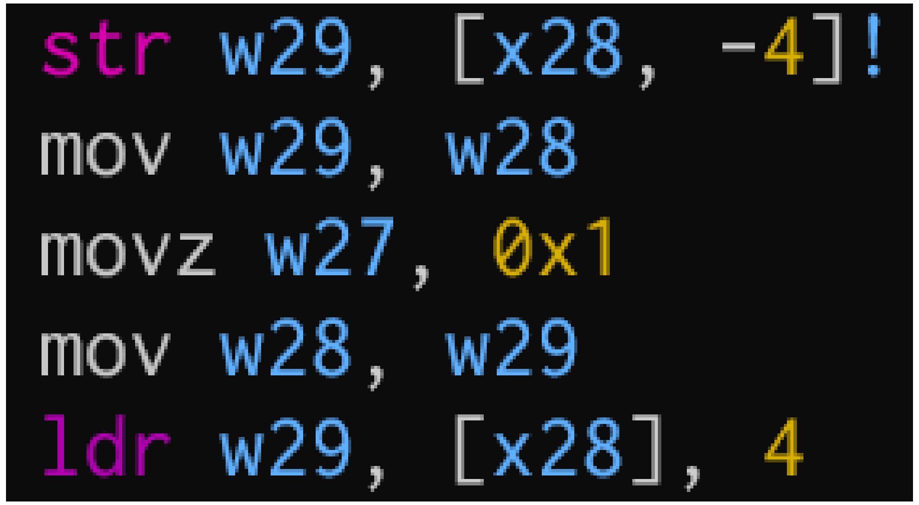


```
typedef struct r_bin_xtac_header_t {
        ut32 magic;
        ut32 version;
        ut32 is_updated;
        ut32 ptr_to_addr_pairs;
        ut32 num_of_addr_pairs;
        ut32 ptr_to_mod_name;
        ut32 size_of_mod_name;
        ut32 ptr_to_nt_pname;
        ut32 size_of_nt_pname;
        ut32 ptr_to_head_blck_stub;
        ut32 ptr_to_tail_blck_stub;
        ut32 size_of_blck_stub_code;
        ut32 ptr_to_xtac_linked_list_head;
        ut32 ptr_to_xtac_linked_list_tail;
        ut16 mod_name[1];
} RBinXtacHeader;
```

Structure of XTA cache file



Example of translated ARM64 assembly



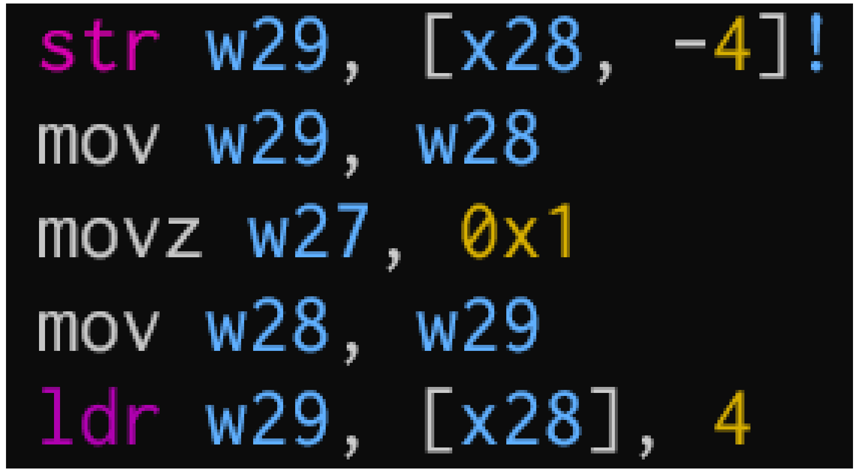
```
str w29, [x28, -4]!
mov w29, w28
movz w27, 0x1
mov w28, w29
ldr w29, [x28], 4
```

Translated code exists without obfuscation and encryption

ARM64 general-purpose register during emulation

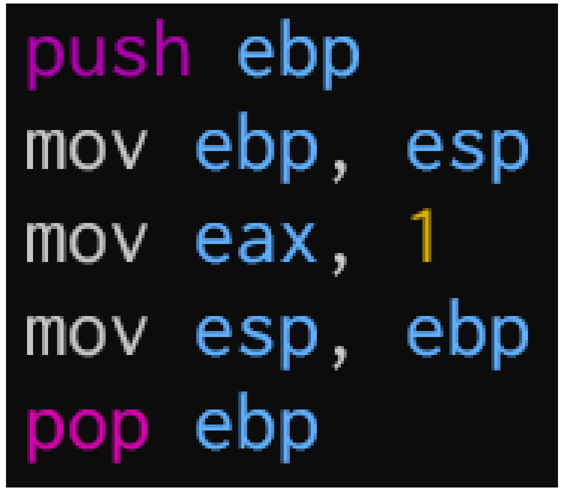
x86	ARM64
edi	w20
esi	w19
ebx	w21
edx	w1
ecx	w0
eax	w27
ebp	w29
eip	w9
esp	w28

Translated ARM64



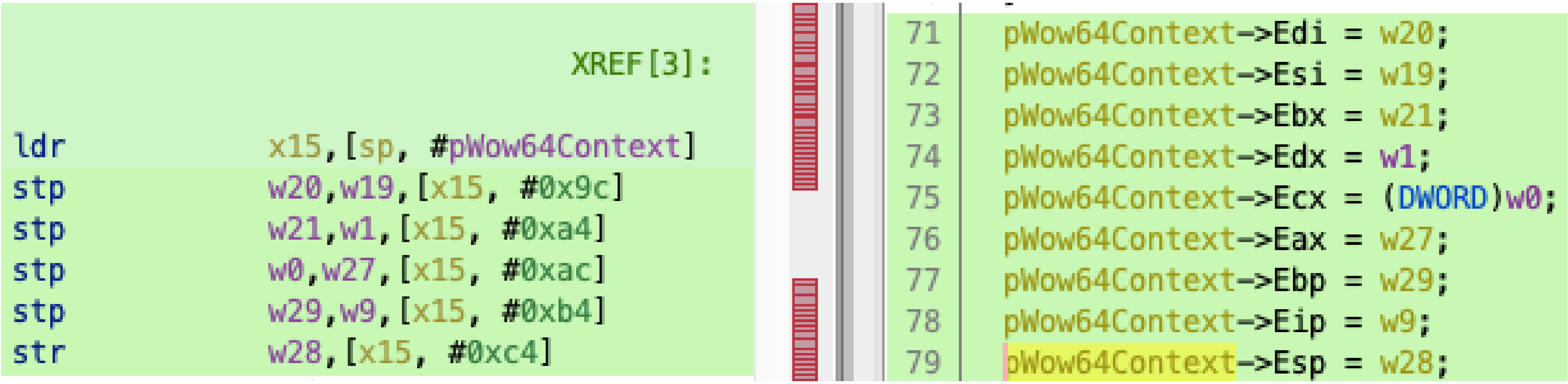
```
str w29, [x28, -4]!
mov w29, w28
movz w27, 0x1
mov w28, w29
ldr w29, [x28], 4
```

Original x86



```
push ebp
mov ebp, esp
mov eax, 1
mov esp, ebp
pop ebp
```

Context is restored/saved to Wow64Context structure



```
                                  XREF[3]:

ldr          x15,[sp, #pWow64Context]
stp          w20,w19,[x15, #0x9c]
stp          w21,w1,[x15, #0xa4]
stp          w0,w27,[x15, #0xac]
stp          w29,w9,[x15, #0xb4]
str          w28,[x15, #0xc4]
```

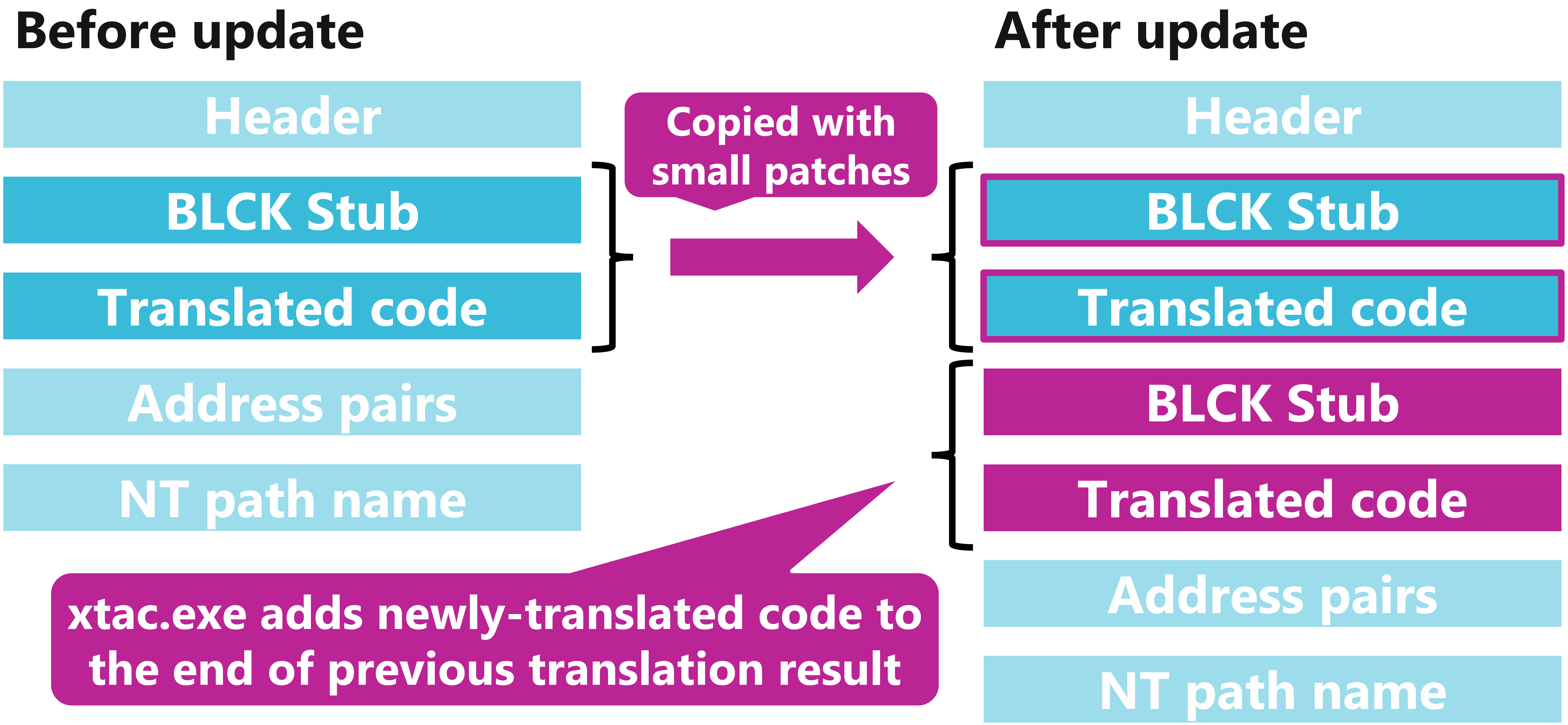
```
pWow64Context->Edi = w20;
pWow64Context->Esi = w19;
pWow64Context->Ebx = w21;
pWow64Context->Edx = w1;
pWow64Context->Ecx = (DWORD)w0;
pWow64Context->Eax = w27;
pWow64Context->Ebp = w29;
pWow64Context->Eip = w9;
pWow64Context->Esp = w28;
```

xtac.exe updates XTA cache file to add newly translated code

The previous translation result is copied to the new cache file

- to reduce the amount of binary translation by xtac
- But **small patches** are applied to the previous translation result

See appendix for more details



Before update

- Header
- BLCK Stub
- Translated code
- Address pairs
- NT path name

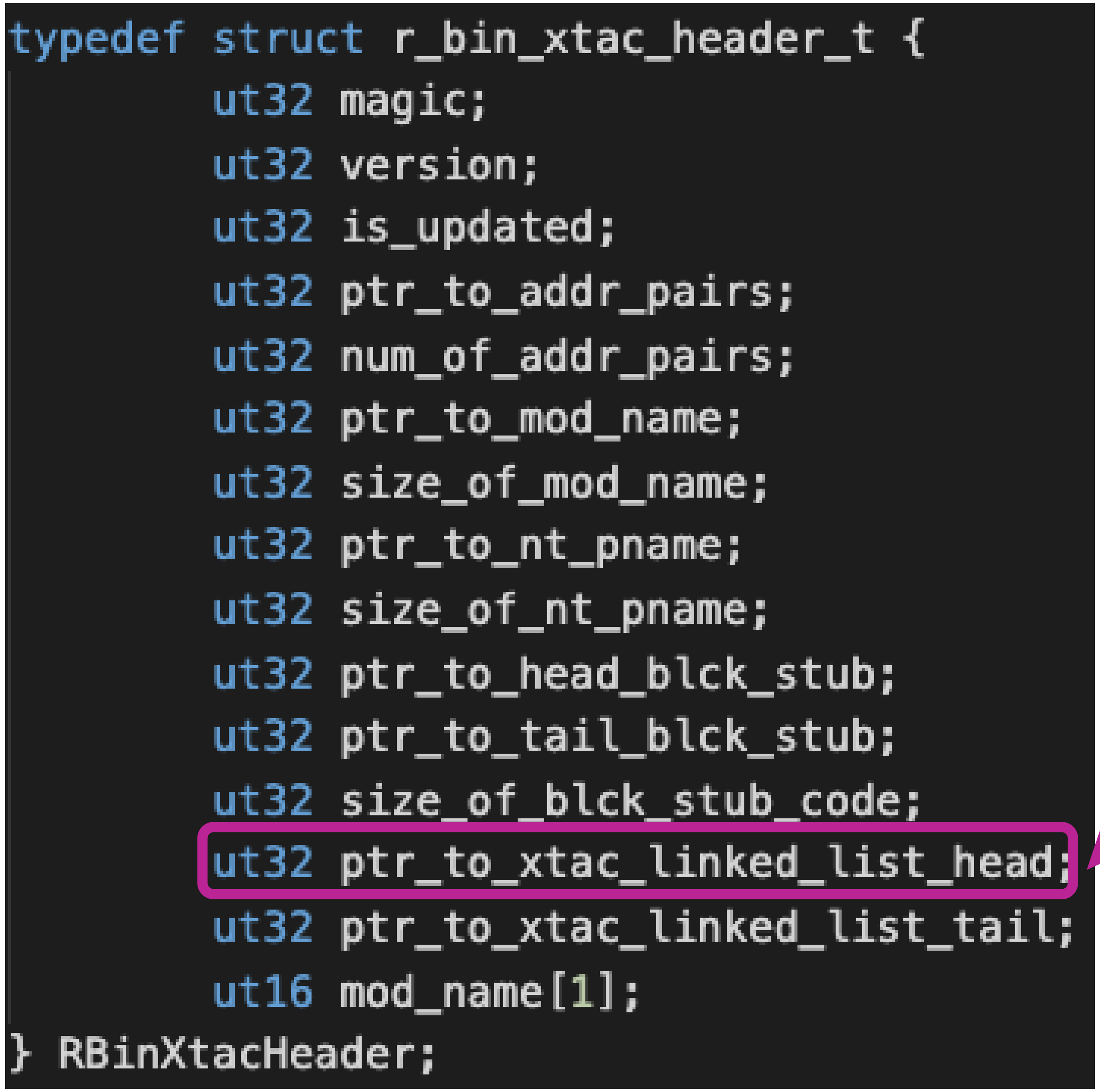
Copied with small patches

After update

- Header
- BLCK Stub
- Translated code
- BLCK Stub
- Translated code
- Address pairs
- NT path name

xtac.exe adds newly-translated code to the end of previous translation result

Prevention of XTA cache file update



```
typedef struct r_bin_xtac_header_t {
        ut32 magic;
        ut32 version;
        ut32 is_updated;
        ut32 ptr_to_addr_pairs;
        ut32 num_of_addr_pairs;
        ut32 ptr_to_mod_name;
        ut32 size_of_mod_name;
        ut32 ptr_to_nt_pname;
        ut32 size_of_nt_pname;
        ut32 ptr_to_head_blck_stub;
        ut32 ptr_to_tail_blck_stub;
        ut32 size_of_blck_stub_code;
        ut32 ptr_to_xtac_linked_list_head;
        ut32 ptr_to_xtac_linked_list_tail;
        ut16 mod_name[1];
} RBinXtacHeader;
```

xtac.exe uses this member for getting the positions to be patched.

Assigning an invalid value (e.g., 0xffffffff) to this member crashes xtac.exe and prevents the update.

Note: this change does not affect the cache file loading and execution of x86 app by xtajit

Quick recap: XTA cache file

It contains translated ARM64 code

- Without obfuscation and encryption
- During emulation, it is mapped to the memory

It is updated during/after emulation

- But this update can be prevented by modifying file header
- Although file header is modified, xtajit.dll can load this cache file

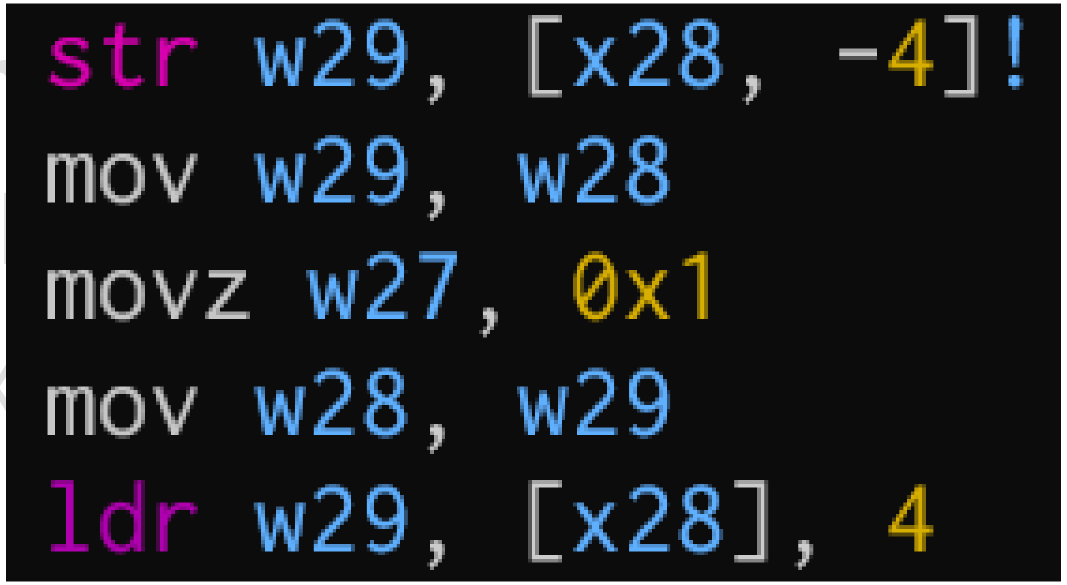
Quick recap: XTA cache file

It contains translated ARM64 code

- Without obfuscation and encryption
- During emulation, it is mapped to the memory

It is updated during emulatio

- But this update can be prevented
- Although file header is modified, x



```
str w29, [x28, -4]!
mov w29, w28
movz w27, 0x1
mov w28, w29
ldr w29, [x28], 4
```

Quick recap: XTA cache file

It contains translated ARM64 code

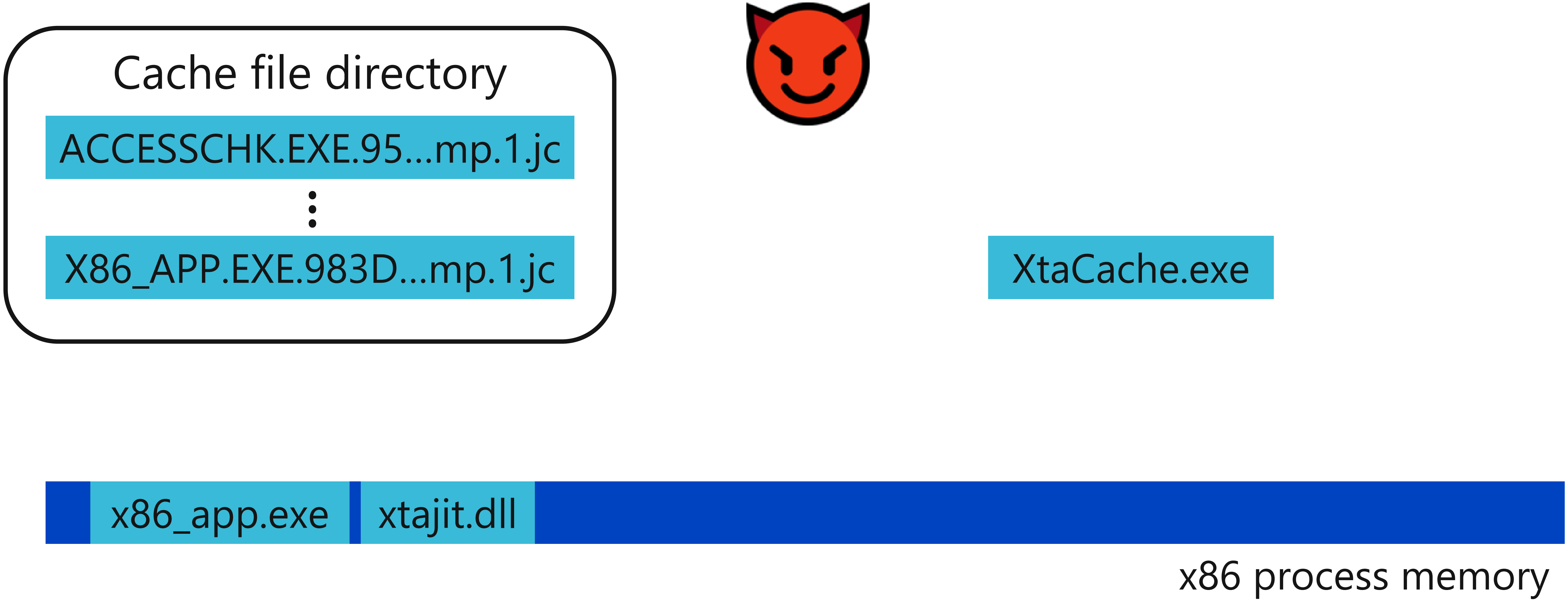
- **Without obfuscation and encryption**
- **During emulation, it is mapped to the memory**

It is updated during emulation

What happens if the XTA cache file is modified? 😈

- Although file header is modified, xtajit.dll can load this

Flow of execution when XTA cache file is modified



😈

Cache file directory

ACCESSCHK.EXE.95...mp.1.jc

⋮

X86_APP.EXE.983D...mp.1.jc

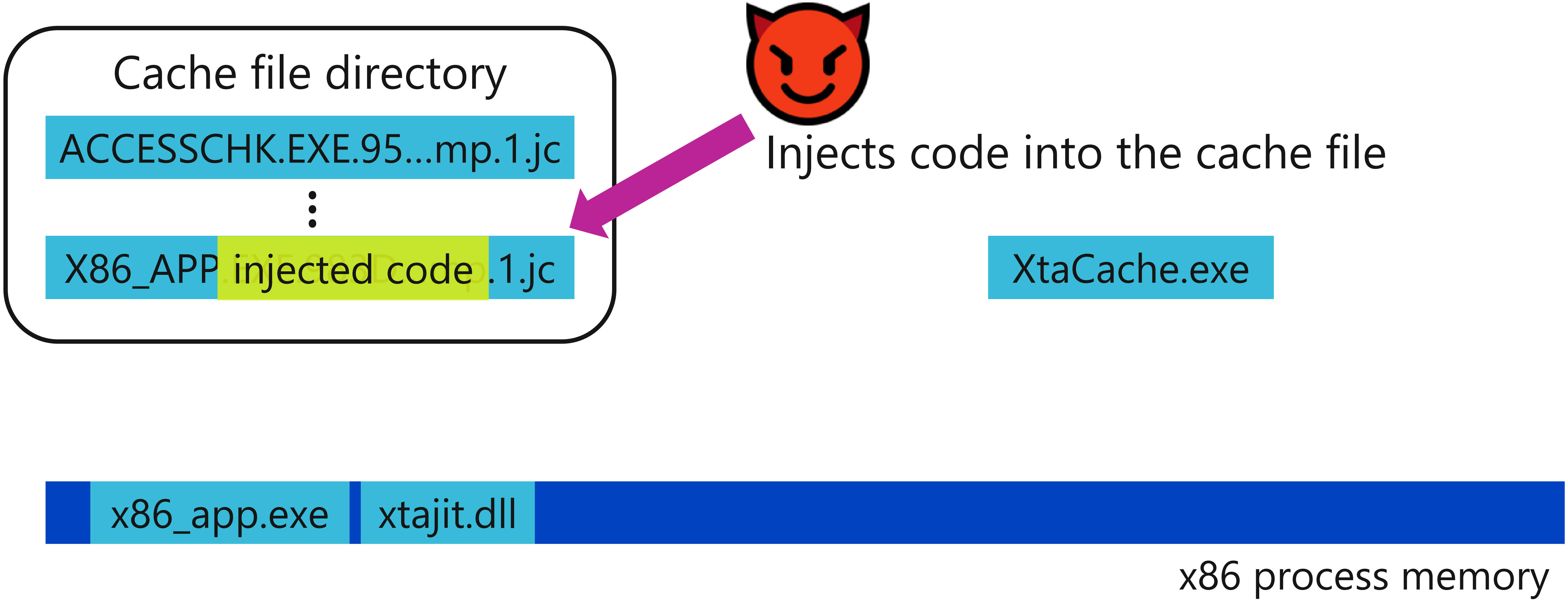
XtaCache.exe

x86_app.exe

xtajit.dll

x86 process memory

Flow of execution when XTA cache file is modified



Cache file directory

ACCESSCHK.EXE.95…mp.1.jc

X86_APP injected code .1.jc

Injects code into the cache file

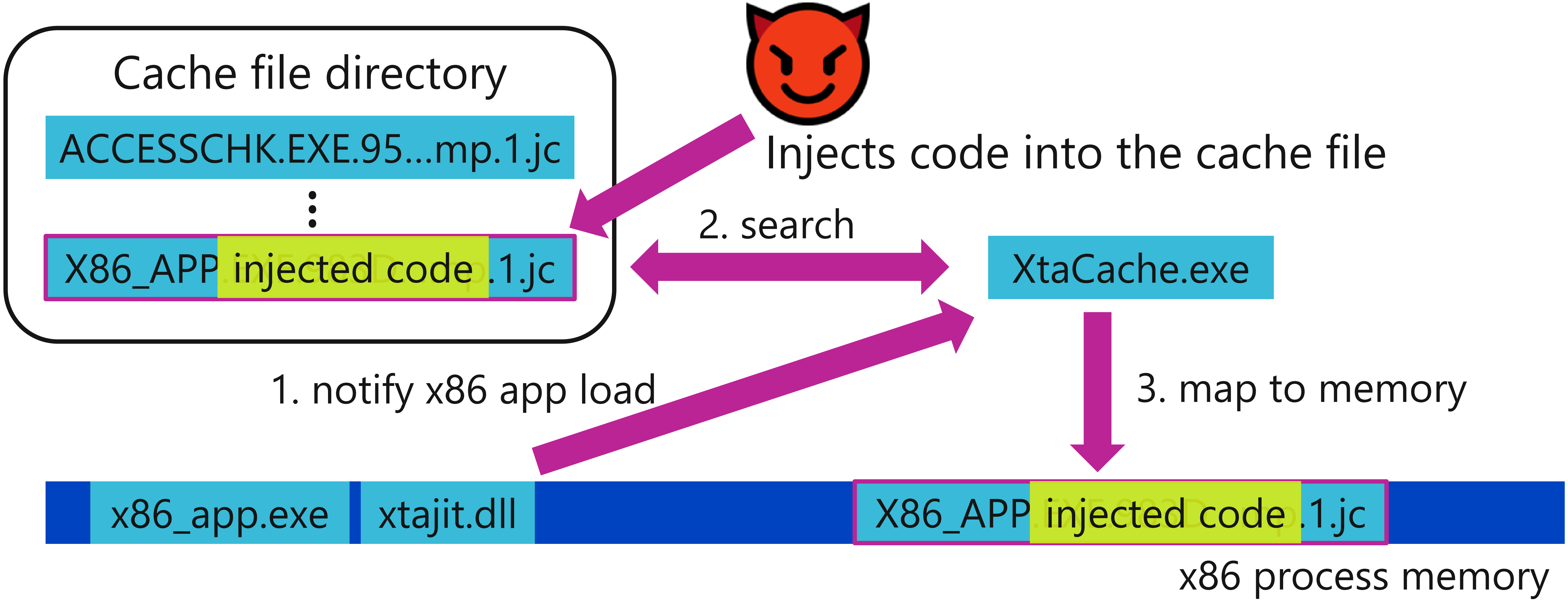
XtaCache.exe

x86_app.exe

xtajit.dll

x86 process memory

Flow of execution when XTA cache file is modified



Cache file directory

ACCESSCHK.EXE.95…mp.1.jc

X86_APP injected code .1.jc

Injects code into the cache file

2. search

XtaCache.exe

1. notify x86 app load

3. map to memory

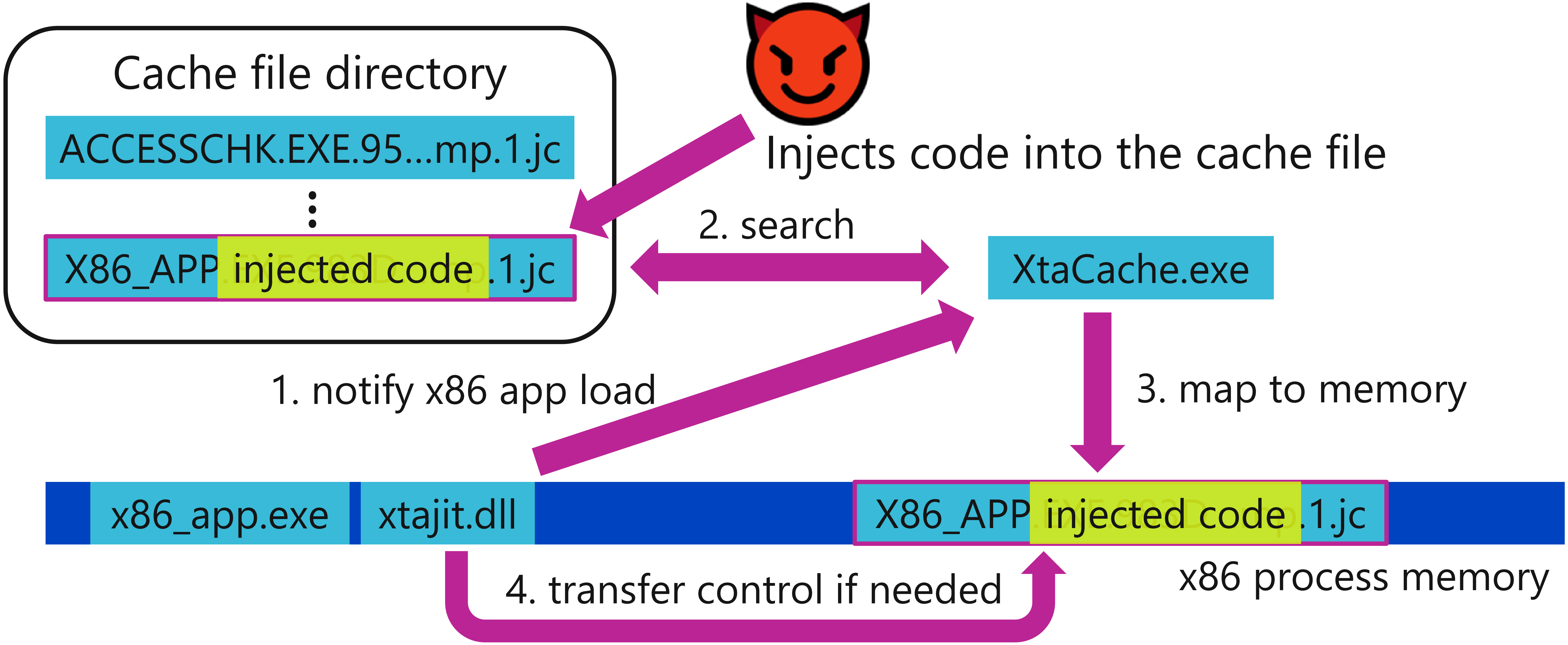
x86_app.exe

xtajit.dll

X86_APP injected code .1.jc

x86 process memory

Flow of execution when XTA cache file is modified



Cache file directory

ACCESSCHK.EXE.95…mp.1.jc

⋮

X86_APP injected code .1.jc

Injects code into the cache file

2. search

XtaCache.exe

1. notify x86 app load

3. map to memory

x86_app.exe

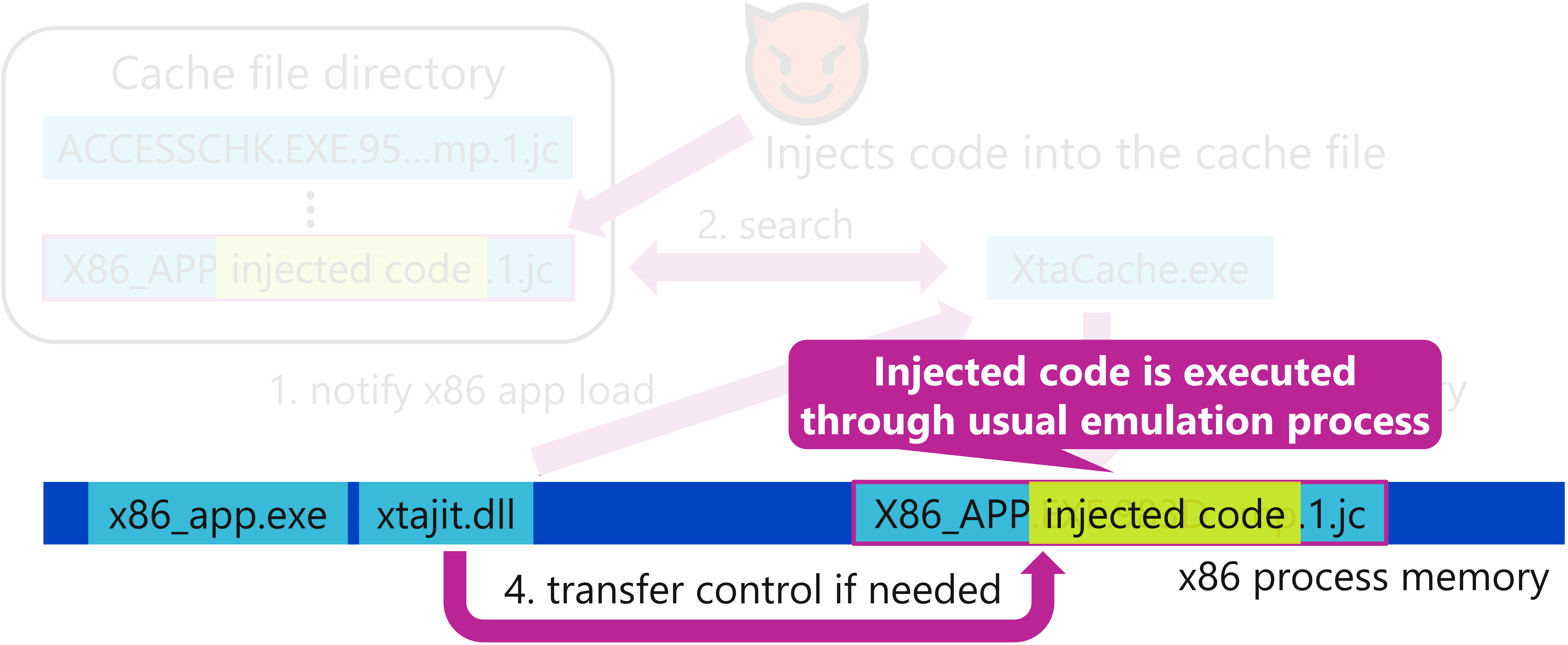
xtajit.dll

X86_APP injected code .1.jc

4. transfer control if needed

x86 process memory

Flow of execution when XTA cache file is modified



Cache file directory

ACCESSCHK.EXE.95...mp.1.jc

X86_APP injected code .1.jc

Injects code into the cache file

2. search

XtaCache.exe

1. notify x86 app load

Injected code is executed through usual emulation process

x86_app.exe

xtajit.dll

X86_APP injected code .1.jc

4. transfer control if needed

x86 process memory

What happens when the XTA cache file is modified?

Code in the XTA cache file is executed even though modified

- because the integrity of XTA cache file is not checked

No limitation for the embeddable content (size or encoding)

- An attacker can embed shellcode in the cache file and run it through emulation
  - But there are some limitations to callable APIs for shellcode (next slide)

We name this code injection **XTA cache hijacking**

Limitations of callable APIs

Some native APIs of DLLs in System32 are not callable

- E.g., GDI, Winsock, …

APIs of WOW64 layers are (of course) callable

Features of XTA cache hijacking

Three features of XTA cache hijacking

- Difficulty in detecting
- Difficulty in root cause analysis
- Persistence

Difficulty in detecting

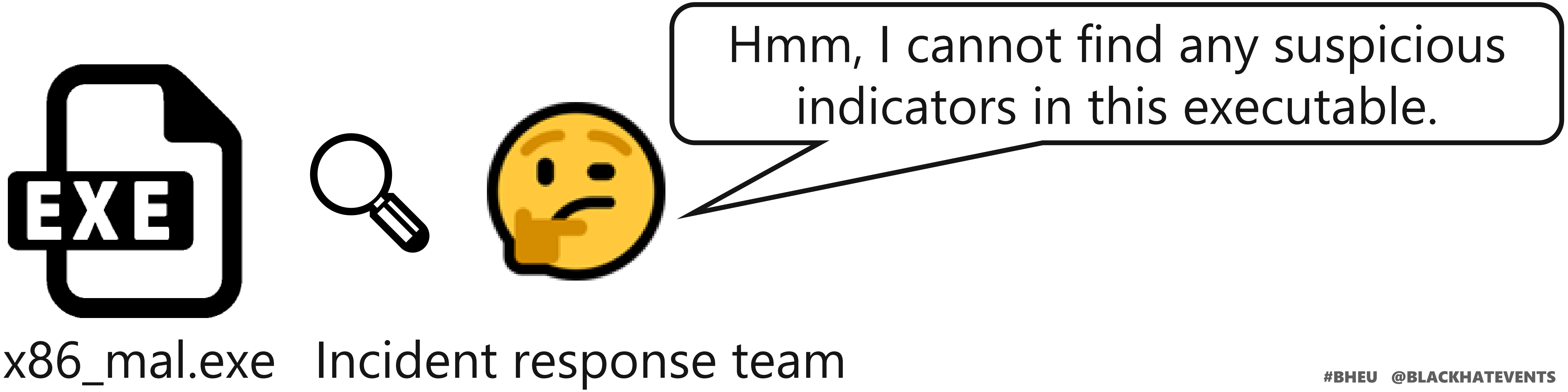
Accesses to the target process are not needed

- Code injection is performed without:
  - acquisition of the target process handle
  - suspicious API calls

Difficulty in root cause analysis

Cannot determine the root cause by examining the x86 app

- Since the executed code is in the XTA cache file, there are no suspicious indicators in the x86 app



Hmm, I cannot find any suspicious indicators in this executable.

x86_mal.exe

Incident response team

Difficulty in root cause analysis (contd.)

If any breakpoint is set to the x86 app, **the XTA cache file of x86 app is not used during emulation**

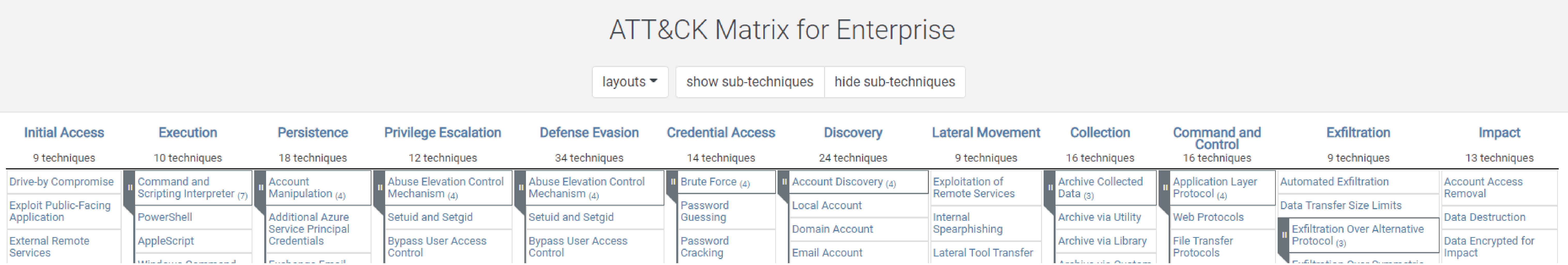
- Therefore, analysts cannot see the suspicious behaviors when setting any breakpoint by debugger
- This anti-debugging feature makes analysis difficult

Persistence

Code injection is persisted even after OS restart

- Code injection is automatically performed when the same x86 EXE or DLL runs again
- Updates of cache files can be prevented by modifying header
  - An attacker can achieve persistence by preventing cache file update

Positions in MITRE ATT&CK



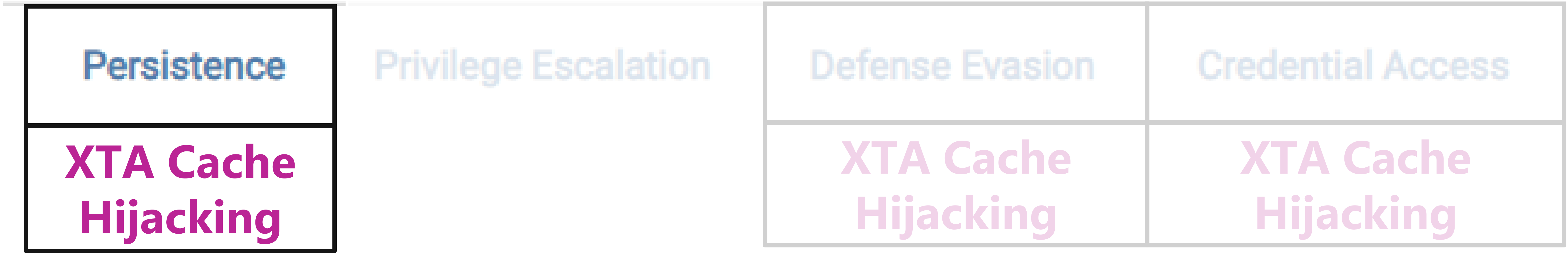
ATT&CK Matrix for Enterprise

Initial Access 9 techniques	Execution 10 techniques	Persistence 18 techniques	Privilege Escalation 12 techniques	Defense Evasion 34 techniques	Credential Access 14 techniques	Discovery 24 techniques	Lateral Movement 9 techniques	Collection 16 techniques	Command and Control 16 techniques	Exfiltration 9 techniques	Impact 13 techniques
Drive-by Compromise	Command and Scripting Interpreter (7)	Account Manipulation (4)	Abuse Elevation Control Mechanism (4)	Abuse Elevation Control Mechanism (4)	Brute Force (4)	Account Discovery (4)	Exploitation of Remote Services	Archive Collected Data (3)	Application Layer Protocol (4)	Automated Exfiltration	Account Access Removal
Exploit Public-Facing Application	PowerShell	Additional Azure Service Principal Credentials	Setuid and Setgid	Setuid and Setgid	Password Guessing	Local Account	Internal Spearphishing	Archive via Utility	Web Protocols	Data Transfer Size Limits	Data Destruction
External Remote Services	AppleScript		Bypass User Access Control	Bypass User Access Control	Password Cracking	Domain Account	Lateral Tool Transfer	Archive via Library	File Transfer Protocols	Exfiltration Over Alternative Protocol (3)	Data Encrypted for Impact
						Email Account					

Persistence	Privilege Escalation	Defense Evasion	Credential Access
XTA Cache Hijacking		XTA Cache Hijacking	XTA Cache Hijacking

https://attack.mitre.org/

Persistence

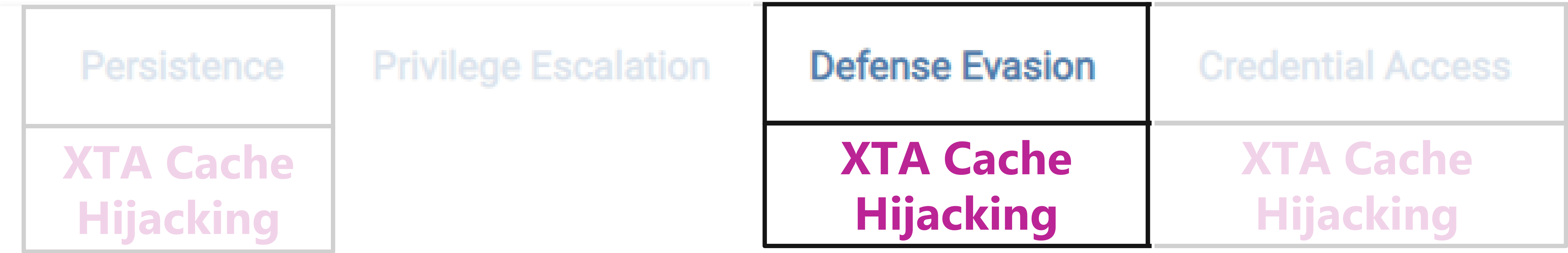


Persistence	Privilege Escalation	Defense Evasion	Credential Access
XTA Cache Hijacking		XTA Cache Hijacking	XTA Cache Hijacking

Used as a persistence method

- Can hide malicious shellcode in XTA cache file

Defense Evasion

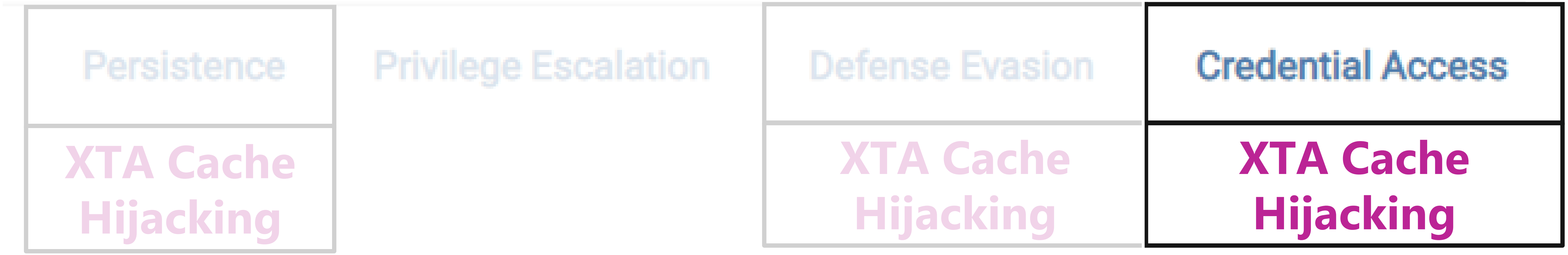


Persistence	Privilege Escalation	Defense Evasion	Credential Access
XTA Cache Hijacking		XTA Cache Hijacking	XTA Cache Hijacking

Used to mask malicious code

- Can run malicious code as a legitimate process

Credential Access



Persistence	Privilege Escalation	Defense Evasion	Credential Access
XTA Cache Hijacking		XTA Cache Hijacking	XTA Cache Hijacking

Used as a credential access method

- Can inject API hooking code into XTA cache files of DLLs that are used in a browser
  - Steal credentials / modify web pages

You might think that...

Hmm? XTA cache hijacking seems to be similar to other conventional code injection techniques.

What makes XTA cache hijacking so special?
Why is it so interesting?

🤔

Ko -> Hiromitsu

It’s a new technique

- targets new OS and its technology (Win10 ARM, xtajit)
- has persistence
- makes investigations difficult

Good. But.. that‘s all?

Remind that..

It is realized by modifying **cache of translation result**

- They are **ARM64 machine codes**

We can change the behavior of x86 processes

w/o any modifications to x86 instructions!

What’s happening?

ARM64 CPUs cannot execute x86 instructions directly

- unlike x86-64 CPUs

x86 instructions are only referenced when translating

- If already cache exist, they are not referenced

The instructions in the cache take precedence

- Even if the behavior of the cache and the original are different..

Side effect: Invisible Execution

There are no changes for x86 instructions on memory

Execution of payloads is invisible to x86 layer

- **The execution state on ARM64 layer is invisible to x86 layer**
  - Even if you follow the execution with debugger, you can see unmodified x86 instructions only

demo

Use-case: Invisible API Hook

We can detect hooks with checking the beginning of API

- commonly used method modifies the instruction at beginning of the function

We can avoid the detection and the tracing for hooks!

- by applying our method to **CHPE DLLs**

CHPE DLLs

bridge DLLs between x86 and ARM64

- used in x86 processes on Win10 ARM

Exist for some DLLs frequently used by applications

- e.g., kernel32.dll, user32.dll, ntdll.dll

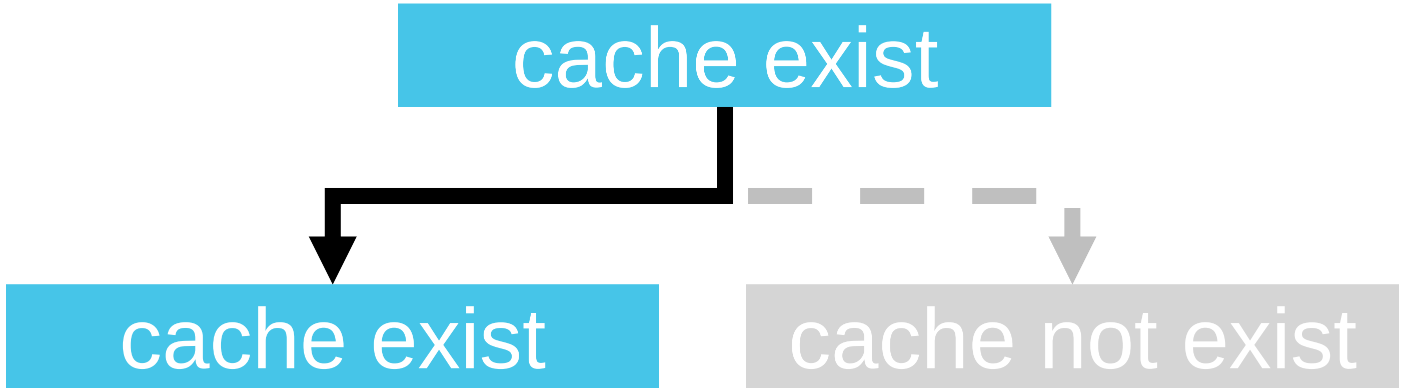
Have x86 stubs for each API

- **Of course, caches are generated for these x86 instructions**

Bonus

Find out path executed from the existence of the cache

- No execution, no cache



Bonus

Find out path executed from the existence of the cache

- **Non-invasive coverage measurement**
  - E.g., fuzzing? (see appendix for more details)

- **Incident Response**
  - E.g., Investigate what the RAT did without a communications log

Tool for this will also be available!

Conclusion

New code injection technique for Windows10 on ARM

- exploits the cache in x86 to ARM64 JIT Translation
- has a unique side effect and some benefits

Advices

For one developing similar system

- **Ensure the integrity of cache**
  - This technique requires privilege escalation, **but still worth**

Everyone

- **Be Aware of the threat**
  - It will be difficult to find out on first sight if one don’t know about this

PoC code and some analysis tools are available at

- Some tools to manipulate XTA cache files
  - https://github.com/FFRI/XtaTools
- Analysis tool for XTA cache files
  - https://github.com/FFRI/radare2

Thank you!

Any questions and comments to

- Twitter DM: @FFRI_Research
- e-mail: research-feedback@ffri.jp

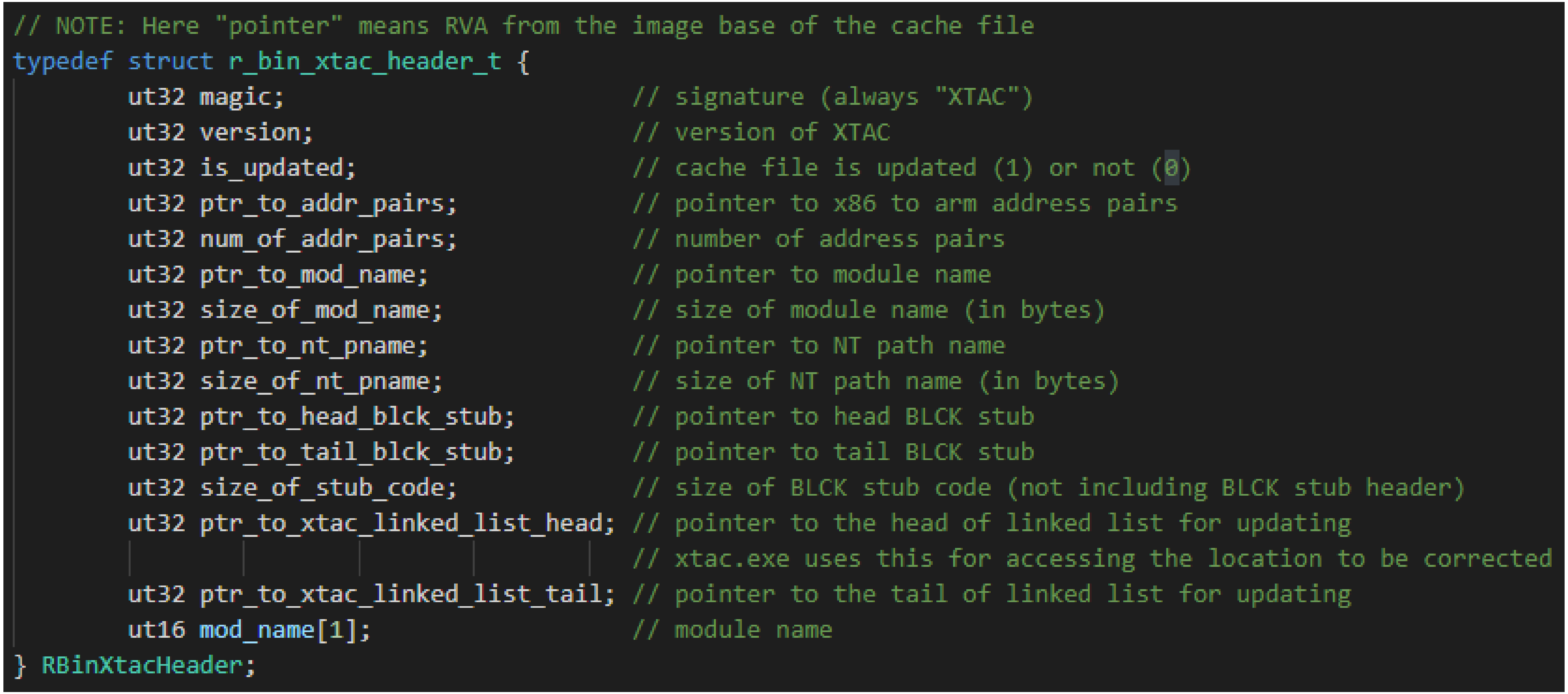
Acknowledgements

Thank my colleagues for giving some helpful comments on this material.

Appendix

Structure of XTA cache file

XTA cache file header and its members



```
// NOTE: Here "pointer" means RVA from the image base of the cache file
typedef struct r_bin_xtac_header_t {
        ut32 magic;                       // signature (always "XTAC")
        ut32 version;                     // version of XTAC
        ut32 is_updated;                  // cache file is updated (1) or not (0)
        ut32 ptr_to_addr_pairs;           // pointer to x86 to arm address pairs
        ut32 num_of_addr_pairs;           // number of address pairs
        ut32 ptr_to_mod_name;             // pointer to module name
        ut32 size_of_mod_name;            // size of module name (in bytes)
        ut32 ptr_to_nt_pname;             // pointer to NT path name
        ut32 size_of_nt_pname;            // size of NT path name (in bytes)
        ut32 ptr_to_head_blck_stub;       // pointer to head BLCK stub
        ut32 ptr_to_tail_blck_stub;       // pointer to tail BLCK stub
        ut32 size_of_stub_code;           // size of BLCK stub code (not including BLCK stub header)
        ut32 ptr_to_xtac_linked_list_head; // pointer to the head of linked list for updating
                                          // xtac.exe uses this for accessing the location to be corrected
        ut32 ptr_to_xtac_linked_list_tail; // pointer to the tail of linked list for updating
        ut16 mod_name[1];                 // module name
} RBinXtacHeader;
```

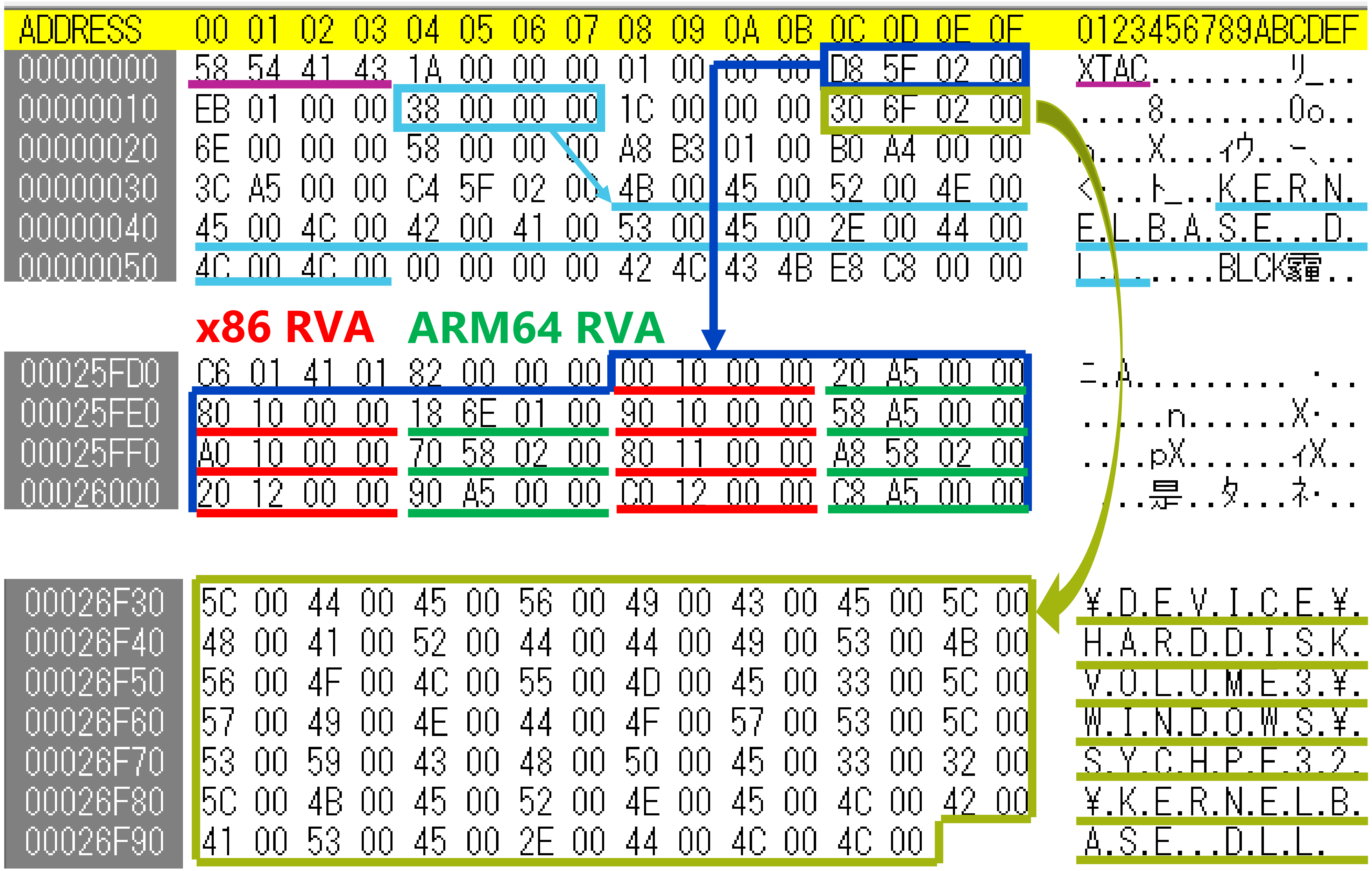
Example:

XTA cache file of SystemRoot¥SysChpe32¥kernelbase.dll

file name:

```
KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc
```

KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc



```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00000000  58 54 41 43 1A 00 00 00 01 00 00 00 D8 5F 02 00   XTAC........ﾘ_..
00000010  EB 01 00 00 38 00 00 00 1C 00 00 00 30 6F 02 00   ....8.......0o..
00000020  6E 00 00 00 58 00 00 00 A8 B3 01 00 B0 A4 00 00   n...X...ｨｳ..ｰ､..
00000030  3C A5 00 00 C4 5F 02 00 4B 00 45 00 52 00 4E 00   <･..ﾄ_..K.E.R.N.
00000040  45 00 4C 00 42 00 41 00 53 00 45 00 2E 00 44 00   E.L.B.A.S.E...D.
00000050  4C 00 4C 00 00 00 00 00 42 4C 43 4B E8 C8 00 00   L.L.....BLCK霤..

00025FD0  C6 01 41 01 82 00 00 00 00 10 00 00 20 A5 00 00   ﾆ.A......... ･..
00025FE0  80 10 00 00 18 6E 01 00 90 10 00 00 58 A5 00 00   .....n......X･..
00025FF0  A0 10 00 00 70 58 02 00 80 11 00 00 A8 58 02 00   ....pX......ｨX..
00026000  20 12 00 00 90 A5 00 00 C0 12 00 00 C8 A5 00 00    ...昱..ﾀ...ﾈ･..

00026F30  5C 00 44 00 45 00 56 00 49 00 43 00 45 00 5C 00   ¥.D.E.V.I.C.E.¥.
00026F40  48 00 41 00 52 00 44 00 44 00 49 00 53 00 4B 00   H.A.R.D.D.I.S.K.
00026F50  56 00 4F 00 4C 00 55 00 4D 00 45 00 33 00 5C 00   V.O.L.U.M.E.3.¥.
00026F60  57 00 49 00 4E 00 44 00 4F 00 57 00 53 00 5C 00   W.I.N.D.O.W.S.¥.
00026F70  53 00 59 00 43 00 48 00 50 00 45 00 33 00 32 00   S.Y.C.H.P.E.3.2.
00026F80  5C 00 4B 00 45 00 52 00 4E 00 45 00 4C 00 42 00   ¥.K.E.R.N.E.L.B.
00026F90  41 00 53 00 45 00 2E 00 44 00 4C 00 4C 00         A.S.E...D.L.L.
```

Magic (always XTAC)

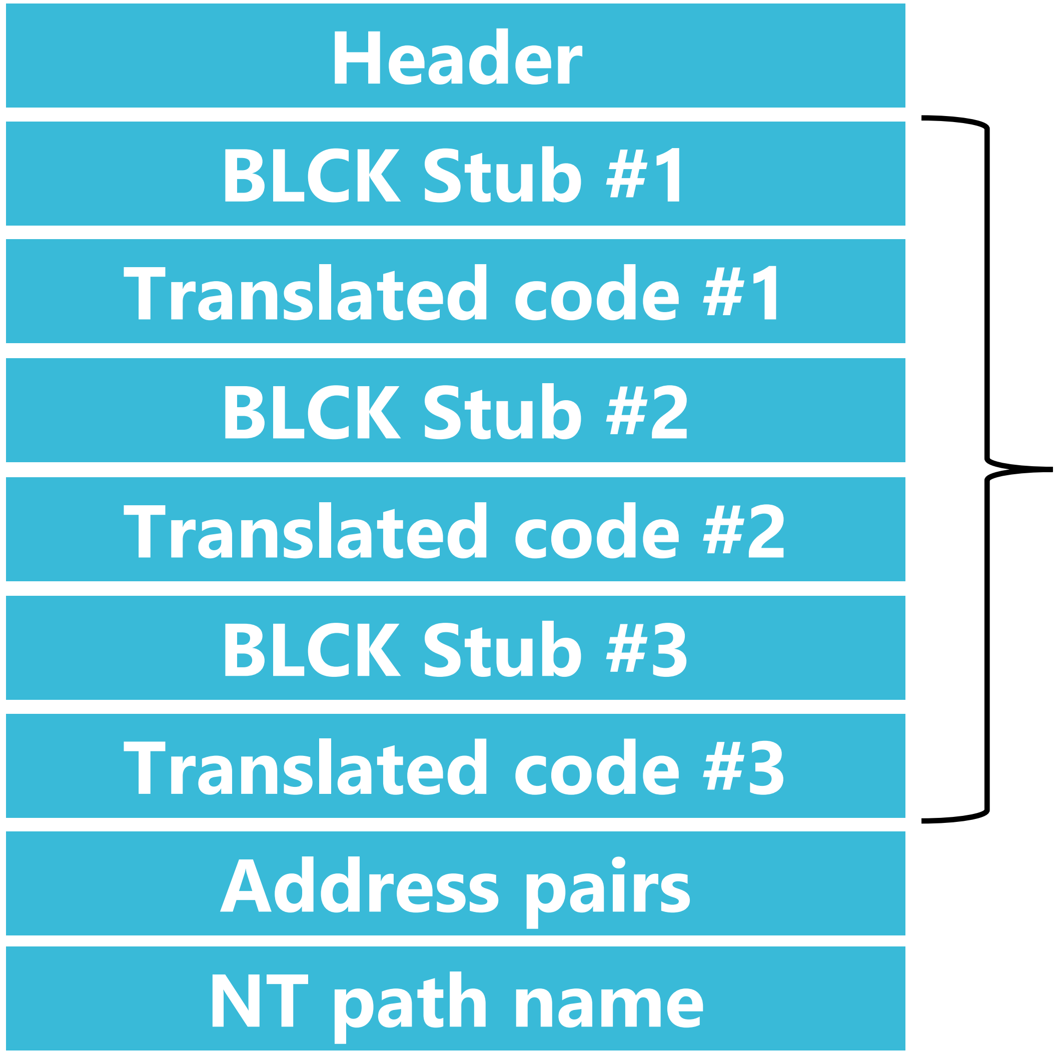
Module name of x86 app

Address pairs

NT path name of x86 app

KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc

The number of updates is 3



Header

BLCK Stub #1

Translated code #1

BLCK Stub #2

Translated code #2

BLCK Stub #3

Translated code #3

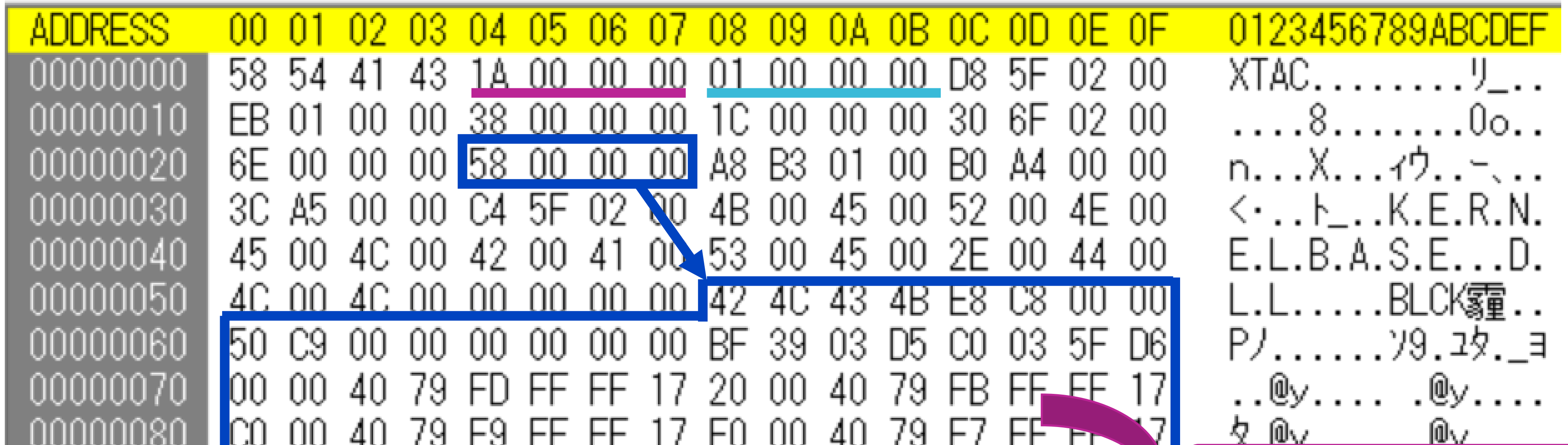
Address pairs

NT path name

BLCK Stub and translated code are repeated for three times.

Each BLCK Stub contains the offset to the next BLCK stub

KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc



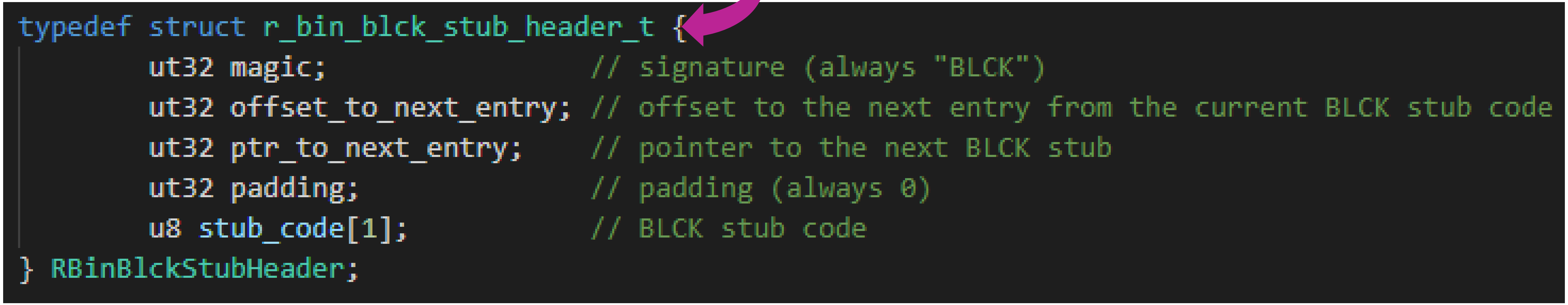
```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00000000  58 54 41 43 1A 00 00 00 01 00 00 00 D8 5F 02 00   XTAC........ﾘ_..
00000010  EB 01 00 00 38 00 00 00 1C 00 00 00 30 6F 02 00   ....8.......0o..
00000020  6E 00 00 00 58 00 00 00 A8 B3 01 00 B0 A4 00 00   n...X...ｲｳ..ｰ､..
00000030  3C A5 00 00 C4 5F 02 00 4B 00 45 00 52 00 4E 00   <･..ﾄ_..K.E.R.N.
00000040  45 00 4C 00 42 00 41 00 53 00 45 00 2E 00 44 00   E.L.B.A.S.E...D.
00000050  4C 00 4C 00 00 00 00 00 42 4C 43 4B E8 C8 00 00   L.L.....BLCK霾..
00000060  50 C9 00 00 00 00 00 00 BF 39 03 D5 C0 03 5F D6   Pﾉ......ｿ9.ﾕﾀ._ﾖ
00000070  00 00 40 79 FD FF FF 17 20 00 40 79 FB FF FF 17   ..@y.... .@y....
00000080  C0 00 40 79 F9 FF FF 17 E0 00 40 79 F7 FF FF 17   ﾀ.@y....ｰ.@y....
```

Cache file version

Cache file is updated or not
(1: updated, 0: not-updated)

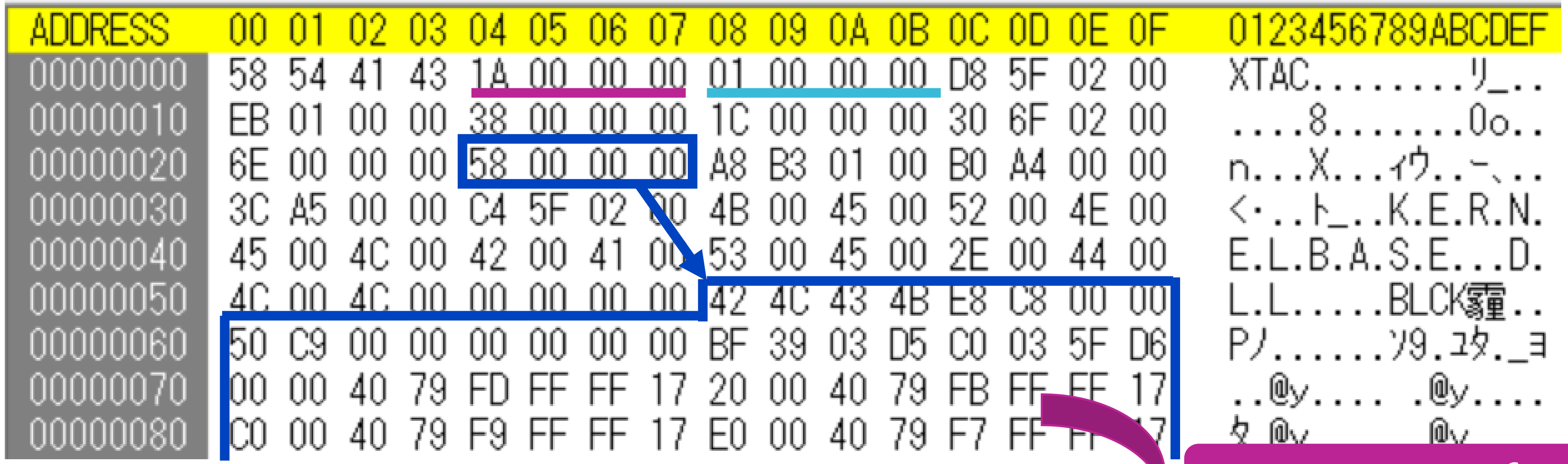
BLCK Stub

Structure of BLCK Stub is …



```c
typedef struct r_bin_blck_stub_header_t {
        ut32 magic;              // signature (always "BLCK")
        ut32 offset_to_next_entry; // offset to the next entry from the current BLCK stub code
        ut32 ptr_to_next_entry;  // pointer to the next BLCK stub
        ut32 padding;            // padding (always 0)
        u8 stub_code[1];         // BLCK stub code
} RBinBlckStubHeader;
```

KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc



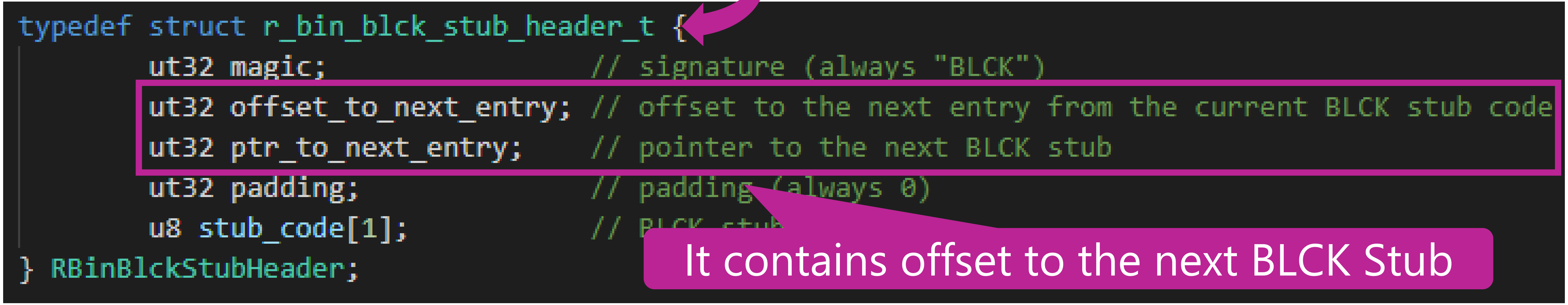
```
ADDRESS    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00000000   58 54 41 43 1A 00 00 00 01 00 00 00 D8 5F 02 00   XTAC........ﾘ_..
00000010   EB 01 00 00 38 00 00 00 1C 00 00 00 30 6F 02 00   ....8.......0o..
00000020   6E 00 00 00 58 00 00 00 A8 B3 01 00 B0 A4 00 00   n...X...ｨｳ..ｰ､..
00000030   3C A5 00 00 C4 5F 02 00 4B 00 45 00 52 00 4E 00   <･..ﾄ_..K.E.R.N.
00000040   45 00 4C 00 42 00 41 00 53 00 45 00 2E 00 44 00   E.L.B.A.S.E...D.
00000050   4C 00 4C 00 00 00 00 00 42 4C 43 4B E8 C8 00 00   L.L.....BLCK霾..
00000060   50 C9 00 00 00 00 00 00 BF 39 03 D5 C0 03 5F D6   Pﾉ......ｿ9.ﾕﾀ._ﾖ
00000070   00 00 40 79 FD FF FF 17 20 00 40 79 FB FF FF 17   ..@y.... .@y....
00000080   C0 00 40 79 F9 FF FF 17 E0 00 40 79 F7 FF FF 17   ﾀ.@y.....@y....
```

Cache file version

Cache file is updated or not (1: updated, 0: not-updated)

BLCK Stub

Structure of BLCK Stub is …

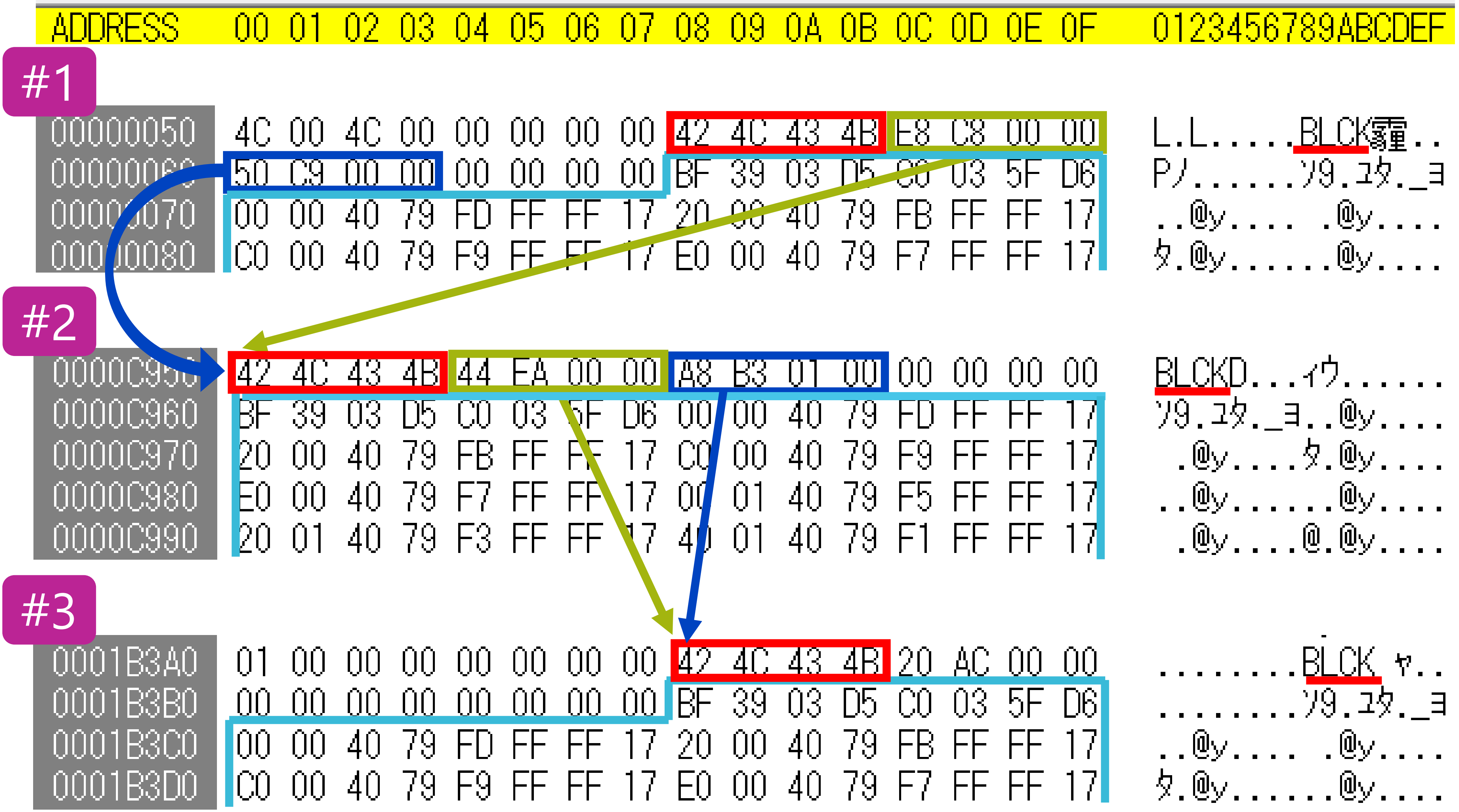


```
typedef struct r_bin_blck_stub_header_t {
        ut32 magic;                 // signature (always "BLCK")
        ut32 offset_to_next_entry; // offset to the next entry from the current BLCK stub code
        ut32 ptr_to_next_entry;     // pointer to the next BLCK stub
        ut32 padding;               // padding (always 0)
        u8 stub_code[1];            // BLCK stub
} RBinBlckStubHeader;
```

It contains offset to the next BLCK Stub

KERNELBASE.DLL.152D9019D54A662A18EC7A673ECB130F.DB966B70C90268F5B3A22AF2FFD62FB9.mp.3.jc

Relation among three BLCK Stub #1, #2, and #3



```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
#1
00000050  4C 00 4C 00 00 00 00 00 42 4C 43 4B E8 C8 00 00   L.L.....BLCK霾..
00000060  50 C9 00 00 00 00 00 00 BF 39 03 D5 C0 03 5F D6   Pﾉ......ｿ9.ﾕﾀ._ﾖ
00000070  00 00 40 79 FD FF FF 17 20 00 40 79 FB FF FF 17   ..@y.... .@y....
00000080  C0 00 40 79 F9 FF FF 17 E0 00 40 79 F7 FF FF 17   ﾀ.@y......@y....
#2
0000C950  42 4C 43 4B 44 EA 00 00 A8 B3 01 00 00 00 00 00   BLCKD...ｨｳ......
0000C960  BF 39 03 D5 C0 03 5F D6 00 00 40 79 FD FF FF 17   ｿ9.ﾕﾀ._ﾖ..@y....
0000C970  20 00 40 79 FB FF FF 17 C0 00 40 79 F9 FF FF 17    .@y....ﾀ.@y....
0000C980  E0 00 40 79 F7 FF FF 17 00 01 40 79 F5 FF FF 17   ..@y......@y....
0000C990  20 01 40 79 F3 FF FF 17 40 01 40 79 F1 FF FF 17    .@y....@.@y....
#3
0001B3A0  01 00 00 00 00 00 00 00 42 4C 43 4B 20 AC 00 00   ........BLCK ｬ..
0001B3B0  00 00 00 00 00 00 00 00 BF 39 03 D5 C0 03 5F D6   ........ｿ9.ﾕﾀ._ﾖ
0001B3C0  00 00 40 79 FD FF FF 17 20 00 40 79 FB FF FF 17   ..@y.... .@y....
0001B3D0  C0 00 40 79 F9 FF FF 17 E0 00 40 79 F7 FF FF 17   ﾀ.@y......@y....
```

Pointer to next entry

BLCK Stub code

Offset to next entry
(relative to BLCK Stub code's start address)

CHPE DLL

Compiled-Hybrid-PE (CHPE) DLL

looks as if x86 PE file, but **contains x86 and ARM64 code** [1, 2]

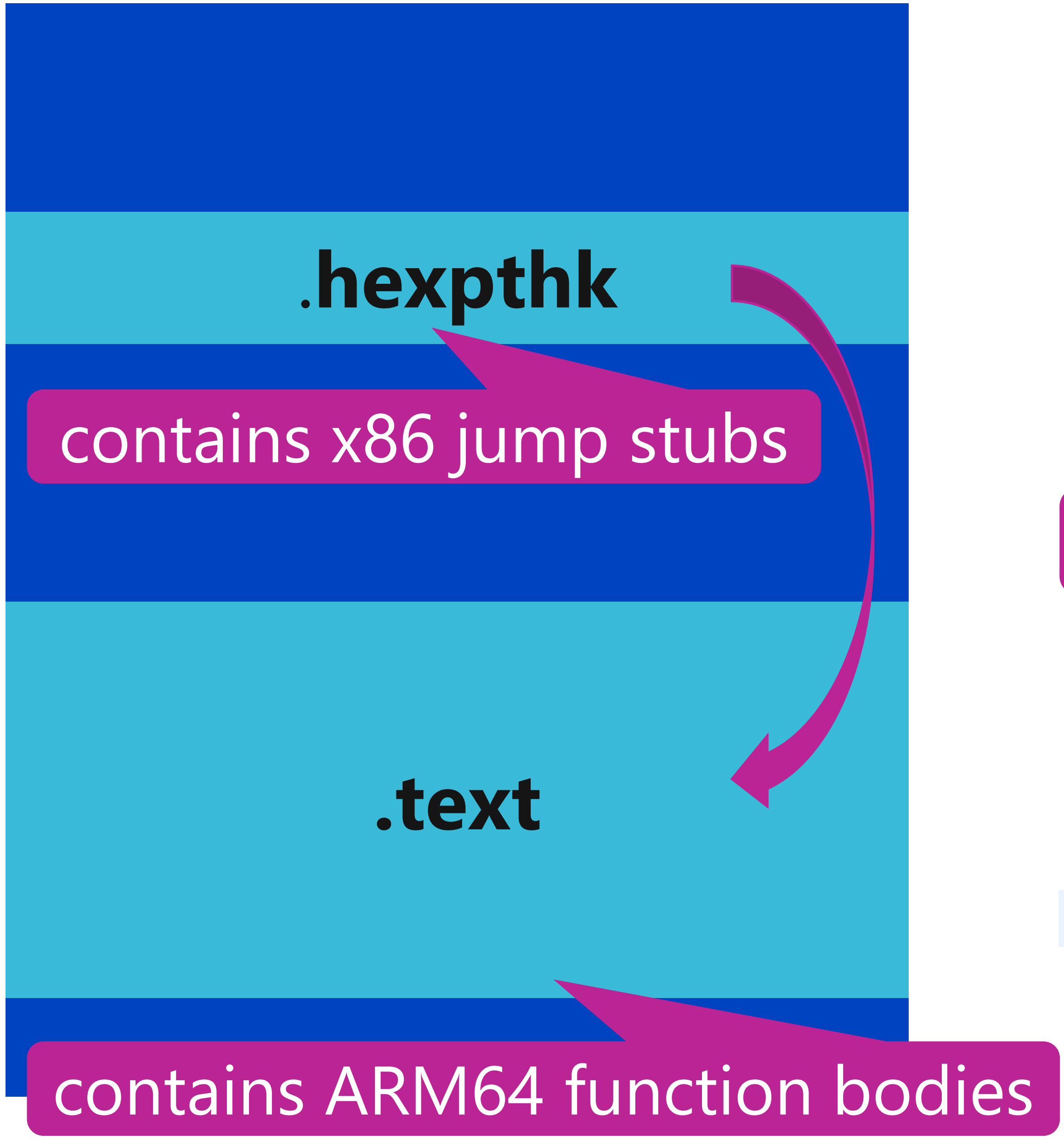
- Small subset of DLLs frequently used by applications

Exported APIs contain x86 jump stubs to ARM64 function bodies

- JIT translation is performed only for these x86 stubs
  - It reduces the amount of JIT binary translation

[1] https://wbenny.github.io/2018/11/04/wow64-internals.html
[2] https://blogs.blackberry.com/en/2019/09/teardown-windows-10-on-arm-x86-emulation

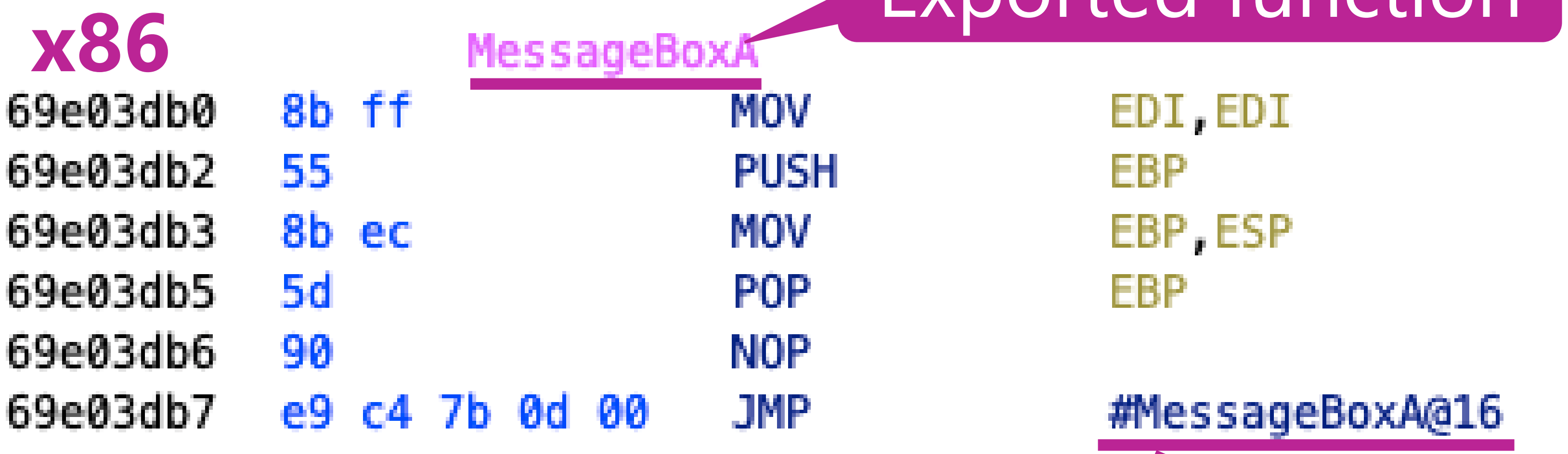
Example: MessageBoxA @ SystemRoot¥SysChpe32¥user32.dll



contains x86 jump stubs

contains ARM64 function bodies

Exported function

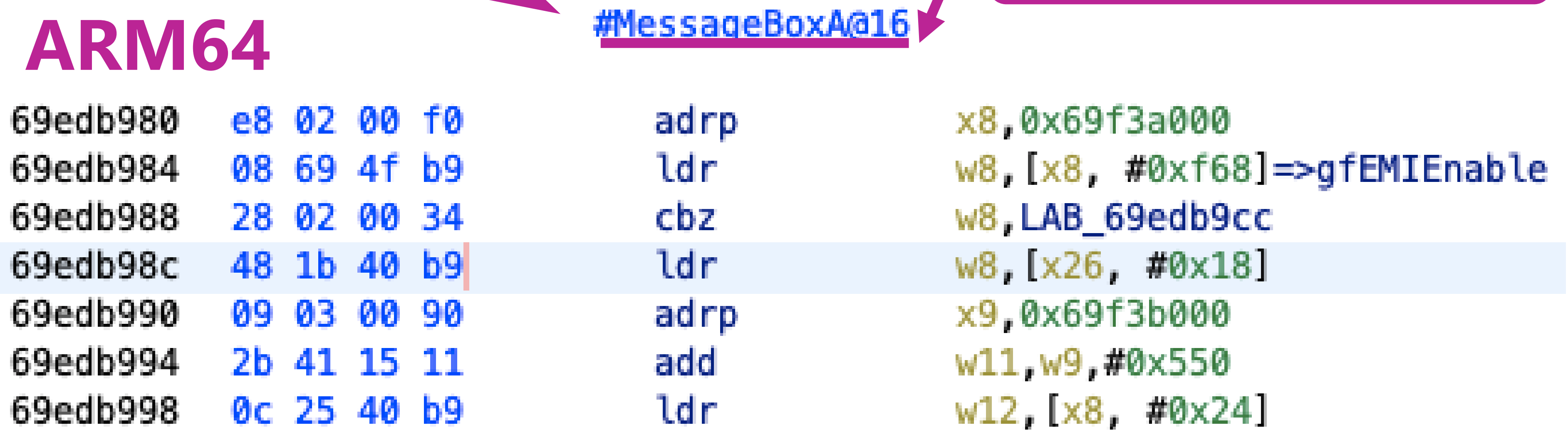


x86

```
                              MessageBoxA
69e03db0    8b ff             MOV              EDI,EDI
69e03db2    55                PUSH             EBP
69e03db3    8b ec             MOV              EBP,ESP
69e03db5    5d                POP              EBP
69e03db6    90                NOP
69e03db7    e9 c4 7b 0d 00    JMP              #MessageBoxA@16
```

Function body

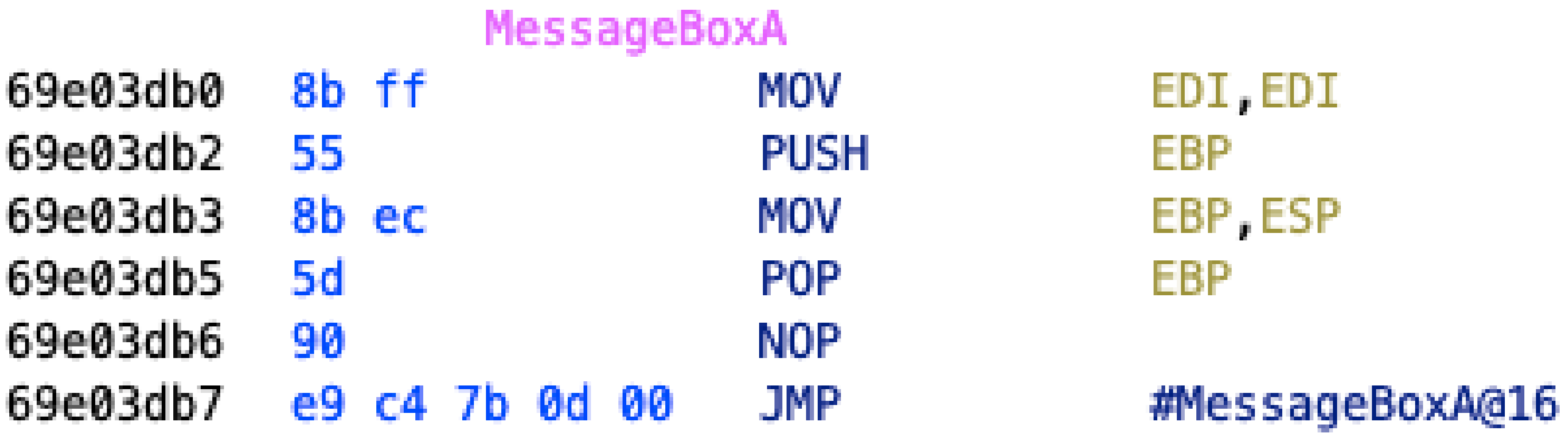
jumps to body



ARM64

```
                                  #MessageBoxA@16
69edb980    e8 02 00 f0       adrp        x8,0x69f3a000
69edb984    08 69 4f b9       ldr         w8,[x8, #0xf68]=>gfEMIEnable
69edb988    28 02 00 34       cbz         w8,LAB_69edb9cc
69edb98c    48 1b 40 b9       ldr         w8,[x26, #0x18]
69edb990    09 03 00 90       adrp        x9,0x69f3b000
69edb994    2b 41 15 11       add         w11,w9,#0x550
69edb998    0c 25 40 b9       ldr         w12,[x8, #0x24]
```

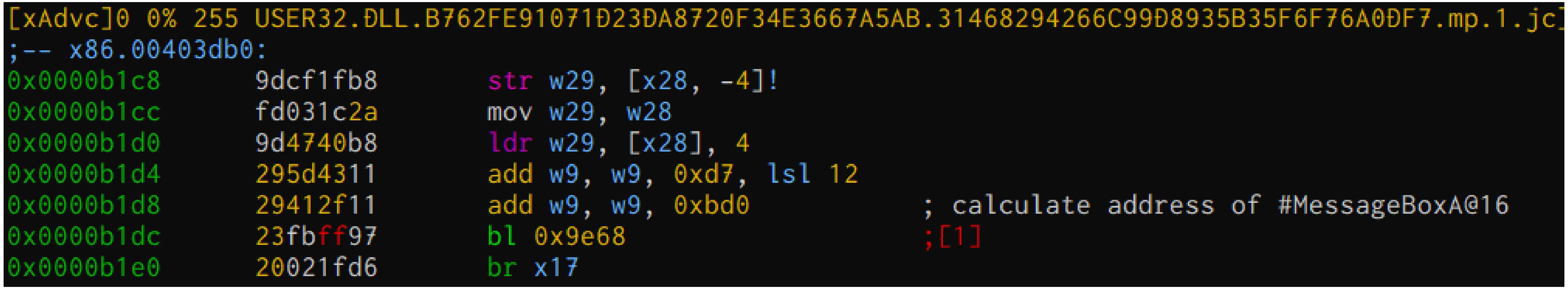
Example: MessageBoxA @ SystemRoot¥SysChpe32¥user32.dll



```
                    MessageBoxA
69e03db0    8b ff             MOV        EDI,EDI
69e03db2    55                PUSH       EBP
69e03db3    8b ec             MOV        EBP,ESP
69e03db5    5d                POP        EBP
69e03db6    90                NOP
69e03db7    e9 c4 7b 0d 00    JMP        #MessageBoxA@16
```

xtac.exe translates this code

XTA cache file contains only the translation result of jump stubs



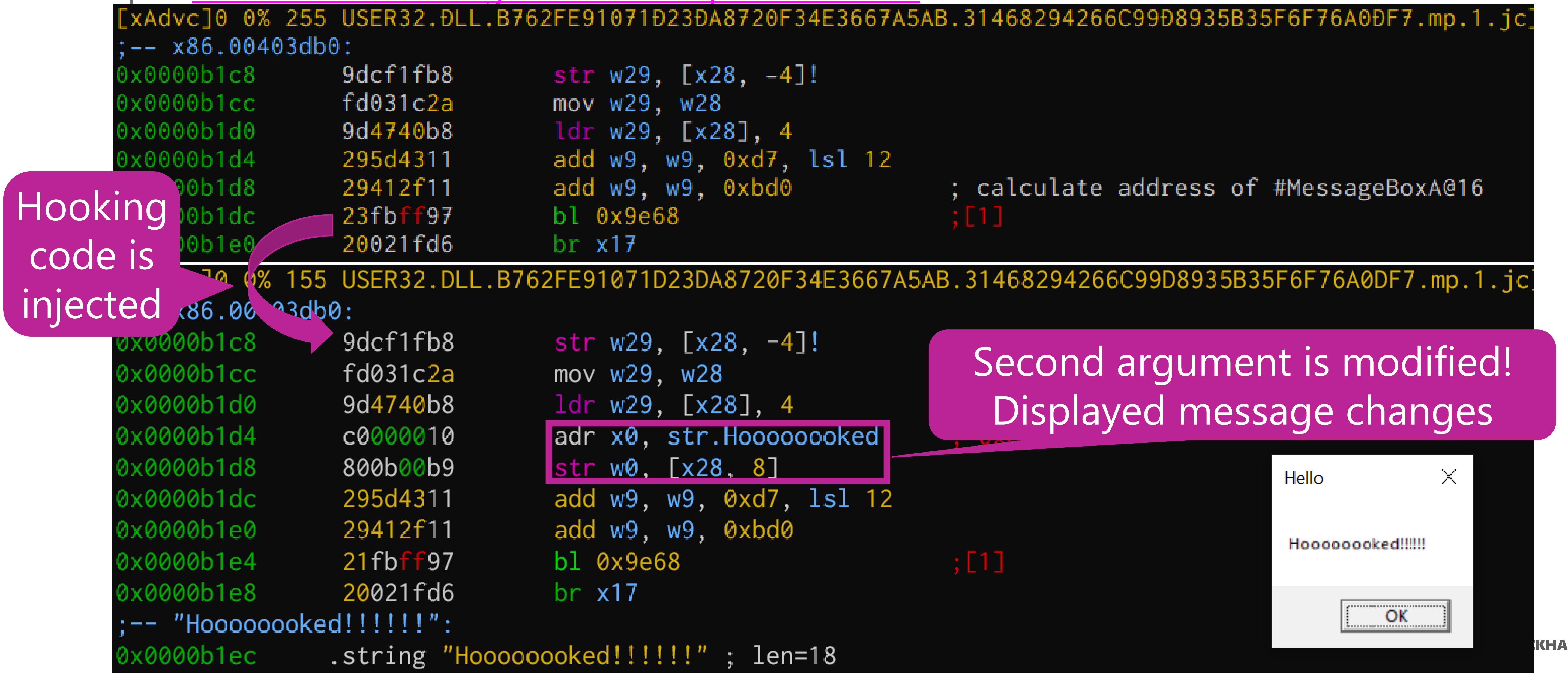
```
[xAdvc]0 0% 255 USER32.DLL.B762FE91071D23DA8720F34E3667A5AB.31468294266C99D8935B35F6F76A0DF7.mp.1.jc]
;-- x86.00403db0:
0x0000b1c8      9dcf1fb8      str w29, [x28, -4]!
0x0000b1cc      fd031c2a      mov w29, w28
0x0000b1d0      9d4740b8      ldr w29, [x28], 4
0x0000b1d4      295d4311      add w9, w9, 0xd7, lsl 12
0x0000b1d8      29412f11      add w9, w9, 0xbd0              ; calculate address of #MessageBoxA@16
0x0000b1dc      23fbff97      bl 0x9e68                      ;[1]
0x0000b1e0      20021fd6      br x17
```

API Hooking through modifying jump stubs

We show an example of invisible API hooking through modifying MessageBoxA’s jump stub

API Hooking through modifying the jump stub code

Example: cache file of MessageBoxA in the previous slide



```
[xAdvc]0 0% 255 USER32.DLL.B762FE91071D23DA8720F34E3667A5AB.31468294266C99D8935B35F6F76A0DF7.mp.1.jc
;-- x86.00403db0:
0x0000b1c8        9dcf1fb8        str w29, [x28, -4]!
0x0000b1cc        fd031c2a        mov w29, w28
0x0000b1d0        9d4740b8        ldr w29, [x28], 4
0x0000b1d4        295d4311        add w9, w9, 0xd7, lsl 12
   00b1d8         29412f11        add w9, w9, 0xbd0             ; calculate address of #MessageBoxA@16
   00b1dc         23fbff97        bl 0x9e68                     ;[1]
   00b1e0         20021fd6        br x17
```

```
   ]0 0% 155 USER32.DLL.B762FE91071D23DA8720F34E3667A5AB.31468294266C99D8935B35F6F76A0DF7.mp.1.jc
  x86.00 03db0:
0x0000b1c8        9dcf1fb8        str w29, [x28, -4]!
0x0000b1cc        fd031c2a        mov w29, w28
0x0000b1d0        9d4740b8        ldr w29, [x28], 4
0x0000b1d4        c0000010        adr x0, str.Hoooooooked
0x0000b1d8        800b00b9        str w0, [x28, 8]
0x0000b1dc        295d4311        add w9, w9, 0xd7, lsl 12
0x0000b1e0        29412f11        add w9, w9, 0xbd0
0x0000b1e4        21fbff97        bl 0x9e68                     ;[1]
0x0000b1e8        20021fd6        br x17
;-- "Hoooooooked!!!!!!":
0x0000b1ec       .string "Hoooooooked!!!!!!" ; len=18
```

Hooking code is injected

Second argument is modified!
Displayed message changes

Hello

Hoooooooked!!!!!!

API Hooking example is included in

https://github.com/FFRI/XtaTools/tree/main/example

Small patches applied during XTA cache file update

Update feature of XTA cache files

xtac.exe updates XTA cache files to add newly-translated result

Previous translation result is copied to new cache file

- to reduce the amount of binary translation by xtac.exe

Before copying, small patches are applied to previous result

Update feature of XTA cache files

xtac.exe updates XTA cache files to add newly-translated result

Previous translation result is copied to new cache file

- to reduce the amount of binary translation by xtac.exe

Before copying, small patches are applied to previous result

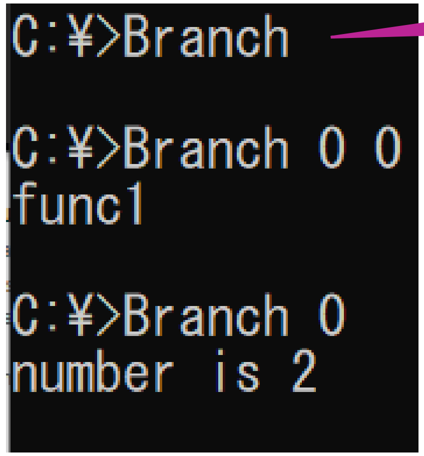
What are these patches?

Example program (Branch.exe)

calls different function depending on the number of arguments

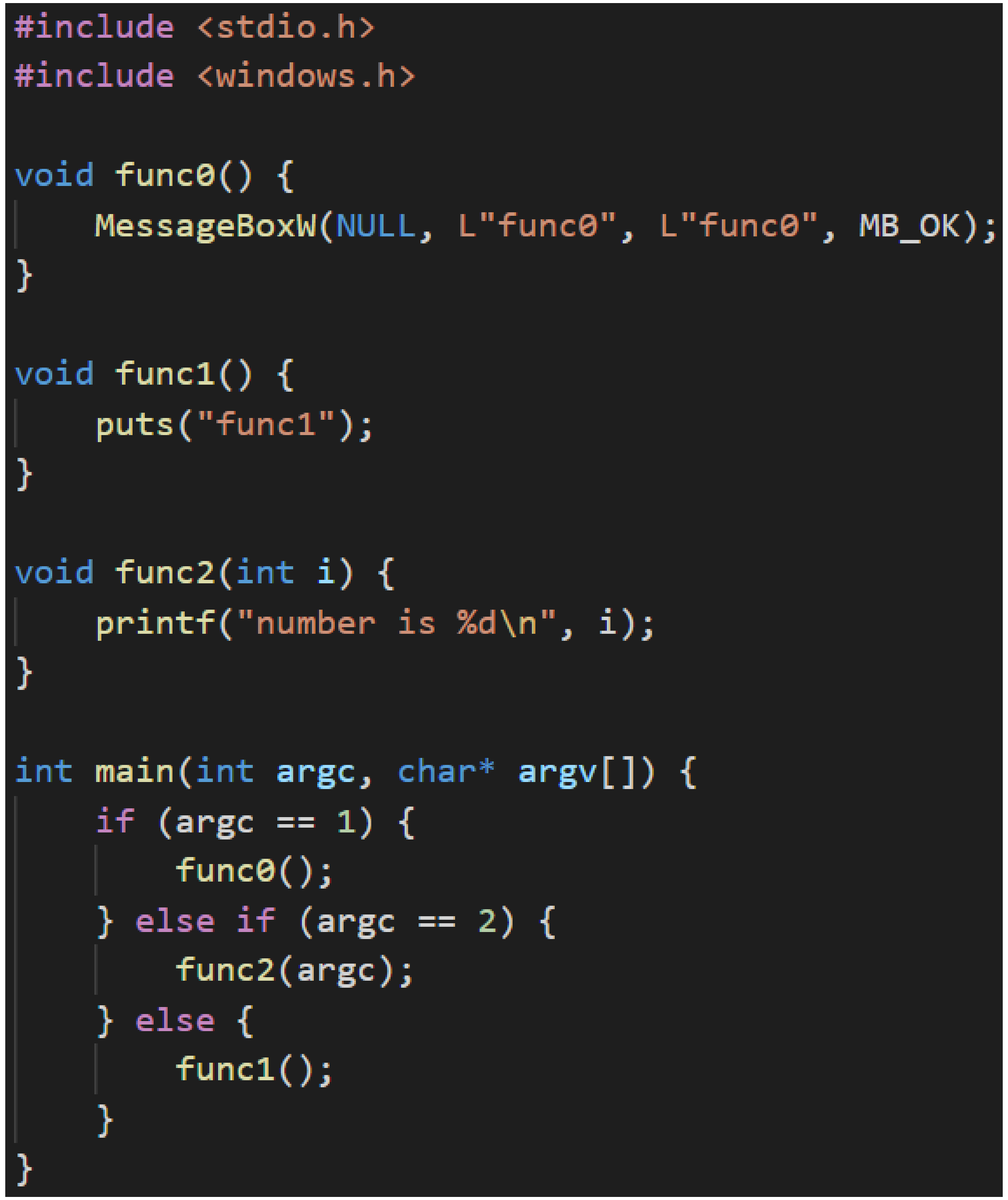
- assuming that func0, func1, and func2 are not inlined by the compiler optimization

We can get three different cache files by changing the number of arguments



```
C:¥>Branch

C:¥>Branch 0 0
func1

C:¥>Branch 0
number is 2
```

xtac makes BRANCH.EXE.*.*.mp.1.jc



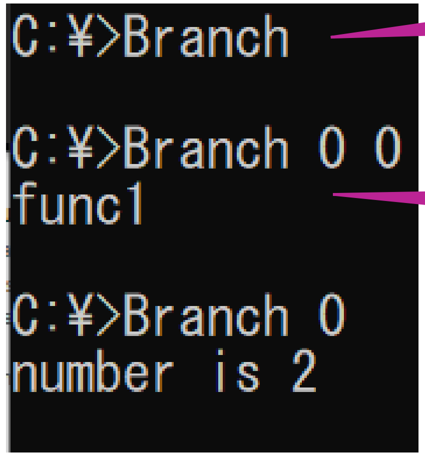
```c
#include <stdio.h>
#include <windows.h>

void func0() {
    MessageBoxW(NULL, L"func0", L"func0", MB_OK);
}

void func1() {
    puts("func1");
}

void func2(int i) {
    printf("number is %d\n", i);
}

int main(int argc, char* argv[]) {
    if (argc == 1) {
        func0();
    } else if (argc == 2) {
        func2(argc);
    } else {
        func1();
    }
}
```

Example program (Branch.exe)

calls different function depending on the number of arguments

- assuming that func0, func1, and func2 are not inlined by the compiler optimization

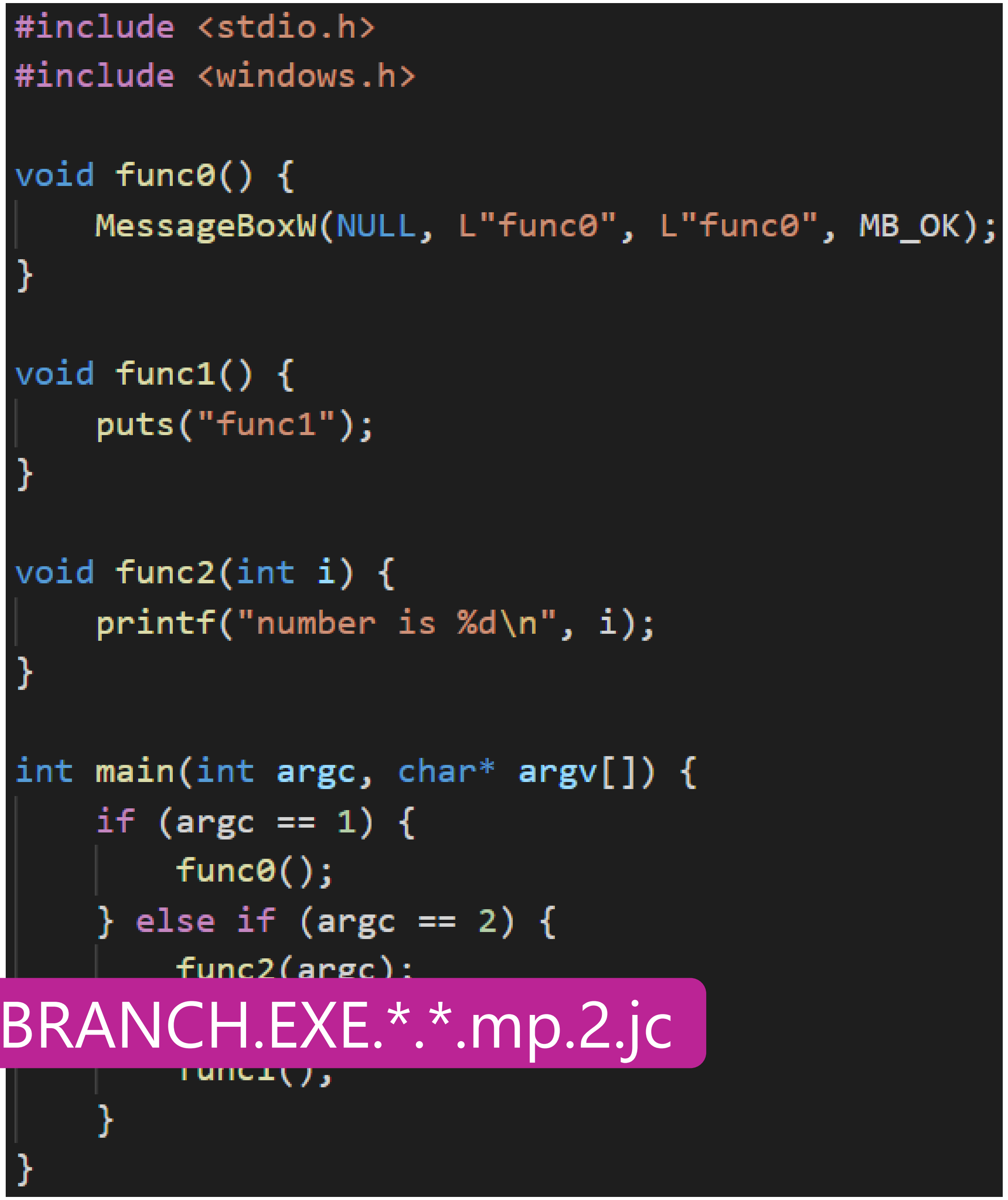
We can get three different cache files by changing the number of arguments



```
C:¥>Branch

C:¥>Branch 0 0
func1

C:¥>Branch 0
number is 2
```

xtac makes BRANCH.EXE.*.*.mp.1.jc

xtac updates the cache file and makes BRANCH.EXE.*.*.mp.2.jc



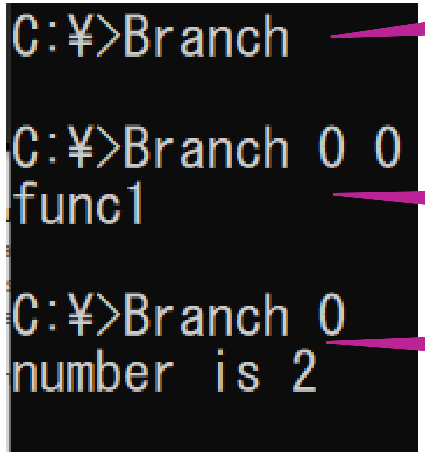
```
#include <stdio.h>
#include <windows.h>

void func0() {
    MessageBoxW(NULL, L"func0", L"func0", MB_OK);
}

void func1() {
    puts("func1");
}

void func2(int i) {
    printf("number is %d\n", i);
}

int main(int argc, char* argv[]) {
    if (argc == 1) {
        func0();
    } else if (argc == 2) {
        func2(argc);

        func1();
    }
}
```

Example program (Branch.exe)

calls different function depending on the number of arguments

- assuming that func0, func1, and func2 are not inlined by the compiler optimization

We can get three different cache files by changing the number of arguments

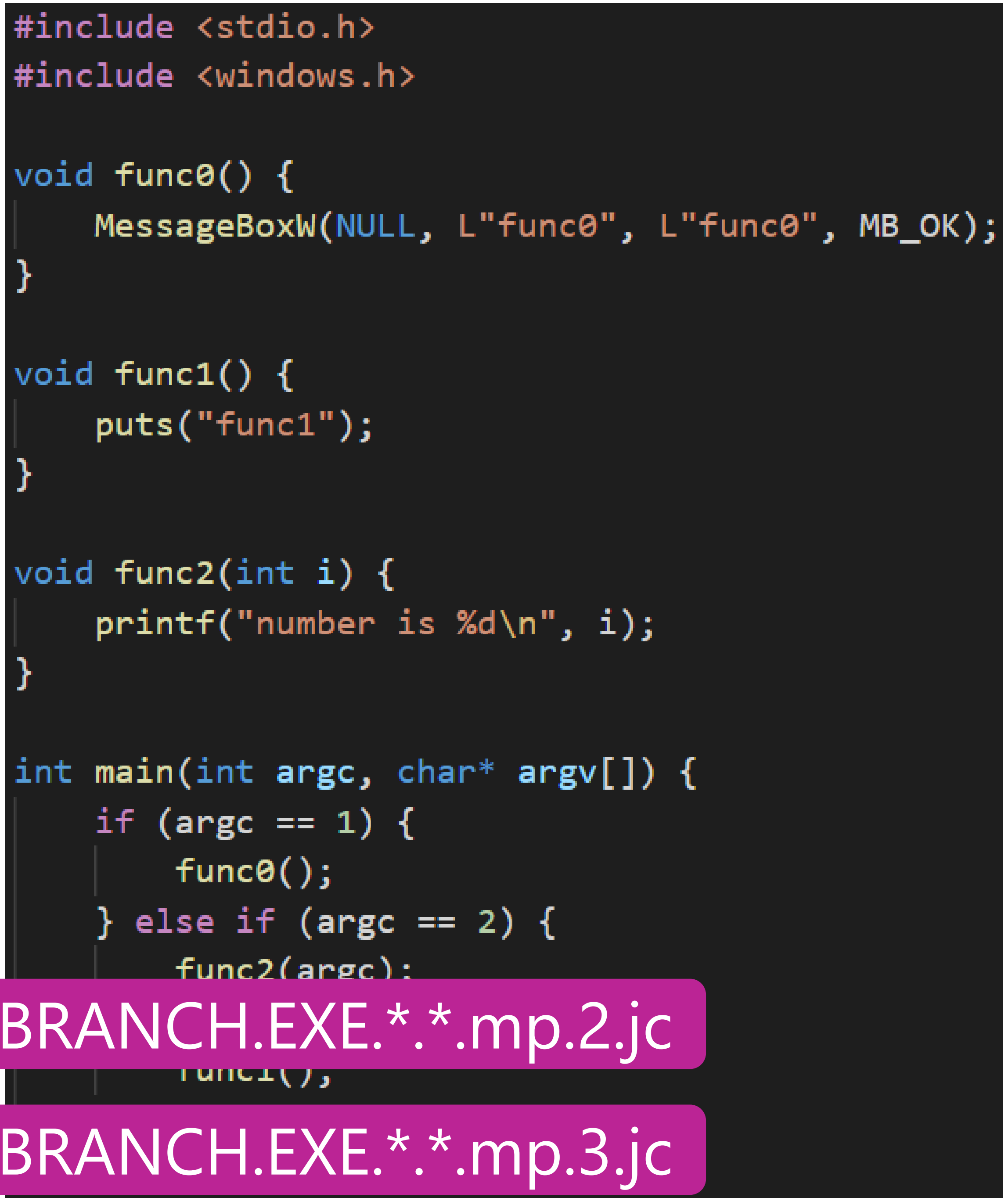


```
C:¥>Branch

C:¥>Branch 0 0
func1

C:¥>Branch 0
number is 2
```

xtac makes BRANCH.EXE.*.*.mp.1.jc

xtac updates the cache file and makes BRANCH.EXE.*.*.mp.2.jc

xtac updates the cache file and makes BRANCH.EXE.*.*.mp.3.jc



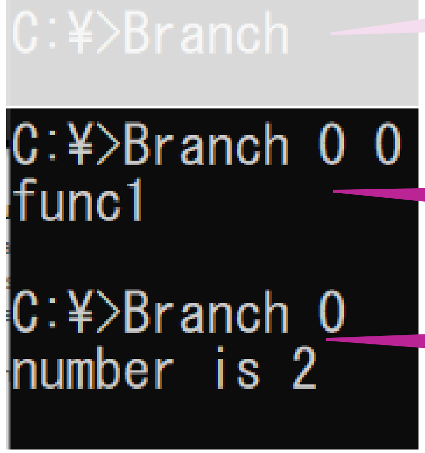
```c
#include <stdio.h>
#include <windows.h>

void func0() {
    MessageBoxW(NULL, L"func0", L"func0", MB_OK);
}

void func1() {
    puts("func1");
}

void func2(int i) {
    printf("number is %d\n", i);
}

int main(int argc, char* argv[]) {
    if (argc == 1) {
        func0();
    } else if (argc == 2) {
        func2(argc);
        func1();
```

Example program (Branch.exe)

calls different function depending on the number of arguments

- assuming that func0, func1, and func2 are not inlined by the compiler optimization

We can get three different
changing the number of a



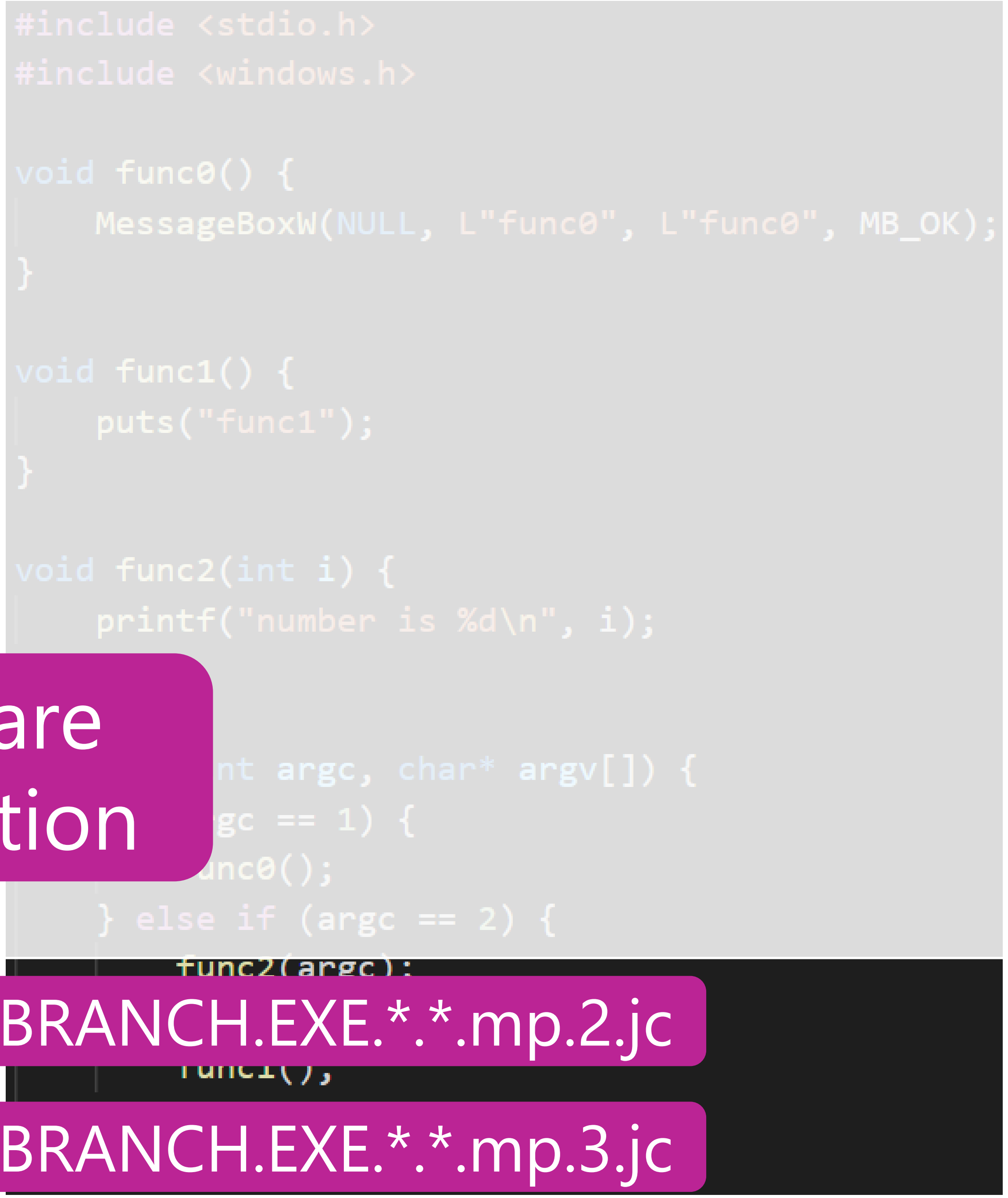
```
C:¥>Branch

C:¥>Branch 0 0
func1

C:¥>Branch 0
number is 2
```

These two files are used for explanation

xtac makes BRANCH.E .mp.1.jc

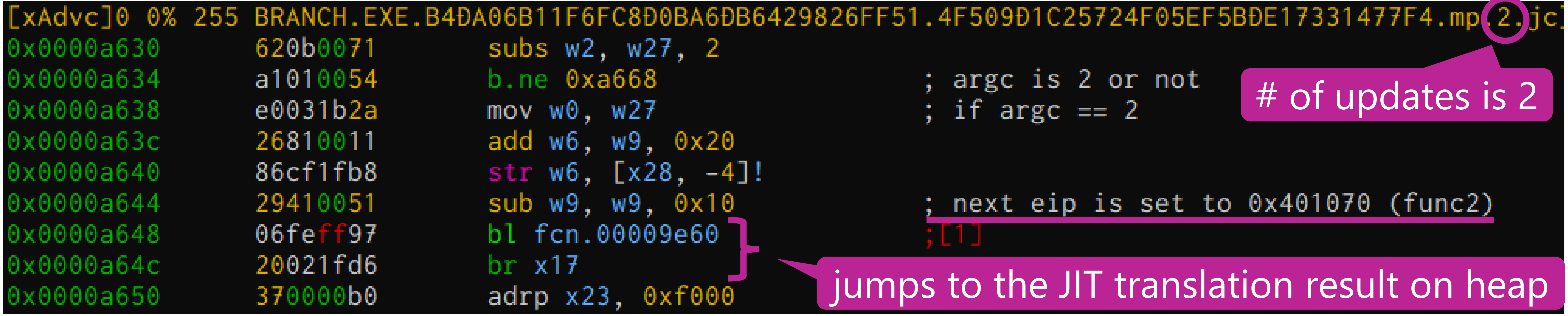
xtac updates the cache file and makes BRANCH.EXE.*.*.mp.2.jc

xtac updates the cache file and makes BRANCH.EXE.*.*.mp.3.jc



```
#include <stdio.h>
#include <windows.h>

void func0() {
    MessageBoxW(NULL, L"func0", L"func0", MB_OK);
}

void func1() {
    puts("func1");
}

void func2(int i) {
    printf("number is %d\n", i);

             nt argc, char* argv[]) {
             gc == 1) {
            unc0();
    } else if (argc == 2) {
        func2(argc);

        func1();
```

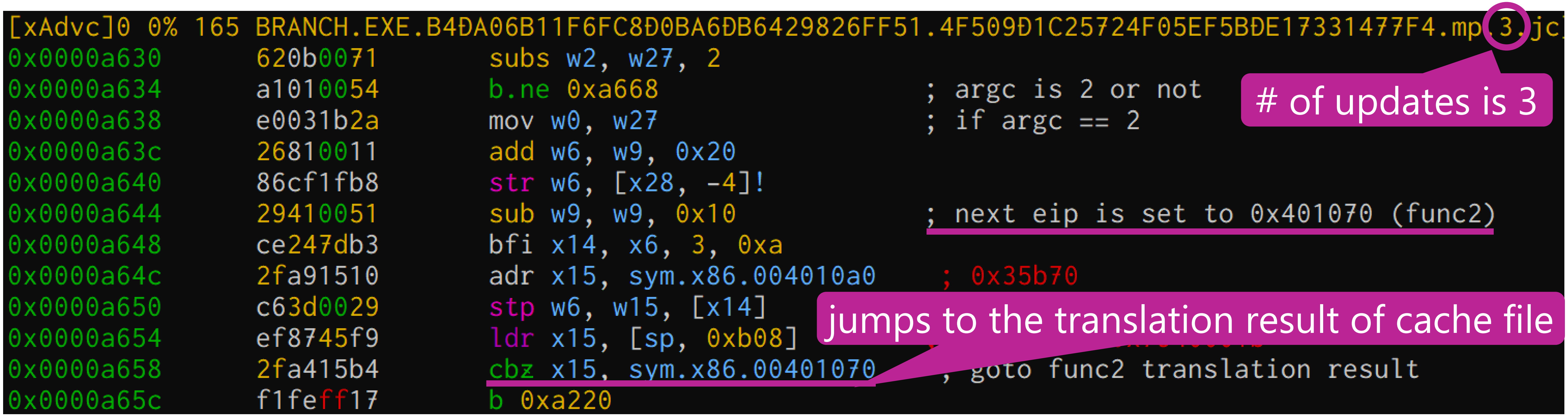
Difference between two XTA cache files



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc]
0x0000a630      620b0071        subs w2, w27, 2
0x0000a634      a1010054        b.ne 0xa668                     ; argc is 2 or not
0x0000a638      e0031b2a        mov w0, w27                     ; if argc == 2
0x0000a63c      26810011        add w6, w9, 0x20
0x0000a640      86cf1fb8        str w6, [x28, -4]!
0x0000a644      29410051        sub w9, w9, 0x10                ; next eip is set to 0x401070 (func2)
0x0000a648      06feff97        bl fcn.00009e60                 ;[1]
0x0000a64c      20021fd6        br x17
0x0000a650      370000b0        adrp x23, 0xf000
```

# of updates is 2

jumps to the JIT translation result on heap

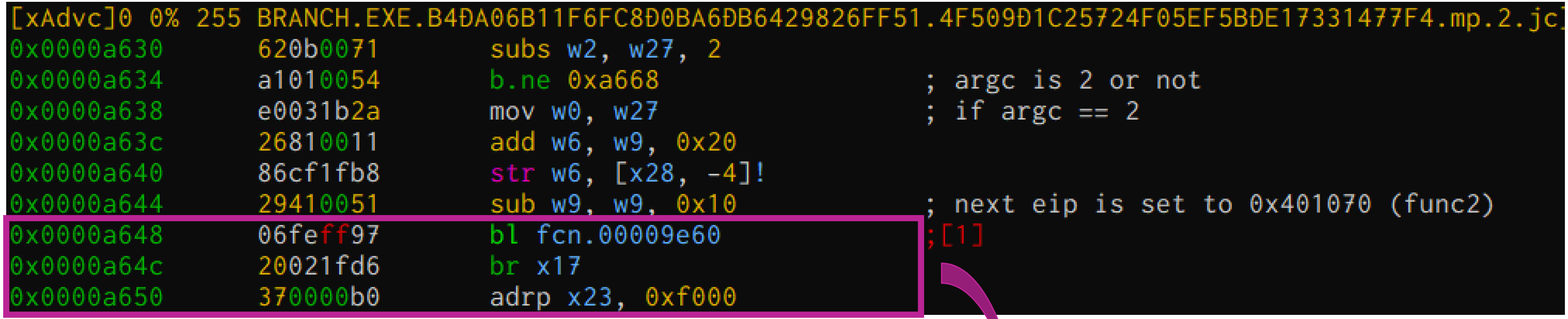


```
[xAdvc]0 0% 165 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.3.jc]
0x0000a630      620b0071        subs w2, w27, 2
0x0000a634      a1010054        b.ne 0xa668                     ; argc is 2 or not
0x0000a638      e0031b2a        mov w0, w27                     ; if argc == 2
0x0000a63c      26810011        add w6, w9, 0x20
0x0000a640      86cf1fb8        str w6, [x28, -4]!
0x0000a644      29410051        sub w9, w9, 0x10                ; next eip is set to 0x401070 (func2)
0x0000a648      ce247db3        bfi x14, x6, 3, 0xa
0x0000a64c      2fa91510        adr x15, sym.x86.004010a0        ; 0x35b70
0x0000a650      c63d0029        stp w6, w15, [x14]
0x0000a654      ef8745f9        ldr x15, [sp, 0xb08]
0x0000a658      2fa415b4        cbz x15, sym.x86.00401070        ; goto func2 translation result
0x0000a65c      f1feff17        b 0xa220
```

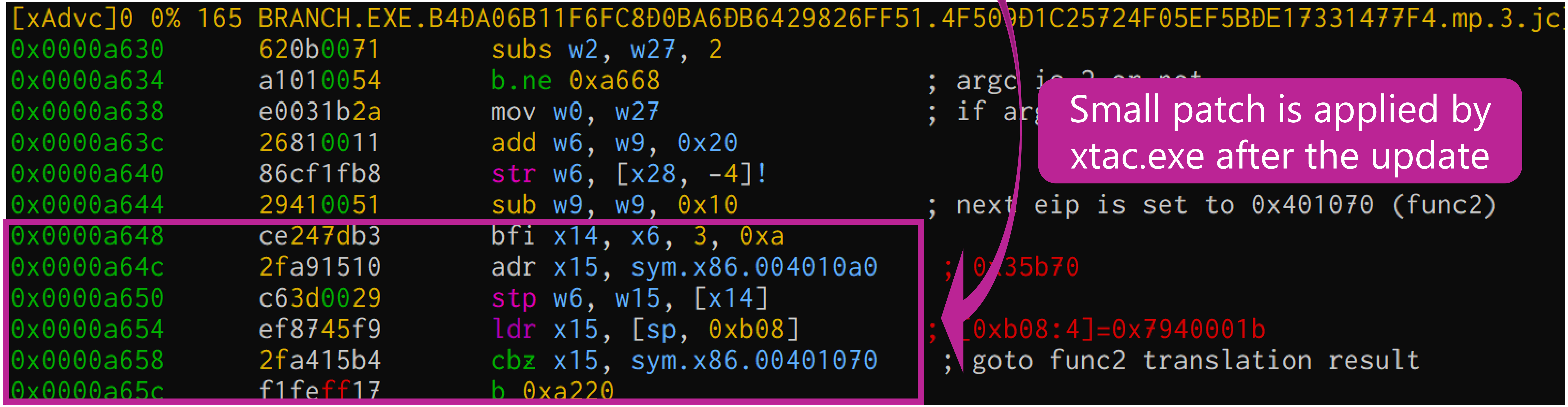
# of updates is 3

jumps to the translation result of cache file

Difference between two XTA cache files



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc
0x0000a630       620b0071       subs w2, w27, 2
0x0000a634       a1010054       b.ne 0xa668                    ; argc is 2 or not
0x0000a638       e0031b2a       mov w0, w27                    ; if argc == 2
0x0000a63c       26810011       add w6, w9, 0x20
0x0000a640       86cf1fb8       str w6, [x28, -4]!
0x0000a644       29410051       sub w9, w9, 0x10               ; next eip is set to 0x401070 (func2)
0x0000a648       06feff97       bl fcn.00009e60                ;[1]
0x0000a64c       20021fd6       br x17
0x0000a650       370000b0       adrp x23, 0xf000
```



```
[xAdvc]0 0% 165 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.3.jc
0x0000a630       620b0071       subs w2, w27, 2
0x0000a634       a1010054       b.ne 0xa668                    ; argc i
0x0000a638       e0031b2a       mov w0, w27                    ; if ar
0x0000a63c       26810011       add w6, w9, 0x20
0x0000a640       86cf1fb8       str w6, [x28, -4]!
0x0000a644       29410051       sub w9, w9, 0x10               ; next eip is set to 0x401070 (func2)
0x0000a648       ce247db3       bfi x14, x6, 3, 0xa
0x0000a64c       2fa91510       adr x15, sym.x86.004010a0       ; 0x35b70
0x0000a650       c63d0029       stp w6, w15, [x14]
0x0000a654       ef8745f9       ldr x15, [sp, 0xb08]           ; [0xb08:4]=0x7940001b
0x0000a658       2fa415b4       cbz x15, sym.x86.00401070       ; goto func2 translation result
0x0000a65c       f1feff17       b 0xa220
```

Small patch is applied by xtac.exe after the update

What is this patch for?

BRANCH.EXE.*.*.mp.3.jc **contains translation result of func2**, but BRANCH.*.*.mp.2.jc **does not contain translation result of func2**

- because translation result of func2 is added after the update of BRANCH.*.*.mp.2.jc

When using BRANCH.EXE.*.*.mp.2.jc …

- should jump to the JIT translation result on heap when calling func2

When using BRANCH.EXE.*.*.mp.3.jc …

- **can directly jump** to the translation result of XTA cache file when calling func2

This patch changes the jump to func2 from …

- JIT translation result on heap -> translation result of XTA cache file

What is this patch for?

BRANCH.EXE.*.*.mp.3.jc **contains translation result of func2**, but BRANCH.*.*.mp.2.jc **does not contain translation result of func2**

- because translation result of func2 is added after the update of BRANCH.*.*.mp.2.jc

When using BRANCH.EXE.*.*.mp.2.jc …

- should jump to the JIT translation result on heap when calling func2

When using BRANCH.EXE.*.*.mp.3.jc …

- **can directly jump** to the translation result of XTA cache file when calling func2

It reduces the amount of JIT binary translation

This patch changes the jump to func2 from …

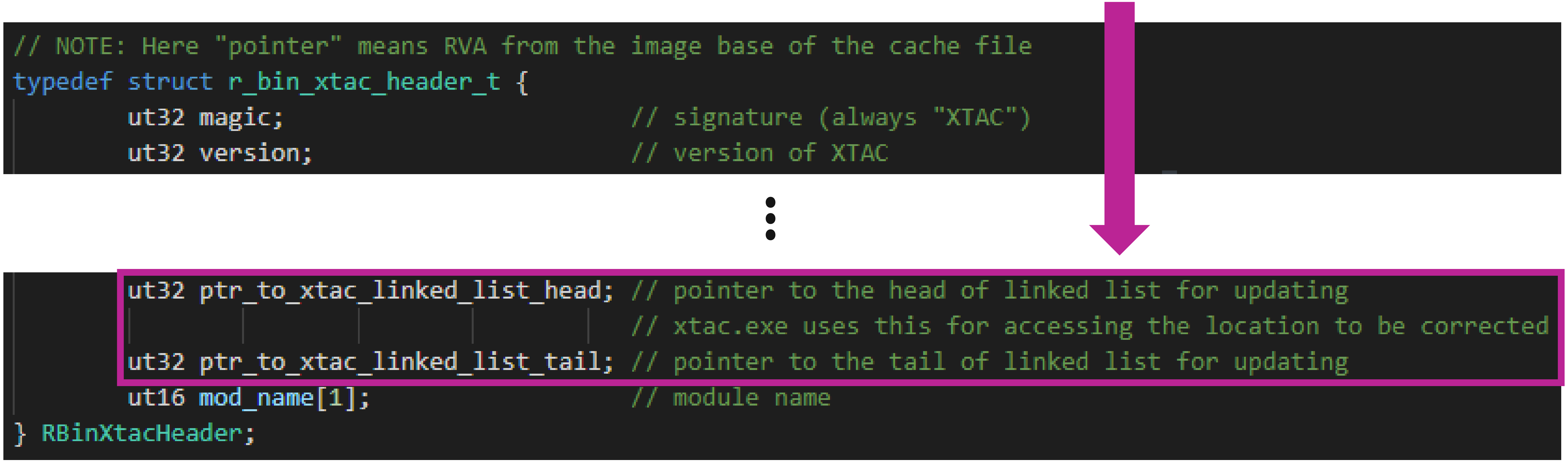
- JIT translation result on heap -> translation result of XTA cache file

How does xtac.exe get the positions to be patched?

XTA cache file header has the member to access the positions to be patched

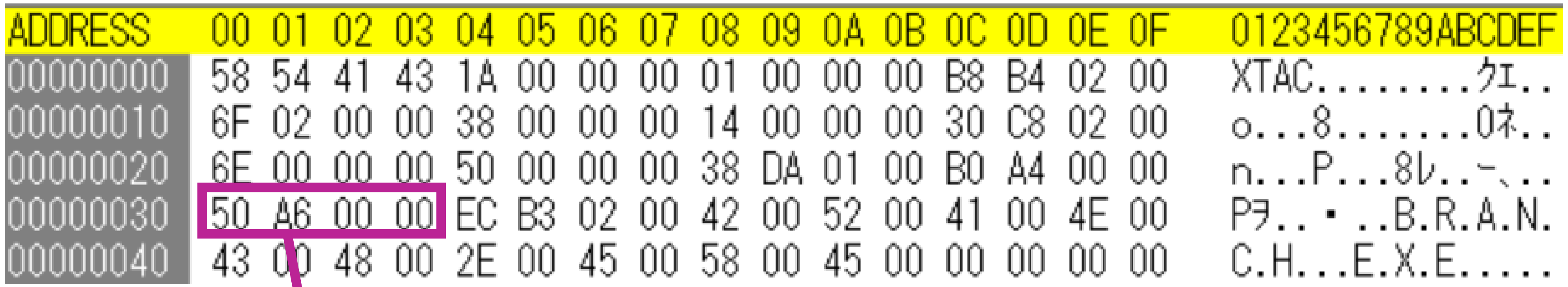
- These positions are stored as a linked list (we are calling it **XTAC linked list**)

The linked list can be accessed by the following cache file header members



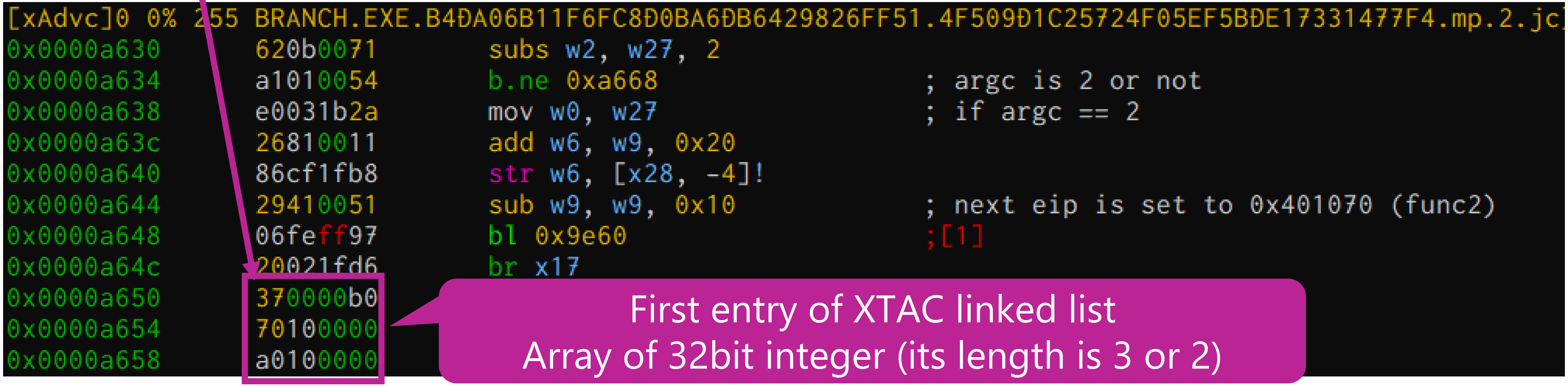
```
// NOTE: Here "pointer" means RVA from the image base of the cache file
typedef struct r_bin_xtac_header_t {
        ut32 magic;                     // signature (always "XTAC")
        ut32 version;                   // version of XTAC
⋮
        ut32 ptr_to_xtac_linked_list_head; // pointer to the head of linked list for updating
                                           // xtac.exe uses this for accessing the location to be corrected
        ut32 ptr_to_xtac_linked_list_tail; // pointer to the tail of linked list for updating
        ut16 mod_name[1];                  // module name
} RBinXtacHeader;
```

XTAC linked list



```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00000000  58 54 41 43 1A 00 00 00 01 00 00 00 B8 B4 02 00   XTAC........ｸｴ..
00000010  6F 02 00 00 38 00 00 00 14 00 00 00 30 C8 02 00   o...8.......0ﾈ..
00000020  6E 00 00 00 50 00 00 00 38 DA 01 00 B0 A4 00 00   n...P...8ﾚ..ｰ､..
00000030  50 A6 00 00 EC B3 02 00 42 00 52 00 41 00 4E 00   Pｦ..・..B.R.A.N.
00000040  43 00 48 00 2E 00 45 00 58 00 45 00 00 00 00 00   C.H...E.X.E.....
```

Pointer to
XTAC linked list head

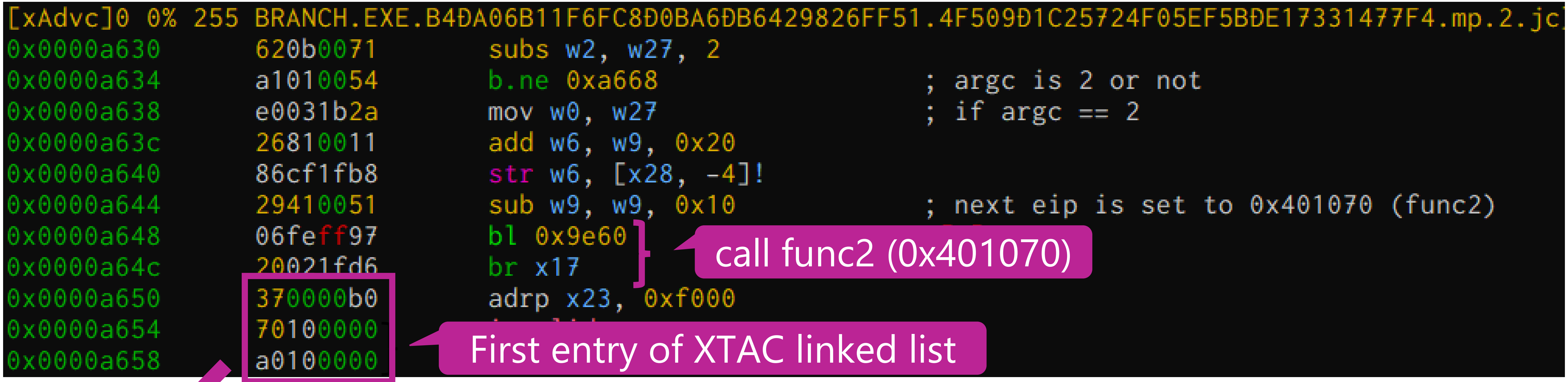


```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc]
0x0000a630      620b0071        subs w2, w27, 2
0x0000a634      a1010054        b.ne 0xa668                     ; argc is 2 or not
0x0000a638      e0031b2a        mov w0, w27                     ; if argc == 2
0x0000a63c      26810011        add w6, w9, 0x20
0x0000a640      86cf1fb8        str w6, [x28, -4]!
0x0000a644      29410051        sub w9, w9, 0x10                ; next eip is set to 0x401070 (func2)
0x0000a648      06feff97        bl 0x9e60                       ;[1]
0x0000a64c      20021fd6        br x17
0x0000a650      370000b0
0x0000a654      70100000
0x0000a658      a0100000
```

First entry of XTAC linked list
Array of 32bit integer (its length is 3 or 2)

Note: above disassembly is the same as previous one

Member of XTAC linked list entry



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc
0x0000a630        620b0071        subs w2, w27, 2
0x0000a634        a1010054        b.ne 0xa668                          ; argc is 2 or not
0x0000a638        e0031b2a        mov w0, w27                          ; if argc == 2
0x0000a63c        26810011        add w6, w9, 0x20
0x0000a640        86cf1fb8        str w6, [x28, -4]!
0x0000a644        29410051        sub w9, w9, 0x10                     ; next eip is set to 0x401070 (func2)
0x0000a648        06feff97        bl  0x9e60
0x0000a64c        20021fd6        br  x17
0x0000a650        370000b0        adrp x23, 0xf000
0x0000a654        70100000
0x0000a658        a0100000
```

call func2 (0x401070)

First entry of XTAC linked list

As uint32 array

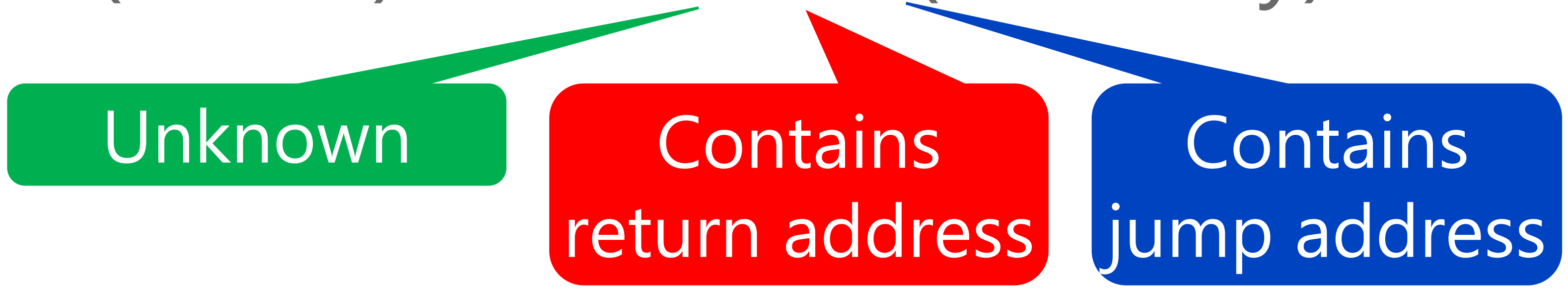
Note: above disassembly is the same as previous one

b0000037: **Meta data (see next slide)** and **quarter of offset to the next entry**
00001070: x86 RVA of jump address (containing RVA of func2 in this case)
000010a0: x86 RVA of return address (containing RVA of return address of this call site)

Member of XTAC linked list entry (contd.)

b0000037: Meta data and quarter of offset to the next entry

0x**b** (in hex) -> 0b**1011** (in binary)

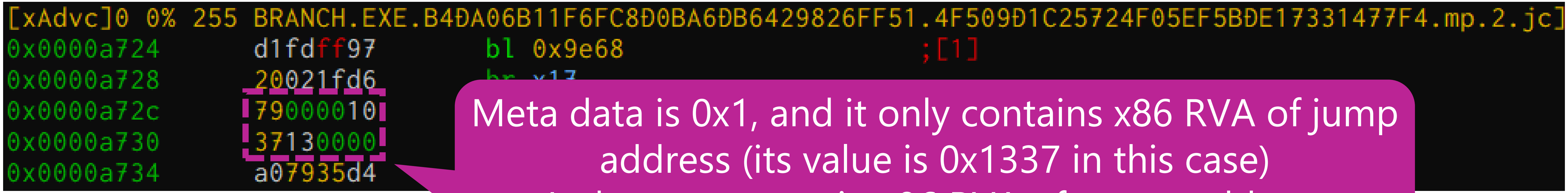


Unknown

Contains return address

Contains jump address

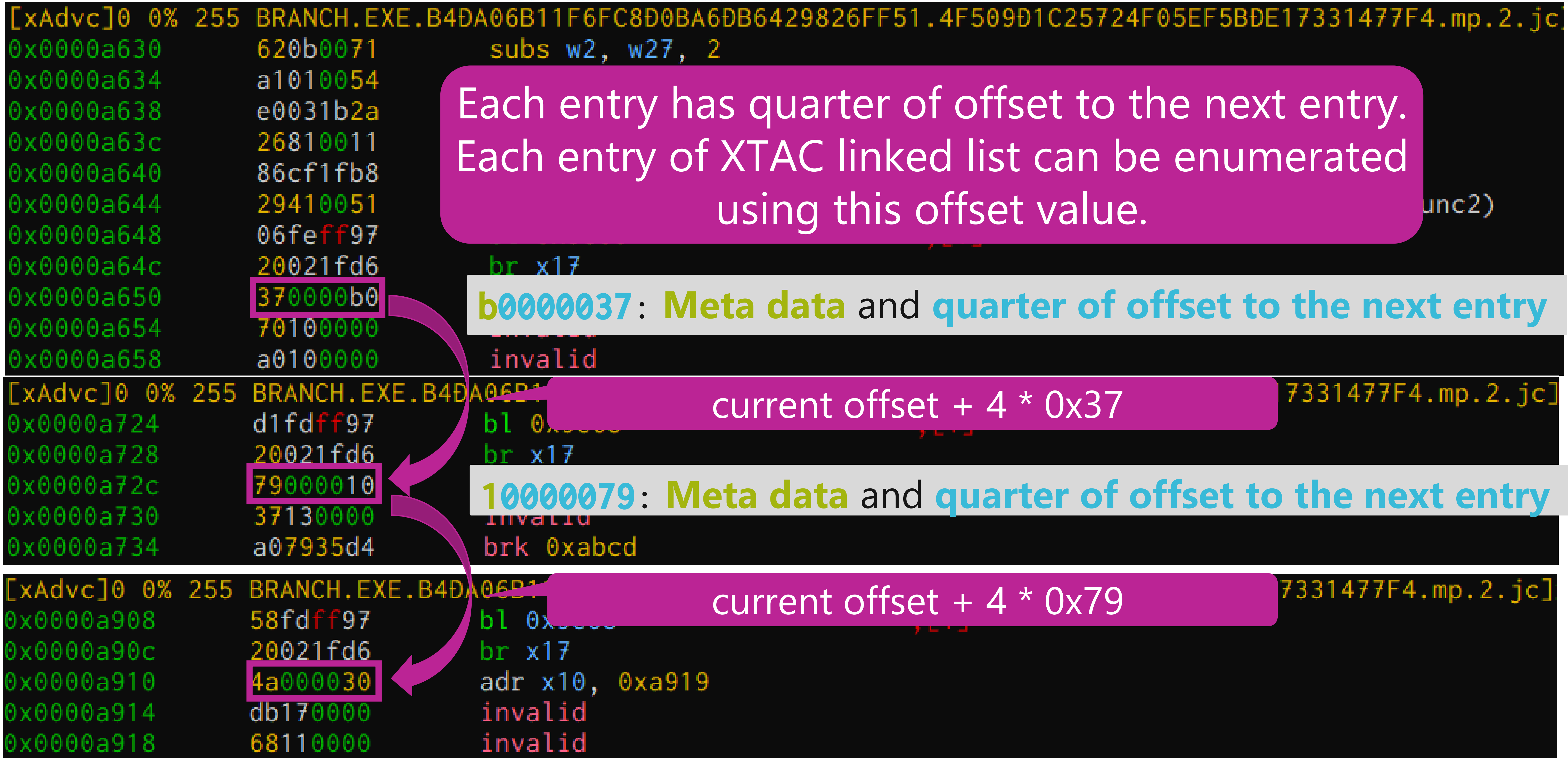
If the meta data is 0x**1**, it contains only jump address (no return address)



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc]
0x0000a724        d1fdff97        bl 0x9e68                    ;[1]
0x0000a728        20021fd6        br x17
0x0000a72c        79000010
0x0000a730        37130000
0x0000a734        a07935d4
```

Meta data is 0x1, and it only contains x86 RVA of jump address (its value is 0x1337 in this case)
It does not contain x86 RVA of return address

Offset to the next entry of linked list



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.2.jc
0x0000a630          620b0071        subs w2, w27, 2
0x0000a634          a1010054
0x0000a638          e0031b2a
0x0000a63c          26810011
0x0000a640          86cf1fb8
0x0000a644          29410051                                          unc2)
0x0000a648          06feff97
0x0000a64c          20021fd6        br x17
0x0000a650          370000b0
0x0000a654          70100000
0x0000a658          a0100000        invalid
```

```
[xAdvc]0  0% 255 BRANCH.EXE.B4DA06B1                        7331477F4.mp.2.jc]
0x0000a724          d1fdff97        bl  0x
0x0000a728          20021fd6        br  x17
0x0000a72c          79000010
0x0000a730          37130000
0x0000a734          a07935d4        brk  0xabcd
```

```
[xAdvc]0  0% 255 BRANCH.EXE.B4DA06B1                        7331477F4.mp.2.jc]
0x0000a908          58fdff97        bl  0x
0x0000a90c          20021fd6        br  x17
0x0000a910          4a000030        adr  x10,  0xa919
0x0000a914          db170000        invalid
0x0000a918          68110000        invalid
```

Each entry has quarter of offset to the next entry.
Each entry of XTAC linked list can be enumerated using this offset value.

b0000037 : Meta data and quarter of offset to the next entry

current offset + 4 * 0x37

10000079 : Meta data and quarter of offset to the next entry

current offset + 4 * 0x79

Technical details of XTA cache hijacking

Notes about injectable payload of XTA cache hijacking

There are no restrictions of:

- size of code
- encoding of code

Both x86 and ARM64 code can be injected!

- x86 shellcode can be executed by calling thread creation function (such as CreateThread and NtCreateThread)

Notes about building shellcode for XTA cache hijacking

Pay special attentions about Windows API calls

- Windows API calls through emulation layer is preferred
  - Function call through emulation layer unlikely causes program crashes
  - Function call that is performed **not through emulation layer** causes program crashes in some cases (this limitation has already been noted here. APIs of GDI or Winsock are not callable.)

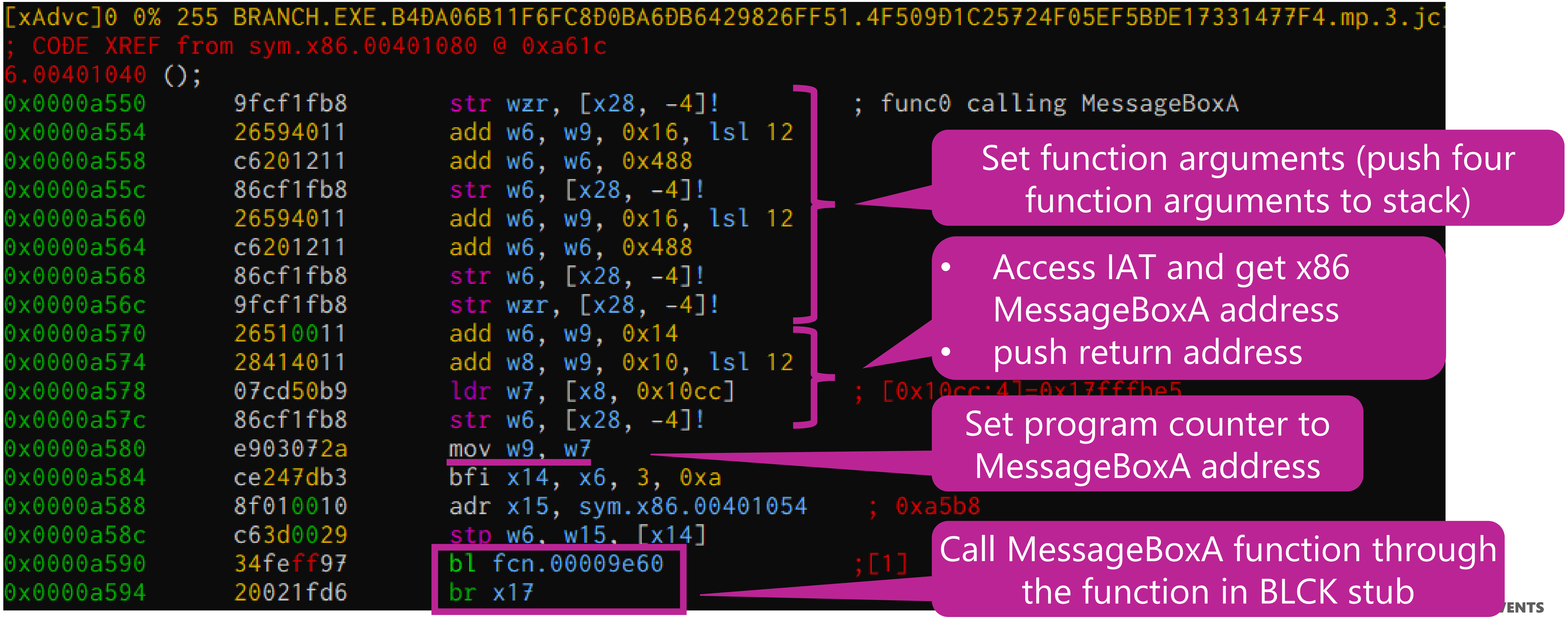
Steps to call Windows API through emulation layer

1. push function arguments to stack (x86 calling convention)
2. push x86 return address to stack (lr register is not used!)
3. get x86 Windows API address through accessing IAT (or PEB)
4. set program counter (w9 register during emulation) to Windows API address
5. call API through a specific function in BLCK stub

(see next slide)

Example of Windows API call through emulation layer

Cache file of this sample program (show only translation result of func0)



```
[xAdvc]0 0% 255 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F509D1C25724F05EF5BDE17331477F4.mp.3.jc
; CODE XREF from sym.x86.00401080 @ 0xa61c
6.00401040 ();
0x0000a550      9fcf1fb8        str wzr, [x28, -4]!           ; func0 calling MessageBoxA
0x0000a554      26594011        add w6, w9, 0x16, lsl 12
0x0000a558      c6201211        add w6, w6, 0x488
0x0000a55c      86cf1fb8        str w6, [x28, -4]!
0x0000a560      26594011        add w6, w9, 0x16, lsl 12
0x0000a564      c6201211        add w6, w6, 0x488
0x0000a568      86cf1fb8        str w6, [x28, -4]!
0x0000a56c      9fcf1fb8        str wzr, [x28, -4]!
0x0000a570      26510011        add w6, w9, 0x14
0x0000a574      28414011        add w8, w9, 0x10, lsl 12
0x0000a578      07cd50b9        ldr w7, [x8, 0x10cc]          ; [0x10cc:4]=0x17fffbe5
0x0000a57c      86cf1fb8        str w6, [x28, -4]!
0x0000a580      e903072a        mov w9, w7
0x0000a584      ce247db3        bfi x14, x6, 3, 0xa
0x0000a588      8f010010        adr x15, sym.x86.00401054      ; 0xa5b8
0x0000a58c      c63d0029        stp w6, w15, [x14]
0x0000a590      34feff97        bl fcn.00009e60               ;[1]
0x0000a594      20021fd6        br x17
```

Set function arguments (push four function arguments to stack)

- Access IAT and get x86 MessageBoxA address
- push return address

Set program counter to MessageBoxA address

Call MessageBoxA function through the function in BLCK stub

- **Some code injection examples are included in …**
  - https://github.com/FFRI/XtaTools/tree/main/example

- **We also have provided tools to support for building shellcode in the above repository**

Code coverage measurement using XTA cache file

Code coverage can be obtained by examining XTA cache file

because XTA cache file holds x86 RVA addresses that executed

- explained in this slide

Before demonstrating this, we will explain what kind of instruction ends the binary translation unit

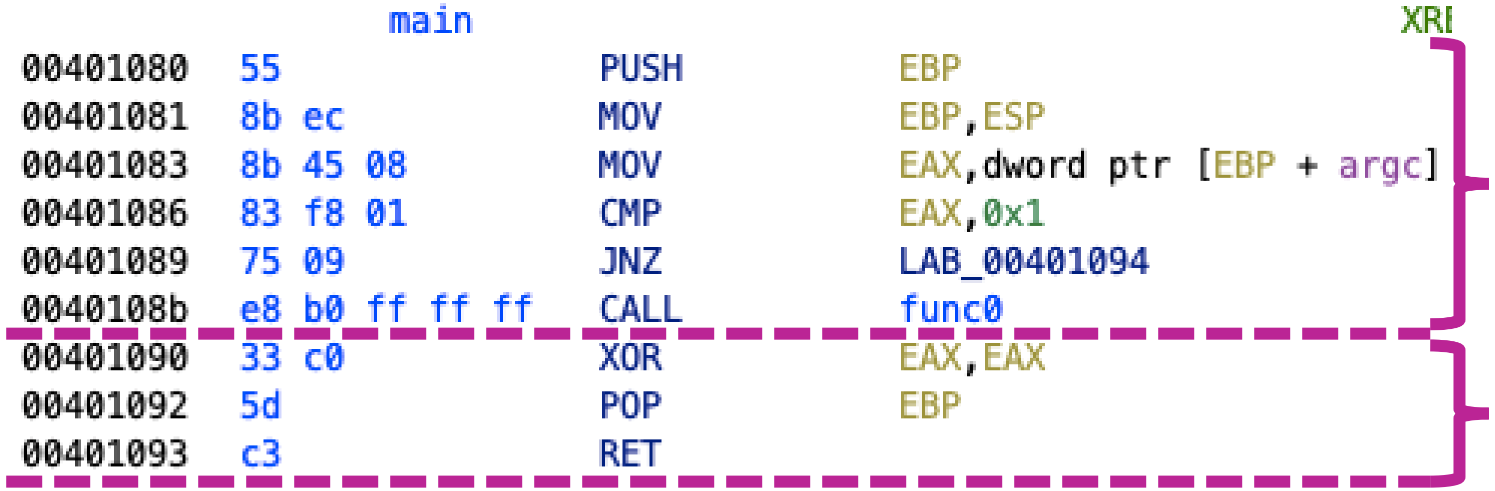
Binary translation unit

x86 code is translated for each code block

- Branch instructions, such as call and ret, end one code block
- However, there are some exceptions:
  - In some case, jmp instructions do not end the code block
  - Some instructions such as x87 instructions and software interrupt instructions end the code block

Example

x86 code of example program



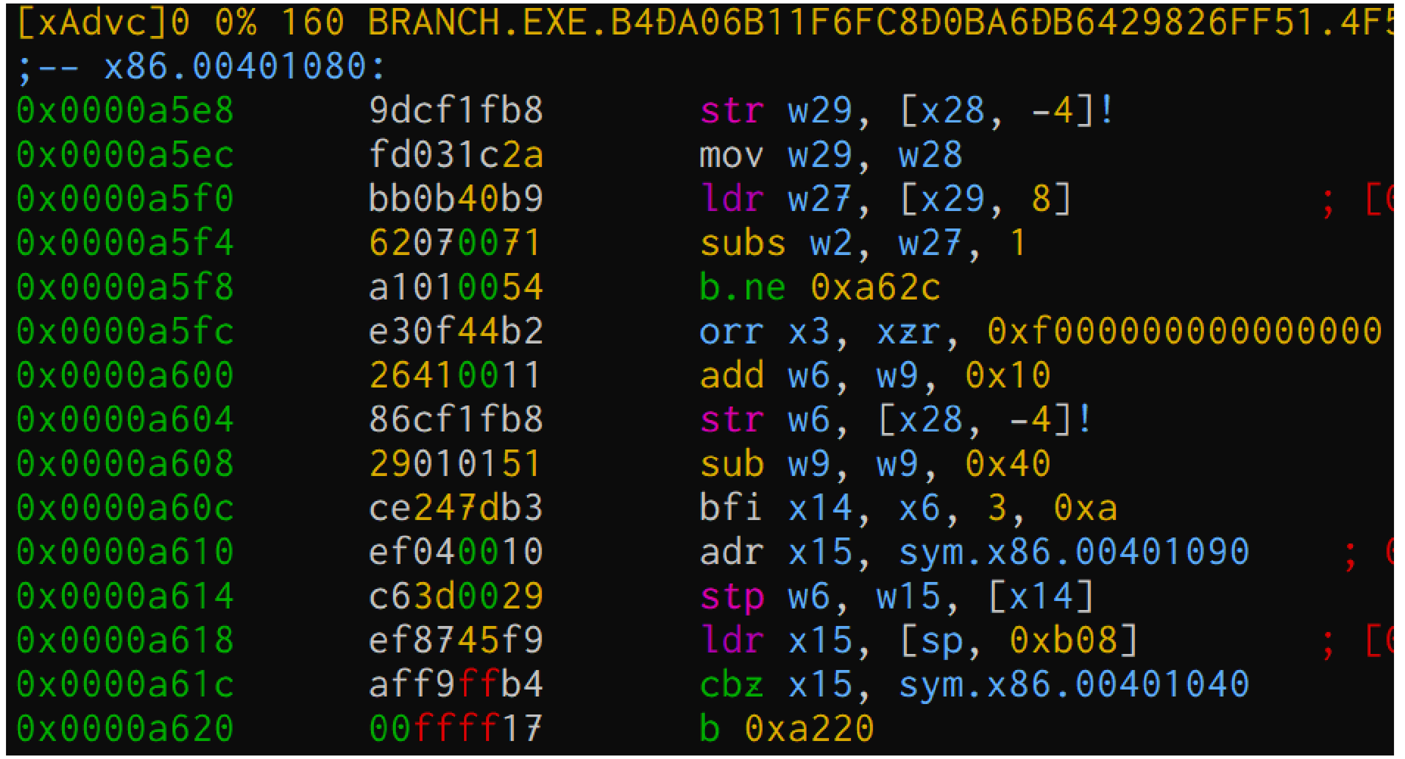
```
                  main                                                   XRE
00401080    55                 PUSH         EBP
00401081    8b ec              MOV          EBP,ESP
00401083    8b 45 08           MOV          EAX,dword ptr [EBP + argc]
00401086    83 f8 01           CMP          EAX,0x1
00401089    75 09              JNZ          LAB_00401094
0040108b    e8 b0 ff ff ff     CALL         func0
00401090    33 c0              XOR          EAX,EAX
00401092    5d                 POP          EBP
00401093    c3                 RET
```

NOTE: func0’s address is 0x401040

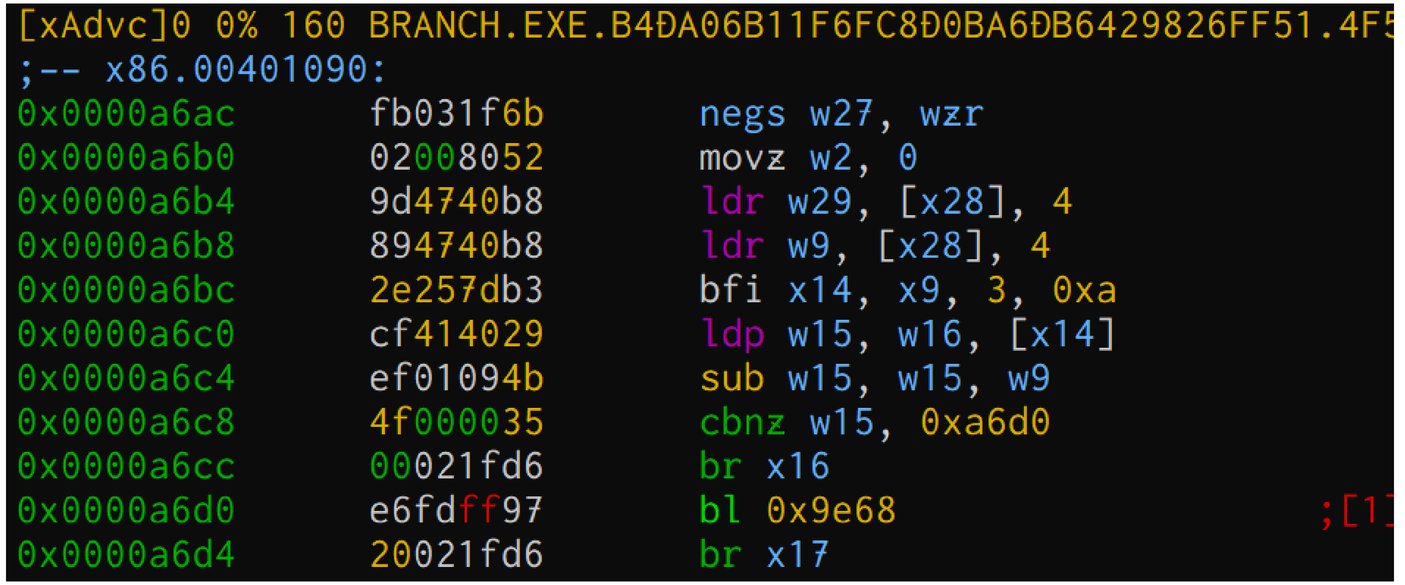
------ end of translation unit

- “call” and “ret” end translation unit
- “jnz” does not end translation unit in this example

Translated ARM64 code



```
[xAdvc]0 0% 160 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F5
;-- x86.00401080:
0x0000a5e8       9dcf1fb8        str w29, [x28, -4]!
0x0000a5ec       fd031c2a        mov w29, w28
0x0000a5f0       bb0b40b9        ldr w27, [x29, 8]                 ; [
0x0000a5f4       62070071        subs w2, w27, 1
0x0000a5f8       a1010054        b.ne 0xa62c
0x0000a5fc       e30f44b2        orr x3, xzr, 0xf000000000000000
0x0000a600       26410011        add w6, w9, 0x10
0x0000a604       86cf1fb8        str w6, [x28, -4]!
0x0000a608       29010151        sub w9, w9, 0x40
0x0000a60c       ce247db3        bfi x14, x6, 3, 0xa
0x0000a610       ef040010        adr x15, sym.x86.00401090          ;
0x0000a614       c63d0029        stp w6, w15, [x14]
0x0000a618       ef8745f9        ldr x15, [sp, 0xb08]              ; [
0x0000a61c       aff9ffb4        cbz x15, sym.x86.00401040
0x0000a620       00ffff17        b 0xa220
```



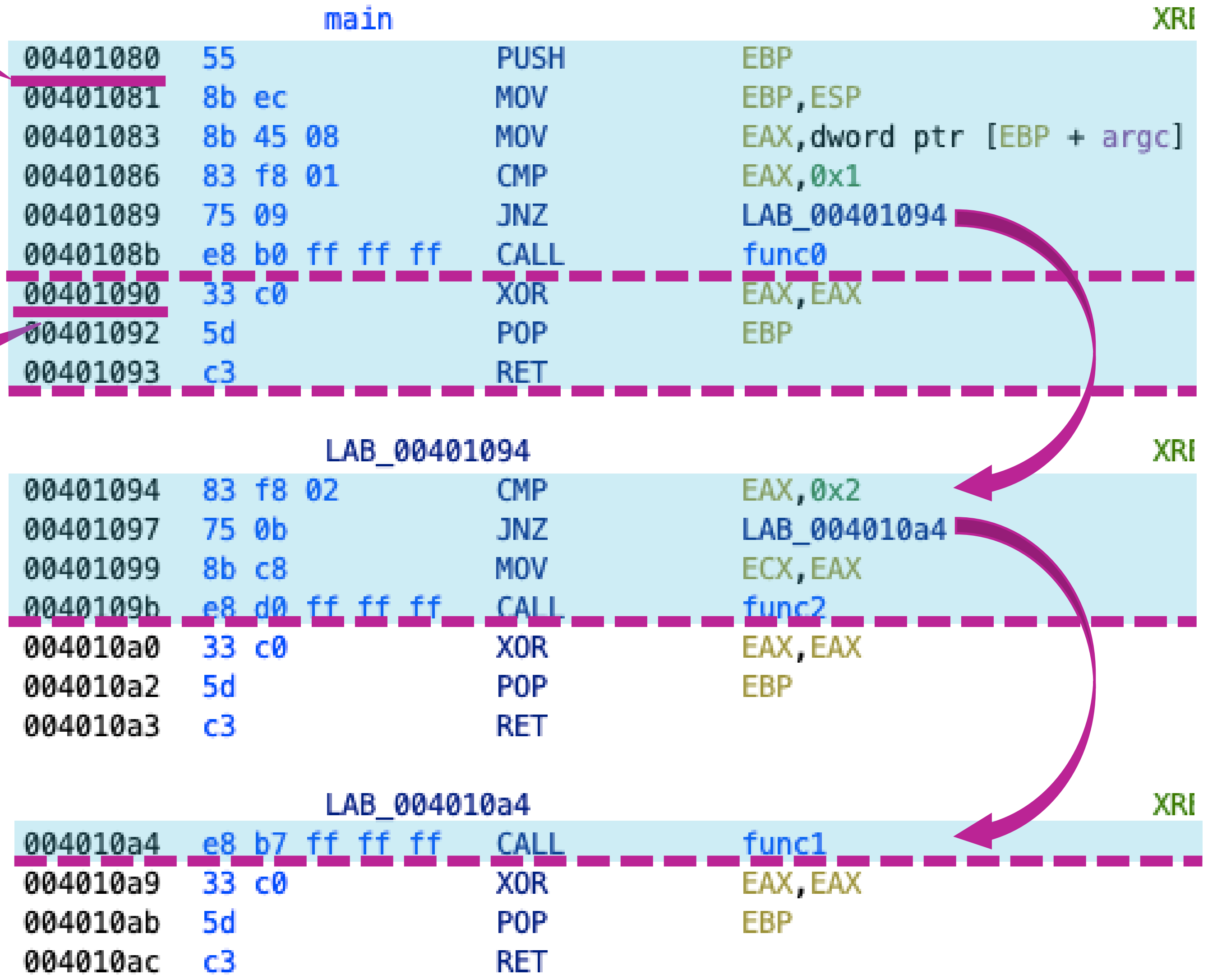
```
[xAdvc]0 0% 160 BRANCH.EXE.B4DA06B11F6FC8D0BA6DB6429826FF51.4F5
;-- x86.00401090:
0x0000a6ac       fb031f6b        negs w27, wzr
0x0000a6b0       02008052        movz w2, 0
0x0000a6b4       9d4740b8        ldr w29, [x28], 4
0x0000a6b8       894740b8        ldr w9, [x28], 4
0x0000a6bc       2e257db3        bfi x14, x9, 3, 0xa
0x0000a6c0       cf414029        ldp w15, w16, [x14]
0x0000a6c4       ef01094b        sub w15, w15, w9
0x0000a6c8       4f000035        cbnz w15, 0xa6d0
0x0000a6cc       00021fd6        br x16
0x0000a6d0       e6fdff97        bl 0x9e68                          ;[1
0x0000a6d4       20021fd6        br x17
```

Example of code coverage measure

Uses BRANCH.EXE.*.*.mp.1.jc of **sample program** for the demonstration

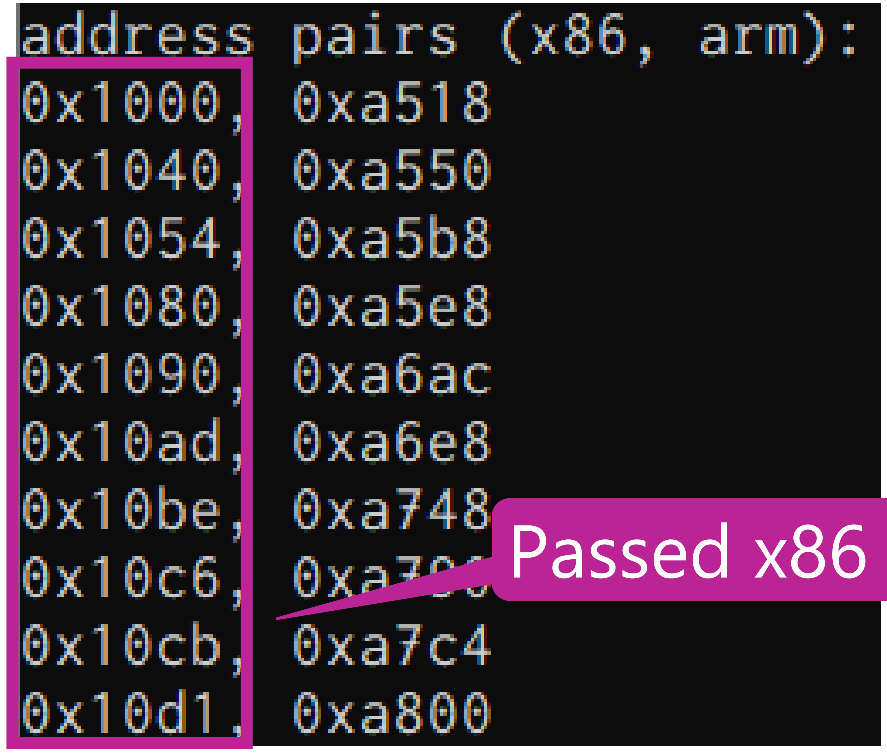
0x401080

0x401090



```
                main                                                XRE
00401080    55                PUSH        EBP
00401081    8b ec             MOV         EBP,ESP
00401083    8b 45 08          MOV         EAX,dword ptr [EBP + argc]
00401086    83 f8 01          CMP         EAX,0x1
00401089    75 09             JNZ         LAB_00401094
0040108b    e8 b0 ff ff ff    CALL        func0
00401090    33 c0             XOR         EAX,EAX
00401092    5d                POP         EBP
00401093    c3                RET

                LAB_00401094                                        XRE
00401094    83 f8 02          CMP         EAX,0x2
00401097    75 0b             JNZ         LAB_004010a4
00401099    8b c8             MOV         ECX,EAX
0040109b    e8 d0 ff ff ff    CALL        func2
004010a0    33 c0             XOR         EAX,EAX
004010a2    5d                POP         EBP
004010a3    c3                RET

                LAB_004010a4                                        XRE
004010a4    e8 b7 ff ff ff    CALL        func1
004010a9    33 c0             XOR         EAX,EAX
004010ab    5d                POP         EBP
004010ac    c3                RET
```

Address pairs (RVA to image base)



```
address pairs (x86, arm):
0x1000, 0xa518
0x1040, 0xa550
0x1054, 0xa5b8
0x1080, 0xa5e8
0x1090, 0xa6ac
0x10ad, 0xa6e8
0x10be, 0xa748
0x10c6, 0xa7
0x10cb, 0xa7c4
0x10d1, 0xa800
```

Passed x86 RVAs

Notes about code coverage measurement

Function coverage can be obtained, but branch coverage can be partially obtained

- because some branch instructions, such as jmp, do not end translation unit in some case (like previous example)

This method has non-invasive feature

- Binary instrumentation is not needed