

205 - Hide and Seek

Team Information

Team Name : HSPACE

Team Member : Jinung Lee, Beomjun Park, DoHyeon Kim, Soyoung Cho

Email Address : hspacedigitalforensicslab@gmail.com

Teams must:

- Provide a detailed, step-by-step description of their problem-solving approach to ensure reproducibility by another examiner.
- List all tools used to arrive at their conclusions.

Tools used:

Name:	Hxd	Publisher:	Maël Hörz
Version:	2.5.0.0		
URL:	https://mh-nexus.de/		

Name:	Python	Publisher:	Python Software Foundation
Version:	3.12.9		
URL:	https://www.python.org/		

Name:	xfs_repair	Publisher:	XFS Development Team
Version:	6.6.0		
URL:	https://xfs.wiki.kernel.org/		

Step-by-step methodology:

문제 풀이에 앞서, dfchallenge.org에 공지된 문제 해시와 다운로드 받은 문제 해시를 비교함으로써 분석 대상이 동일한 파일임을 증명한다.

Hash Value (MD5)

- disk.img : c3ce3ed46fa87aa07c754ecc11037116

[그림 1] dfchallenge.org에 공지된 문제 해시(MD5) 값.

disk.img 속성	
일반	디지털 서명
이름	해시 값
CRC32	65507D58
MD5	C3CE3ED46FA87AA07C754ECC11037116
SHA-1	3A42F548B587EC14FD12B0893B08A0ECCC4AA3...

[그림 2] HashTab을 통해 확인한 문제 해시(MD5) 값.

1. 첫번째 은닉된 데이터를 복구한 파일의 SHA-256 해시 값은 무엇인가?

제공된 disk.img를 xfs_repair을 통해 분석 해본 결과, [그림 3]과 같이 1번 슈퍼블럭에서 사용되지 않는 부분이 0이 아닌 데이터로 채워져 있다는 것을 알 수 있다. XFS 슈퍼블럭은 파일시스템의 메타데이터를 저장하는 핵심 데이터 구조이다. 슈퍼블럭은 파일시스템의 전체적인 구성 정보와 상태를 기록하며, 시스템 부팅 시 파일시스템을 인식하고 마운트하는 데 필요한 필수 정보를 제공한다.

```
diwls@WIN-I2GI44N112:/mnt/d/DFC2025/205/disk.img$ xfs_repair -f -n disk.img
Cannot get host filesystem geometry.
Repair may fail if there is a sector size mismatch between
the image and the host filesystem.
Phase 1 - find and verify superblock...
Cannot get host filesystem geometry.
Repair may fail if there is a sector size mismatch between
the image and the host filesystem.
Phase 2 - using internal log
    - zero log...
    - scan filesystem freespace and inode maps
would zero unused portion of primary superblock (AG #0)
    - found root inode chunk
Phase 3 - for each AG...
    - scan (but don't clear) agi unlinked lists...
    - process known inodes and perform inode discovery...
    - agno = 0
    - agno = 1
    - agno = 2
    - agno = 3
    - process newly discovered inodes...
Phase 4 - check for duplicate blocks...
    - setting up duplicate extent list...
    - check for inodes claiming duplicate blocks...
    - agno = 0
    - agno = 2
    - agno = 3
    - agno = 1
No modify flag set, skipping phase 5
Phase 6 - check inode connectivity...
    - traversing filesystem ...
    - traversal finished ...
    - moving disconnected inodes to lost+found ...
Phase 7 - verify link counts...
No modify flag set, skipping filesystem flush and exiting.
```

[그림 3] xfs_repair을 통해 disk.img를 분석.

XFS 슈퍼블럭의 내부 구조는 [그림 4]와 같이 구성되어 있다. 각 필드는 고정된 오프셋 위치에 저장되며, 0x00부터 시작하는 매직 넘버(0x58465342, "XFSB")를 통해 XFS 파일시스템임을 식별할 수 있다. 주요 메타데이터 정보들이 체계적으로 배치되어 있으며, 실제 사용되는 영역 이후의 공간이 슬랙 영역으로 활용 가능하다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F											
0x00	Magic Number (0x58465342)				Block_Size				Block Count																		
0x10	Blocks(used Real-time device)								Extents(used Real-time device)																		
0x20	UUID																										
0x30	Journaling Log start block(offset)								Root Inode #																		
0x40	Bitmap Inode								Real-time Bitmap Inode																		
0x50	Extent size(in Real-time device)				AG Block Size				AG Count			Bitmap Block cnt(in real-time device)															
0x60	Log Block cnt		Ver. #		Sector Size		Inode Size		Inode Per Block		File System Name[12]																
0x70	File System Name[12]								Block Log	Sector Log	Inode Size Log	Inopblk Log	AGblk Log	Rextents Log	In Progress	Imaxpct											
0x80	Inode Count								Free Inode Count																		
0x90	Free Block Count								Free Extent Count																		
0xA0	uquotino	gquotino		qflags				flag	share vn		inoalignment																
0xB0	unit	width		Dirblk log	Logsect log	Logsec size			Log sunit																		
0xC0	features2	bad features2		features compat				features ro compat																			
0xD0	features log incompat	crc		spino align				pquotino																			
0xE0	pquotino	lsn		meta uuid																							
0xF0	meta uuid								rrmapino																		

[그림 4] XFS의 슈퍼블럭 구조 정리.

HxD를 이용하여 AG #0(섹터 0)을 확인한다면, [그림 5]와 같이 확인할 수 있다. 하지만 이는 논문에서 언급한것과 조금 다르게. 논문에서는 해당 정상 영역을 272바이트를 사용한다고 하였지만, 해당 문제에서는 256바이트가 정상 영역으로써 사용되고 있었다.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	58	46	53	42	00	00	10	00	00	00	00	00	08	00	00	XFSB.....	
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000020	21	94	F5	C5	F7	79	42	F7	AC	58	47	3F	66	AE	EB	57 !“ôÀ-yB-~XG?f@ëW	
00000030	00	00	00	00	00	04	00	07	00	00	00	00	00	00	00	80€	
00000040	00	00	00	00	00	00	00	00	81	00	00	00	00	00	00	82	
00000050	00	00	00	01	00	02	00	00	00	00	04	00	00	00	00	00	
00000060	00	00	40	00	B4	B5	02	00	02	00	00	08	00	00	00	..@.p.....	
00000070	00	00	00	00	00	00	00	00	00	00	0C	09	03	11	00	00	19
00000080	00	00	00	00	00	00	03	00	00	00	00	00	00	00	00	00	6C1
00000090	00	00	00	00	00	06	88	BD	00	00	00	00	00	00	00	00“.....
000000A0	FF	FF	YYYYYYYYYYYYYYYY														
000000B0	00	00	00	00	00	00	08	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	01	00	00	01	8A	00	00	01	8AŠ	
000000D0	00	00	00	00	00	00	00	0F	00	00	00	2B	00	00	00	00+....
000000E0	6A	2F	DF	78	00	00	04	FF	j/Bx...YYYYYYYY								
000000F0	00	00	00	01	00	00	02	50	00	00	00	00	00	00	00	00 ..P.....	
00000100	00	00	00	38	39	61	10	00	10	B3	0E	00	4B	8E	C3	...89a...*,KZÄ	
00000110	D2	E3	F0	3C	84	B6	E4	F1	87	B4	D7	B4	D0	E6	1E	ÖÀ<..Nöäñ#*`Bw.	
00000120	71	B4	F0	F6	FA	CC	DF	EE	C3	D9	EB	35	7D	BB	5A	92 q`ööäüisüÜë5»Z'	
00000130	C7	FF	FF	FF	68	AF	FF	FF	FF	00	00	00	21	F9	04 Çyyy..h'yy..ù.		
00000140	01	00	00	0E	02	C0	00	00	00	10	00	10	00	00	04	
00000150	41	D0	B5	49	6B	95	36	EB	6C	4A	FA	1B	95	30	24	13 ADµIk..6ééJú..0\$.	
00000160	4E	E4	41	10	67	43	26	AD	00	90	01	00	84	63	C9	04 NAA.gcü..„CÉ.	
00000170	B8	CE	C0	1B	19	0D	20	68	BD	5A	89	4F	C1	6C	32 ,få...h+z'YEÁl2		
00000180	49	88	E6	64	41	AD	52	49	83	EA	D4	5A	C5	6A	31	48 I^ædA.RIféÖZÅj1H	
00000190	47	04	00	3B	00	00	00	00	00	00	00	00	00	00	00	G.;.....	
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

[그림 5] HxD를 이용하여 확인한 AG #0 정리.

슬랙 공간 분석을 위해 256바이트 이후부터 404바이트까지의 영역을 16진수 덤프로 추출하여 파일 시그니처 분석을 수행하였다. 분석 결과 [그림 6]과 같이 259바이트 위치에서 '38 39 61' 시

퀀스를 발견하였으며, 이는 GIF89a 파일 포맷의 일부에 해당한다. 그러나 정상적인 GIF 헤더 시작 부분인 '47 49 46'(GIF) 시그니처가 누락되어 있어 헤더가 손상된 상태로 확인된다.

손상된 GIF 파일의 논리 화면 기술자 부분과 컬러 테이블 일부가 남아있어 원본 이미지의 크기가 16x16 픽셀임을 확인할 수 있다. 이러한 부분적 손상은 의도적인 헤더 조작을 통해 일반적인 파일 복구 도구나 이미지 뷰어에서 인식되지 않도록 하는 안티포렌식 기법으로 판단된다.

손상된 헤더 시그니처		GIF 크기 정보	
000000000	00 00 00 38 39 61	10 00 10 00	B3 0E 00 4B 8E C3 ...89a....'..KŽÄ
000000010	D2 E3 F0 3C 84 BE D6 E4 F1 87 B4 D7 B4 D0 E6 1E		Öäö<„%Öäñ†'×'Đæ.
000000020	71 B4 F0 F6 FA CC DF EE C3 D9 EB 35 7D BB 5A 92		q'8öúíßiÄÙë5}»Z'
000000030	C7 FF FF FF OF 68 AF FF FF FF 00 00 00 21 F9 04		Çÿy.ñ~ÿy...!ù.
000000040	01 00 00 0E 00 2C 00 00 00 00 10 00 10 00 00 04	,.....
000000050	41 D0 B5 49 6B 95 36 EB 6C 4A FA 1B 95 30 24 13		AÐµIk•6ëlJÚ..•0\$.
000000060	4E E4 41 10 67 43 26 AD 00 90 01 00 84 63 C9 04		NÄA.gC&.....,cÉ.
000000070	B8 CE C0 1B 19 0D 20 68 BD 5A A8 9F 45 C1 6C 32		,fÀ... hñz"ÝEÁ12
000000080	49 88 E6 64 41 AD 52 49 83 EA D4 5A C5 6A 31 48		I^ædA.RIfëÖZÅj1H
000000090	47 04 00 3B		G..;

[그림 6] 슬랙영역을 추출하여 나온 헤더가 손상된 GIF.

[그림 7]과 같이 헤더를 정상적으로 복구하여, facebook 로고의 이미지를 확인할 수 있다.

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text
000000000	47 49 46 38 39 61 10 00 10 00 B3 0E 00 4B 8E C3	GIF89a....'..KŽÄ
000000010	D2 E3 F0 3C 84 BE D6 E4 F1 87 B4 D7 B4 D0 E6 1E	Öäö<„%Öäñ†'×'Đæ.
000000020	71 B4 F0 F6 FA CC DF EE C3 D9 EB 35 7D BB 5A 92	q'8öúíßiÄÙë5}»Z'
000000030	C7 FF FF FF OF 68 AF FF FF FF 00 00 00 21 F9 04	Çÿy.ñ~ÿy...!ù.
000000040	01 00 00 0E 00 2C 00 00 00 00 10 00 10 00 00 04,.....
000000050	41 D0 B5 49 6B 95 36 EB 6C 4A FA 1B 95 30 24 13	AÐµIk•6ëlJÚ..•0\$.
000000060	4E E4 41 10 67 43 26 AD 00 90 01 00 84 63 C9 04	NÄA.gC&.....,cÉ.
000000070	B8 CE C0 1B 19 0D 20 68 BD 5A A8 9F 45 C1 6C 32	,fÀ... hñz"ÝEÁ12
000000080	49 88 E6 64 41 AD 52 49 83 EA D4 5A C5 6A 31 48	I^ædA.RIfëÖZÅj1H
000000090	47 04 00 3B	G..;



[그림 7] 복원된 헤더와 GIF 이미지.

복원된 파일을 HashTab을 통해 확인한다면, SHA-256의 해시는 [그림 8]과 같이 **764308A95688BE904762C4732C39863D3DE02DB4BB3537B4801C577D7CFDB0EF** 이다.



[그림 8] HashTab을 이용한 facebook 이미지의 해시.

답: **764308A95688BE904762C4732C39863D3DE02DB4BB3537B4801C577D7CFDB0EF**

2. 두번째 은닉된 데이터를 복구한 파일의 SHA-256 해시 값은 무엇인가?

Fergus Toolan의 Data hiding in the XFS file system 논문에서는 XFS에 대한 다양한 은닉 기법을 설명하고 있다. 해당 논문에는 5개의 은닉 기법을 설명하고 있다. 이 중 Inode slack space에서 유의미한 정보를 찾을 수 있었다.

Inode는 index node의 줄임말로, 각 파일과 디렉토리마다 하나씩 할당된다. 파일의 실제 데이터는 별도 공간에 저장되고, inode는 그 파일에 대한 정보만 담고 있다. 하지만 해당 논문에서는 "XFS inode 구조는 inode 코어(176바이트), 데이터 포크, 속성 포크의 세 영역으로 구성되며, 확장 속성이 없는 파일의 경우 inode 슬랙 공간을 데이터 은닉에 활용할 수 있다"고 설명한다.

[별첨 1]의 파이썬 스크립트를 이용하여 disk.img를 분석하면, [그림 9]와 같이 0x1F200~0x1F600의 3개의 inode에 첫번째와 같이 손상된 GIF 헤더 시그니처의 GIF파일을 확인할 수 있다.

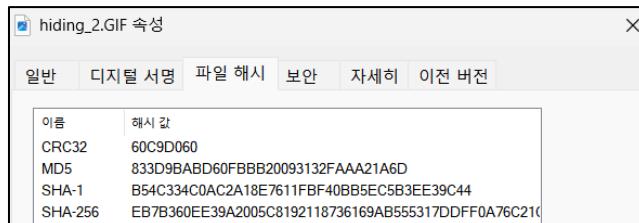
```
Inode at 0x0001F200:  
String: '='  
  000~~~{{zzzxxwwwvvvttrrrqqpppmmkkiiigge'  
00B0: 00 00 00 00 00 00 00 00 00 09 3D 20 00 0B  
0150: 00 00 00 38 39 61 0D 00 0D 00 D5 00 00 00 00  
0160: FF FF F8 F8 F7 F7 EA EA EA E2 E2 E1  
0170: E1 E1 E0 E0 DF DF DB DB DB D9 D9 D0 D0  
0180: D0 C7 C7 C6 C6 C2 C2 C0 C0 C0 BD BD BD  
0190: BC BC BC AD AD AB AB AA AA AA A5 A5 A4  
01A0: A4 A4 A2 A2 A0 A0 9F 9F 9F 9C 9C 9C 9B 9B  
01B0: 9B 99 99 99 97 97 97 93 93 93 91 91 8E 8E 8E  
01C0: 8D 8D 8C 8C 8C 8A 8A 8A 89 89 86 86 86 83  
01D0: 83 83 7F 7F 7E 7E 7B 7B 7B 7A 7A 7A 78 78  
01E0: 78 77 77 77 76 76 74 74 74 72 72 72 71 71 71  
01F0: 70 70 70 6D 6D 6B 6B 6B 69 69 69 67 67 67 65  
  
Inode at 0x0001F400:  
String: '>'  
  -@| )~FBTWP  
  eecccaaa``ZZZYXXQQQ!J` ? ,  
r^5) 760 (167,0:8:0:=9'@-@|.%%1/`@/`-1&'32`223`54`40`  
00B0: 00 00 00 00 00 00 00 00 00 00 09 3E 80 00 1B  
0150: 65 65 63 63 61 61 61 60 60 5F 5F 5F 5A 5A  
0160: 5A 59 59 59 58 58 51 51 51 FF FF FF 21 F9 04  
0170: 01 00 00 3F 00 2C 00 00 00 00 0D 00 0D 00 00 06  
0180: 90 40 05 09 F4 29 7E 46 A3 0F 88 A4 08 99 42 A2  
0190: A8 54 E4 0C 8D 50 A7 0A E9 74 F2 54 3C 27 D4 48  
01A0: 94 92 04 2C 28 4B 20 20 D8 A4 44 24 CC FA A2 4A  
01B0: 14 54 84 C4 8A 54 72 B5 02 19 2D 02 0E 2D 0E 05  
01C0: 2E 25 26 31 2F 01 1A 8C 0F 2F 0F 06 31 26 27 33  
01D0: 32 01 1C 32 07 0C 32 0B 07 33 27 29 35 34 01 1D  
01E0: 34 10 03 0D 01 13 35 29 2C 37 36 11 28 36 B2 08  
01F0: 11 36 37 2C 30 3A 38 BE BF BF 3A 30 3B 3D 39 C6  
  
Inode at 0x0001F600:  
String: 'A '  
  =;@>;>;>@A ;'  
00B0: 00 00 00 00 00 00 00 00 00 00 00 09 41 E0 00 0C  
01E0: C7 C7 3D 3B 14 3E 3C 3B CF 3B 3E 3E 3B 3C 3E 14  
01F0: 41 00 3B 00 00 00 00 00 00 00 00 00 00 00 00 00  
  =;@>;>;>@A ;'  
  GIF의 푸터 시그니처
```

[그림 9] 은닉되어 있는 헤더 시그니처가 손상된 GIF 파일.

[그림 10]과 같이 손상된 헤더 시그니처를 복구하고, 해당 사진을 확인해보면 “10”이 적힌 아이콘을 확인할 수 있다.

[그림 10] 복원된 헤더와 GIF 이미지.

복원된 파일을 HashTab을 통해 확인한다면, SHA-256의 해시는 [그림 11]과 같이 **EB7B360EE39A2005C8192118736169AB555317DDFF0A76C21CA330749959D762**이다.



[그림 11] HashTab을 이용한 “10” 이미지의 해시.

답: EB7B360EE39A2005C8192118736169AB555317DDFF0A76C21CA330749959D762

3. 세번째 은닉된 데이터의 SHA-256 해시 값은 무엇인가?

문제 2번에서 언급한 논문에서는 "XFS는 네 개의 타임스탬프(MACB)를 기록하며, 각각 4바이트 Unix 시간과 4바이트 나노초 값을 포함한다"며, "나노초 구성 요소를 덮어써서 정보를 숨길 수 있다"고 설명한다. 또한 "유효하지 않은 나노초 값을 피하기 위해 나노초 구성 요소의 최하위 3바이트만을 데이터 은닉에 활용한다"고 제안한다.

XFS 파일시스템에서 각 inode는 [그림 12]과 같이 Access, Modify, Change, Birth 시간의 4개 타임스탬프를 저장하며, 각 타임스탬프의 나노초 필드에서 3바이트씩 총 12바이트의 데이터를 은닉 할 수 있다. 이 방법은 파일 수정 시 타임스탬프가 갱신되어 은닉 데이터가 손실될 수 있는 안정성 문제가 있지만, 탐지하기 어려운 특징을 가진다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F							
0x00	Magic Number	mode		Ver	format	onlink		UID			GID												
0x10	nlink			Project ID		Padding						Flushiter											
0x20	atime						mtime						Size										
0x30	ctime						Extents Size						Extents Count										
0x40	Block Count						Flags			Generation						An Extent							
0x50	An Extent	Fork off	aformat	Dmdevmask	Dmstate	Flags			changecount							Fork off							
0x60	FFFF			crc		crc						flag2					aformat						
0x70	lsm						pad2						pad2				Dmstate						
0x80	cowextsize			pad2				ino						ino			Dmdevmask						
0x90	crtime						UUID						UUID				Dmstate						
0xA0	UUID						UUID						UUID										

[그림 12] inode의 구조와 각 시간의 위치 정보.

두번째 은닉된 데이터를 분석하며, [그림 13]과 같이 0x1BC00 inode에서, "Here it is!"라는 문자열을 확인할 수 있다. 이를 근거하여 해당 inode를 분석하였다.

```
Inode at 0x0001BC00:  
String: #  
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 08 5C 00 00 86 | ... #... | Here it is!  
01F0: 48 65 72 65 20 69 74 20 69 73 21 00 00 00 00 00 | Here it is!....|
```

[그림 13] 0x1BC00 inode에서 발견된 "Here it is!" 문자열."

0x1BC00으로 이동하면 [그림 14]와 같은 inode정보를 확인할 수 있다.

0001BC00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN.'.....è...è
0001BC10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BC20	36 23 2C C0 C8 E1 44 46 35 7F DD 90 53 99 43 20 6#,ÀÈáDF5.Ý.S™C
0001BC30	36 23 2C C0 C8 EC 32 30 00 00 00 00 00 00 08 5B 19 6#,ÀÈí20.....[.
0001BC40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001BC50	00 00 18 02 00 00 00 00 00 00 00 00 00 00 00 00
0001BC60	FF FF FF FF 3B EB 92 00 00 00 00 00 00 00 00 07 yyyy;ë'.....
0001BC70	00 00 00 01 00 00 00 16 00 00 00 00 00 00 00 18
0001BC80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001BC90	36 23 2C C0 C8 E1 32 35 00 00 00 00 00 00 00 00 DE 6#,ÀÈá25.....þ
0001BCA0	21 94 F5 C5 F7 79 42 F7 AC 58 47 3F 66 AE EB 57 !"ôÀ-yB÷¬XG?f®ëW
0001BCB0	00 00 00 00 00 00 00 00 00 00 00 08 5C 00 00 86\t

[그림 14] 0x1BC00 inode.

나노초 필드에서 3바이트씩 추출하여 재조합한 결과, [그림 15]과 같이 유의미한 정보를 획득할 수 없었다. 하지만 마지막 2바이트를 이용하여 타임스탬프 조작 여부를 확인하였을 땐, "DFC 2025" 문자열을 획득할 수 있었다.

마지막 3바이트로 확인	마지막 2바이트로 확인
<pre>0001BC00 49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN.'.....è...è 0001BC10 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01 0001BC20 36 23 2C C0 C8 E1 44 46 35 7F DD 90 53 99 43 20 6#,ÀÈáDF5.Ý.S™C 0001BC30 36 23 2C C0 C8 EC 32 30 00 00 00 00 00 00 08 5B 19 6#,ÀÈí20.....[. 0001BC40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</pre>	<p style="text-align: right;">áDF™C í20á25</p>
<pre>0001BC50 00 00 18 02 00 00 00 00 00 00 00 00 00 00 00 00</pre>	<p style="text-align: right;">DFC 2025</p>
<pre>0001BC60 FF FF FF FF 3B EB 92 00 00 00 00 00 00 00 00 07 yyyy;ë'.....</pre>	
<pre>0001BC70 00 00 00 01 00 00 00 16 00 00 00 00 00 00 00 18</pre>	
<pre>0001BC80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</pre>	
<pre>0001BC90 36 23 2C C0 C8 E1 32 35 00 00 00 00 00 00 00 00 DE 6#,ÀÈá25.....þ</pre>	
<pre>0001BCA0 21 94 F5 C5 F7 79 42 F7 AC 58 47 3F 66 AE EB 57 !"ôÀ-yB÷¬XG?f®ëW</pre>	
<pre>0001BCB0 00 00 00 00 00 00 00 00 00 00 00 08 5C 00 00 86\t</pre>	

[그림 15] 타임스탬프를 이용한 데이터 은닉 행위 확인.

분석 결과로 "Here it is!"와 "DFC 2025"를 획득할 수 있었다. "Here it is!"는 inode Slack Space를 이용한 데이터 은닉 행위로 확인할 수 있고, "DFC 2025"는 타임스탬프에서 나노초 변조를 이용한 데이터 은닉 행위를 확인할 수 있었다. 출제자의 의도로 판단한다면, 타임스탬프가 정답이지만, 두 행위 모두 데이터 은닉 행위라 판단하여 해당 행위의 답은 2개로 서술하였다.

문자열	SHA-256
Here it is!	f17924be06a611b77a2cc5a73269d849f70ad226720dec2e86cf79de95151549
DFC 2025	3e8421c920889bc0edeb84c9d05db485516e082e53f97461a2bf1358aab93556

[표 1] 세번째 은닉된 데이터의 SHA-256 해시 값(정답).

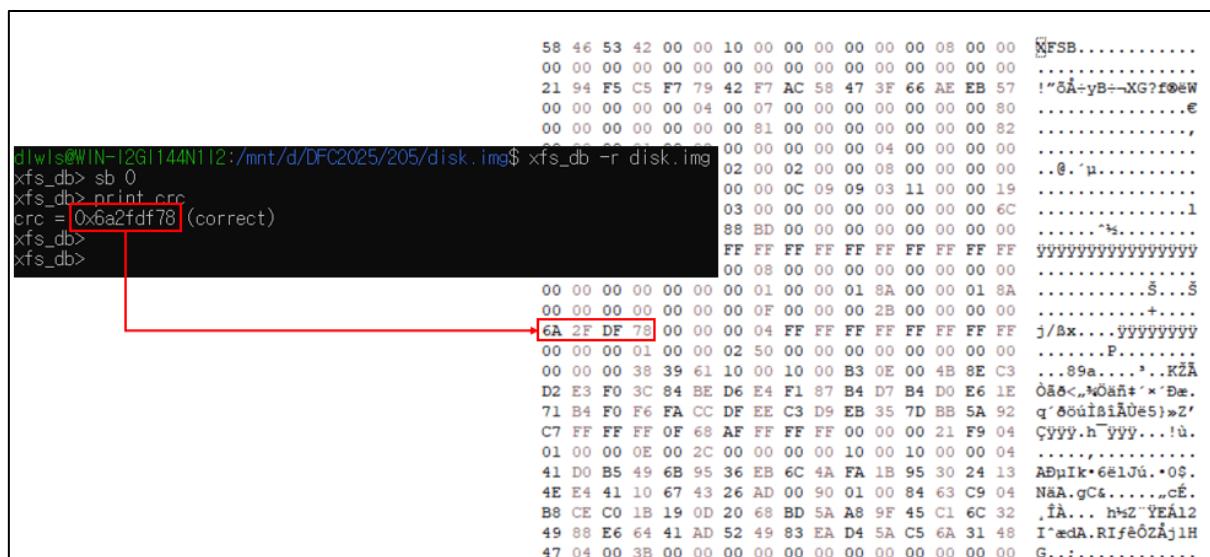
4. 은닉된 데이터를 모두 제거하고 은닉 이전의 디스크 이미지를 만드는 방법을 서술하시오.

XFS 파일시스템에서 발견된 세 가지 유형의 데이터 은닉 기법에 대한 완전한 제거 방법은 각 기법의 특성에 따라 다르게 접근해야 한다. 문제 2 번에서 언급한 논문에서 제시된 슈퍼블럭 슬랙 공간, inode 슬랙 공간, 나노초 타임스탬프 조작 기법을 순차적으로 정리하여 원본 상태로 복원하는 방법을 다음과 같이 제시한다.

4.1 슈퍼블럭 슬랙 공간 정리

첫 번째로 발견된 슈퍼블럭 슬랙 공간의 은닉 데이터 제거는 상대적으로 단순하다. XFS 슈퍼블럭의 실제 구조체 크기는 272 바이트이므로, 각 할당 그룹의 첫 번째 섹터에서 272 바이트 이후부터 섹터 끝까지의 공간을 0으로 초기화하면 된다.

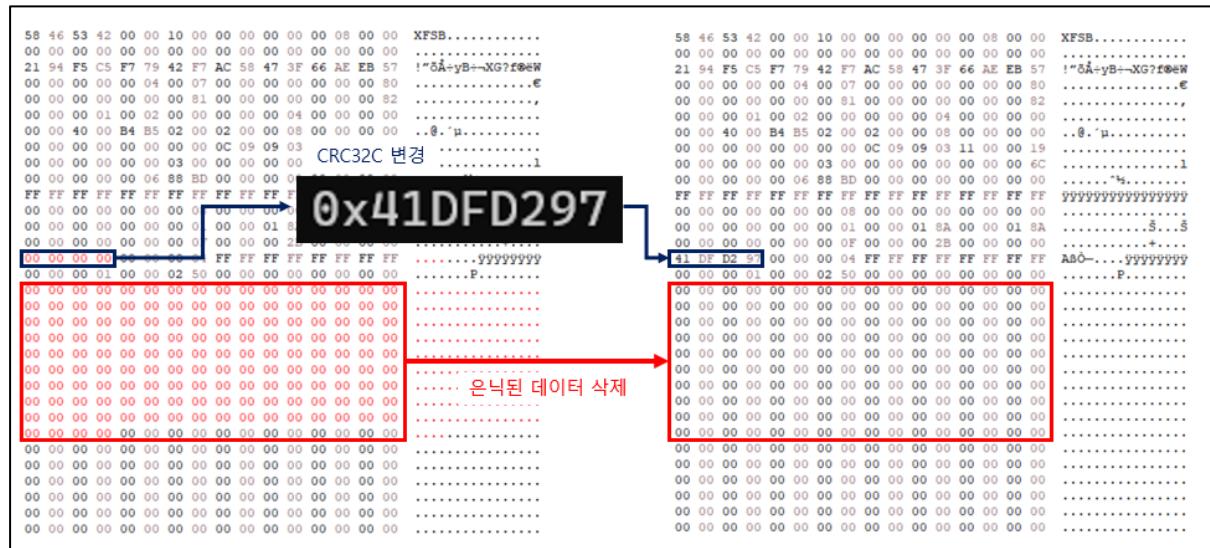
CRC32 를 확인하기 위하여 [그림 16]과 같이 xfs_db 를 이용하여 해당 정보를 확인했다. 이를 통해 AG #0 의 superblock 는 0x6a2fdf78 이라는 것을 알 수 있다.



```
58 46 53 42 00 00 10 00 00 00 00 00 08 00 00 XFSB.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
21 94 F5 C5 F7 79 42 F7 AC 58 47 3F 66 AE EB 57 !*óÁ-yB+-XG?@éW
00 00 00 00 00 04 00 07 00 00 00 00 00 00 00 80 .....
00 00 00 00 00 00 81 00 00 00 00 00 00 00 00 82 .....
00 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 .....
02 00 02 00 00 08 00 00 00 00 00 ..@.µ.....
00 00 0C 09 09 03 11 00 00 19 .....
03 00 00 00 00 00 00 00 00 00 6C .....1
88 BD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FF FFFFFFFF
00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 01 00 00 01 8A 00 00 01 8A .....š...
00 00 00 00 00 00 00 00 0F 00 00 00 2B 00 00 00 00 .....+....
6A 2F DF 78 00 00 00 04 FF FF FF FF FF FF FF FF FF j/Bx...ÿÿÿÿÿÿ
00 00 00 01 00 00 02 50 00 00 00 00 00 00 00 00 .....P...
00 00 00 38 39 61 10 00 10 00 B3 0E 00 4B 8E C3 ...89a...³..KŽÄ
D2 E3 F0 3C 84 BE D6 E4 F1 87 B4 D7 B4 D0 E6 1E Òåð<„¾äñ†×'Ðæ.
71 B4 F0 F6 FA CC DF EE C3 D9 EB 35 7D BB 5A 92 q' ðöúíðiÅüé5)»Z'
C7 FF FF OF 68 AF FF FF FF 00 00 00 21 F9 04 Çÿÿ.h~ÿÿ...!ù.
01 00 00 0E 00 2C 00 00 00 10 00 10 00 00 04 .....,
41 D0 B5 49 6B 95 36 EB 6C 4A FA 1B 95 30 24 13 ABþIk•6ëlJú..•$.
4E E4 41 10 67 43 26 AD 00 90 01 00 84 63 C9 04 Nää.gC&.....cÉ.
B8 CE C0 1B 19 0D 20 68 BD 5A A8 9F 45 C1 6C 32 ,Í... h4z"ÝÅ12
49 88 E6 64 41 AD 52 49 83 EA D4 5A C5 6A 31 48 I^ædA.RIféÖZÅj1H
47 04 00 3B 00 00 00 00 00 00 00 00 00 00 00 00 G;.....
```

[그림 16] xfs_db를 이용한 데이터가 은닉된 superblock의 CRC32 정보 확인.

[그림 17]과 같이, CRC32C 값을 획득하기 위하여 은닉된 데이터와 CRC32C 값을 모두 0 으로 변경한다. 이 후 해당 superblock 을 새로운 바이너리로 만들었다. 이를 [별첨 2]의 파이썬 스크립트를 이용하여 CRC32C 해시를 확인하고 이를 disk.img 에 삽입하여 superblock 에 은닉된 데이터와 흔적을 제거할 수 있다.



[그림 17] CRC32 및 은닉된 데이터 삭제 과정.

4.2 Inode 슬랙 공간 정리

Inode 슬랙 공간의 정리는 각 inode 를 개별적으로 검사하여 처리해야 한다. 해당 논문에서 설명하는 바와 같이 inode 슬랙 공간의 용량은 $512 - (176 + (16 \times x))$ 공식으로 계산되며, 연속적인 파일의 경우 각 inode 당 320 바이트를 사용할 수 있다.

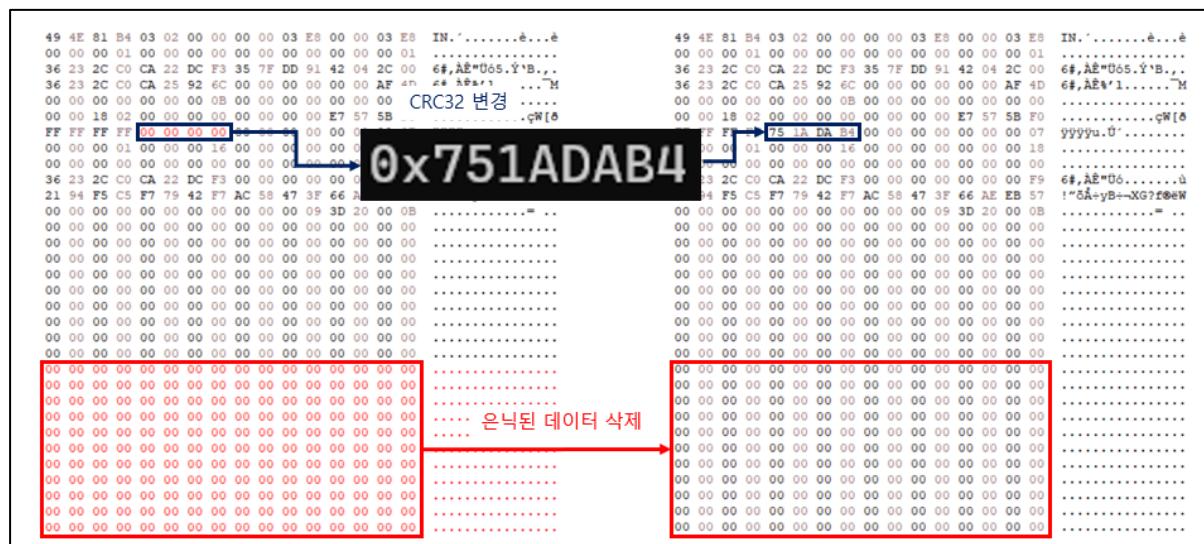
정리 과정은 다음과 같다. 먼저 모든 할당된 inode 를 순회하면서 'IN' 시그니처를 확인한다. 각 inode 에서 176 바이트의 코어 영역과 실제 사용 중인 데이터 포크 크기를 계산하여 사용되지 않는 슬랙 공간 부분을 0 으로 초기화한다.

[그림 12]에서의 inode 구조체에서 볼 수 있듯, inode 의 CRC32C 해시는, 0x65~0x68 까지의 4 바이트이다. 이를 [그림 18]에서 확인하면 F1 01 99 57 의 4 바이트를 확인할 수 있다.

0001F200	49 4E 81 B4 03 02 00 00 00 00 03 E8 00 00 03 E8	IN. '.....è...è
0001F210	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001F220	36 23 2C C0 CA 22 DC F3 35 7F DD 91 42 04 2C 00	6#,ÀÈ"Ü65.Ý'B.,.
0001F230	36 23 2C C0 CA 25 92 6C 00 00 00 00 00 00 AF 4D	6#,ÀÈ%'1.....~M
0001F240	00 00 00 00 00 00 00 0B 00 00 00 00 00 00 00 00
0001F250	00 00 18 02 00 00 00 00 00 00 00 00 E7 57 5B F0çW[δ
0001F260	FF FF FF F1 01 99 57 00 00 00 00 00 00 00 00 07	YYYYY[.DEW.....
0001F270	00 00 00 01 00 00 00 16 00 00 00 00 00 00 00 18
0001F280	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001F290	36 23 2C C0 CA 22 DC F3 00 00 00 00 00 00 F9	6#,ÀÈ"Üó.....ù
0001F2A0	21 94 F5 C5 F7 79 42 F7 AC 58 47 3F 66 AE EB 57	!"ÖÄ-yB-XG?fØëW
0001F2B0	00 00 00 00 00 00 00 00 00 00 00 09 3D 20 00 0B= ..

[그림 18] inode에서 CRC32해시 값의 위치.

[그림 19]와 같이, CRC32 값을 획득하기 위하여 은닉된 데이터와 CRC32C 값을 모두 0 으로 변경한다. 이 후 해당 inode 를 새로운 바이너리로 만들어 CRC32C 해시를 확인하고 이를 disk.img 에 삽입하여 inode 에 은닉된 데이터와 흔적을 제거할 수 있다.



[그림 19] CRC32C 및 은닉된 데이터 삭제 과정.

이러한 방법을 이용하여, 0x1F200~0x1F600의 총 3개의 inode에 대해 동일한 과정으로 은닉한 데이터를 제거해야한다. 이를 마친 결과는 [표 2]와 같이 정리할 수 있다.

오프셋	은닉된 데이터가 제거되지 않은 CRC32C	은닉된 데이터가 제거된 CRC32C
0x1F200	F1 01 99 57	DA 37 FA 94
0x1F400	D3 15 96 92	BA 85 B3 7E
0x1F600	78 AF 59 45	CA 28 04 D7

[표 2] 데이터가 은닉된 inode에 대하여, 데이터 및 흔적 삭제 후 CRC32 해시 정리.

4.3 나노초 타임스탬프 정리

타임스탬프 조작을 통한 은닉 데이터 제거는 가장 신중하게 접근해야 한다. 논문에서는 "나노초 구성 요소의 최하위 3바이트만을 데이터 은닉에 활용한다"고 설명한다. 하지만 해당 문제에서는 최하위 2바이트만을 데이터 은닉에 활용하고 있기 때문에, 이 부분을 정상적인 나노초 값으로 복원해야 한다.

앞서 확인한 은닉된 데이터의 시간 정보를 실제 수정 시간 패턴을 고려하여 자연스러운 나노초 값으로 설정해야 한다. 이를 위하여 [별첨 2]의 파이썬 스크립트를 이용하여 inode의 시간 정보를 획득한 후에 분석을 진행하였다.

[그림 20]은 해당 파이썬 스크립트를 이용해 시간 정보를 추출한 결과 중 일부이다. 0x1BC00은 "Here it is!" 문자열이 존재한 inode의 주소이며, 이를 중심으로 시간을 비교하였을 때, 주변 파일과는 근소한 차이라는 것을 알 수 있다.

```
0x0001B600, 1998-10-13 19:34:40.3368971691, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3369527360, 1998-10-13 19:34:40.3368971691  
0x0001B800, 1998-10-13 19:34:40.3369527360, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3369853711, 1998-10-13 19:34:40.3369527360  
0x0001BA00, 1998-10-13 19:34:40.3369853711, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3370244933, 1998-10-13 19:34:40.3369853711  
0x0001BC00, 1998-10-13 19:34:40.3370266278, 1998-06-11 22:37:20.1402553120, 1998-10-13 19:34:40.3370922544, 1998-10-13 19:34:40.3370201653  
0x0001BE00, 1998-10-13 19:34:40.3370929176, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3371221446, 1998-10-13 19:34:40.3370929176  
0x0001C000, 1998-10-13 19:34:40.3371221446, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3371773198, 1998-10-13 19:34:40.3371221446  
0x0001C200, 1998-10-13 19:34:40.3371773198, 1998-06-11 22:37:20.1402536960, 1998-10-13 19:34:40.3372537268, 1998-10-13 19:34:40.3371773198
```

[그림 20] inode의 시간 정보를 파싱한 정보 중 일부.

또한 이를 해석하면 [표 3]과 같이 나노초가 1초를 초과해, 비정상적인 데이터라는 것을 확인할 수 있다.

시간 정보	바이너리 정보	나노초 변환	비고
atime	36 23 2C C0 C8 E1 44 46	3,370,132,550	1초 초과 (비정상)
mtime	35 7F DD 90 53 99 43 20	1,402,643,232	1초 초과 (비정상)
ctime	36 23 2C C0 C8 EC 32 30	3,370,840,624	1초 초과 (비정상)
crttime	36 23 2C C0 C8 E1 32 35	3,370,128,949	1초 초과 (비정상)

[표 3] 0x1BC00주소의 inode 시간 정보 분석.

4.3.1 Access Time

Access Time의 경우, [그림 21]과 같이, 바로 앞 inode인 0x1BA00과 최하위 2바이트만 다른 것이 확인되어, 해당 시간 정보는 해당 파일과 동일하다고 추측하였다.

0001BA00	49 4E 81 B4 03 02 00 00 00 00 03 E8 00 00 03 E8	IN.‘.....è....è
0001BA10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BA20	36 23 2C C0 C8 DB E3 OF 35 7F DD 90 53 99 04 00	6#,ÀÈÛã.5.Ý.S™..
0001BA30	36 23 2C C0 C8 E1 DB 45 00 00 00 00 00 04 E2 1B	6#,ÀÈáÛE.....â..
0001BA40	00 00 00 00 00 00 00 00 4F 00 00 00 00 00 00 00O.....
0001BC00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8	IN.‘.....è....è
0001BC10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BC20	36 23 2C C0 C8 E1 44 46 35 7F DD 90 53 99 43 20	6#,ÀÈáDF5.Ý.S™C
0001BC30	36 23 2C C0 C8 EC 32 30 00 00 00 00 00 08 5B 19	6#,ÀÈì20.....[..
0001BC40	00 00 00 00 00 00 00 00 86 00 00 00 00 00 00 00t.....

[그림 21] Access Time의 시간 정보 변경 사항.

4.3.2 Modified Time

Modified Time의 경우, [그림 22]과 같이, 바로 앞 inode인 0x1BA00과 최하위 2바이트만 다른 것이 확인되어, 해당 시간 정보는 해당 파일과 동일하다고 추측하였다. 이러한 이유로 최하위 2바이트를 교체하였다.

0001BA00	49 4E 81 B4 03 02 00 00 00 00 03 E8 00 00 03 E8	IN.‘.....è....è
0001BA10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BA20	36 23 2C C0 C8 DB E3 OF 35 7F DD 90 53 99 04 00	6#,ÀÈÛã.5.Ý.S™..
0001BA30	36 23 2C C0 C8 E1 DB 45 00 00 00 00 00 04 E2 1B	6#,ÀÈáÛE.....â..
0001BC00	49 4E 81 B4 03 02 00 00 00 00 03 E8 00 00 03 E8	IN.‘.....è....è
0001BC10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BC20	36 23 2C C0 C8 E1 DB 45 35 7F DD 90 53 99 43 20	6#,ÀÈáÛE5.Ý.S™C
0001BC30	36 23 2C C0 C8 EC 32 30 00 00 00 00 00 08 5B 19	6#,ÀÈì20.....[..
0001BC40	00 00 00 00 00 00 00 00 86 00 00 00 00 00 00 00t.....

[그림 22] Modified Time의 시간 정보 변경 사항.

4.3.3 Changed Time

Changed Time의 경우, [그림 23]과 같이, 바로 뒤 inode인 0x1BE00과 최하위 2바이트만 다른 것이 확인되어, 해당 시간 정보는 해당 파일과 동일하다고 추측하였다. 이러한 이유로 최하위 2바이트를 교체하였다.

0001BE00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN. è.... è
0001BE10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BE20	36 23 2C C0 C8 EC 4C 18 35 7F DD 90 53 99 04 00 6#,ÀÈíL.5.Ý.S™..
0001BE30	36 23 2C C0 C8 F0 C1 C6 00 00 00 00 00 02 99 1B 6#,ÀÈøÁE.....™..
0001BE40	00 00 00 00 00 00 00 2A 00 00 00 00 00 00 00 00
0001BC00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN. è.... è
0001BC10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BC20	36 23 2C C0 C8 E1 DB 45 35 7F DD 90 53 99 04 00 6#,ÀÈáÜE5.Ý.S™..
0001BC30	36 23 2C C0 C8 EC 32 30 00 00 00 00 00 08 5B 19 6#,ÀÈí20.....[.
0001BC40	00 00 00 00 00 00 00 86 00 00 00 00 00 00 00 00

[그림 23] Changed Time의 시간 정보 변경 사항.

4.3.4 Changed Time

Changed Time의 경우, [그림 24]과 같이, 바로 뒤 inode인 0x1BE00과 최하위 2바이트만 다른 것이 확인되어, 해당 시간 정보는 해당 파일과 동일하다고 추측하였다. 이러한 이유로 최하위 2바이트를 교체하였다.

0001BC00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN. è.... è
0001BC10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BC20	36 23 2C C0 C8 E1 DB 45 35 7F DD 90 53 99 04 00 6#,ÀÈáÜE5.Ý.S™..
0001BC30	36 23 2C C0 C8 EC 4C 18 00 00 00 00 00 08 5B 19 6#,ÀÈíL.....[.
0001BC40	00 00 00 00 00 00 00 86 00 00 00 00 00 00 00 00
0001BC50	00 00 18 02 00 00 00 00 00 00 00 00 CD C6 8D 31,ÍÆ.1
0001BC60	FF FF FF FF 3B EB 92 00 00 00 00 00 00 00 00 07 -yyyyy;è'.....
0001BC70	00 00 00 01 00 00 00 16 00 00 00 00 00 00 00 18
0001BC80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001BC90	36 23 2C C0 C8 E1 32 35 00 00 00 00 00 00 00 00 DE 6#,ÀÈá25.....Þ
0001BCA0	21 94 F5 C5 F7 79 42 F7 AC 58 47 3F 66 AE EB 57 !"ôÄ÷yB÷→XG?f@ew
0001BA00	49 4E 81 B4 03 02 00 00 00 00 00 03 E8 00 00 03 E8 IN. è.... è
0001BA10	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01
0001BA20	36 23 2C C0 C8 DB E3 0F 35 7F DD 90 53 99 04 00 6#,ÀÈÛã.5.Ý.S™..
0001BA30	36 23 2C C0 C8 E1 DB 45 00 00 00 00 00 04 E2 1B 6#,ÀÈáÜE.....â..
0001BA40	00 00 00 00 00 00 00 4F 00 00 00 00 00 00 00 00

[그림 24] Changed Time의 시간 정보 변경 사항.

4.3.5 나노초 타임스탬프 마무리

[그림 25]와 같이, CRC32 값을 획득하기 위하여 은닉된 데이터와 CRC32 값을 모두 0 으로 변경한다. 이 후 해당 inode 를 새로운 바이너리로 만들어 CRC32C 해시를 확인하고 이를 disk.img 에 삽입하여 TimeStamp 에 은닉된 데이터와 흔적을 제거할 수 있다.

[그림 25] CRC32C 및 은닉된 데이터 삭제 과정.

4.4 결론

모든 은닉된 데이터를 삭제하고, CRC32C를 이용하여 파일 시스템의 무결성을 검증한다면, [그림 26]과 같은 해시 정보를 획득할 수 있고, 이를 정리하면 [표 4]와 같다.



[그림 26] 은닉된 데이터가 사라진 disk.img 파일 해시 정보.

이름	값
MD5	B79FF8B1A233EB2406AA4508D765C961
SHA-1	472F2D8827B6E5934AB6DF9F62FD6B4FA3C0B29B
SHA-256	919AD1558EB1D376ADA9BAB9DA338C405A2B9151E5CABE9254EB4B199CC0053C

[표 4] 은닉된 데이터가 사라진 disk.img 파일 해시 정보 정리.

```

import struct
import re

class XFSInodeAnalyzer:
    def __init__(self):
        self.inode_size = 512
        self.inode_core_size = 176

    def parse_hex_input(self, hex_string):
        hex_clean = re.sub(r'[^0-9A-Fa-f]', "", hex_string)
        return bytes.fromhex(hex_clean)

    def analyze_inode(self, inode_data):
        if len(inode_data) < self.inode_size:
            return None

        magic = inode_data[0:2]
        if magic != b'IN':
            return None

        slack_data = inode_data[self.inode_core_size:]
        non_zero_count = len(slack_data) - slack_data.count(0)

        if non_zero_count == 0:
            return None

        strings = []
        try:
            ascii_str = slack_data.decode('ascii', errors='ignore').strip('\x00')
            if ascii_str and len(ascii_str.strip()) > 2:
                strings.append(ascii_str.strip())
        except:
            pass

        return {
            'slack_data': slack_data,
            'strings': strings
        }

```

```

def print_hex_dump(self, data, base_offset=176):
    for i in range(0, len(data), 16):
        chunk = data[i:i+16]
        if any(b != 0 for b in chunk):
            offset = base_offset + i
            hex_part = ' '.join(f'{b:02X}' for b in chunk)
            ascii_part = ''.join(chr(b) if 32 <= b <= 126 else '.' for b in chunk)
            print(f"  {offset:04X}: {hex_part:<47} |{ascii_part}|")

def scan_file(self, filepath):
    found_count = 0

    with open(filepath, 'rb') as f:
        f.seek(0, 2)
        file_size = f.tell()
        f.seek(0)

        for offset in range(0, file_size - self.inode_size, self.inode_size):
            f.seek(offset)
            data = f.read(self.inode_size)
            result = self.analyze_inode(data)
            if result:
                found_count += 1
                print(f"I-node at 0x{offset:08X}:")
                if result['strings']:
                    for s in result['strings']:
                        print(f"  String: '{s}'")

                self.print_hex_dump(result['slack_data'])
                print()

    print(f"Total found: {found_count}")

def main():
    import sys

    if len(sys.argv) != 2:
        sys.exit(1)

```

```
analyzer = XFSInodeAnalyzer()
analyzer.scan_file(sys.argv[1])

if __name__ == "__main__":
    main()
```

[별첨 1] inode slack space data parser script.

```
import sys

def crc32c(data):
    table = []
    for i in range(256):
        crc = i
        for _ in range(8):
            crc = (crc >> 1) ^ 0x82F63B78 if crc & 1 else crc >> 1
        table.append(crc)

    crc = 0xFFFFFFFF
    for byte in data:
        crc = (crc >> 8) ^ table[(crc ^ byte) & 0xFF]
    return crc ^ 0xFFFFFFFF

data = bytes.fromhex(''.join(sys.argv[1:]).replace(' ', ''))
crc = crc32c(data)
print(f"0x{((crc & 0xFF) << 24) | (((crc >> 8) & 0xFF) << 16) | (((crc >> 16) & 0xFF) << 8) | ((crc >> 24) & 0xFF):08X}")
```

[별첨 2] crc32c check script.