Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.26, NO.5, Oct. 2016 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2016.26.5.1141

오픈 소스 중복 제거 파일시스템에서의 완전 삭제*

조 현 웅,† 김 슬 기, 권 태 경‡
연세대학교 정보대학원 정보보호연구실

Sanitization of Open-Source Based Deduplicated Filesystem*

Hyeonwoong Cho,† SeulGi Kim, Taekyoung Kwon‡
Graduate School of Information, Yonsei University

요 약

중복 제거 파일시스템은 저장 공간 절약의 이점이 있지만, 기존 완전 삭제 도구를 이용하는 경우 여전히 지워진 블록이 복원될 우려가 있다. 본 논문에서는 FUSE(Filesystem in USErspace)를 이용하여 개발된 오픈 소스 중복 제거 파일시스템인 LessFS를 바탕으로 완전 삭제 기법을 연구하였다. 먼저 중복 제거 파일시스템에서 지워진 데이터 블록을 복구하는 취약점을 보였으며, 데이터 블록과 함께 fingerprint DB영역을 고려한 완전 삭제 기법을 제안하고 구현하였다. 성능 측정 결과 완전 삭제에 필요한 시간은 완전 삭제가 적용되지 않은 경우에 비해 60~70배 가량으로 나타났으며, 이러한 작업 수행시간의 증가는 chunk의 증가로 인한 fingerprint DB 접근에 따른 오버헤드가 큰 비중을 차지하는 것으로 나타났다. 또한 chunk 크기가 65,536바이트를 넘는 경우에는 기존 파일시스템의 완전 삭제 기법보다 더 좋은 완전 삭제 성능을 보였다.

ABSTRACT

Deduplicated filesystem can reduce usage of storage. However, it be able to recover deleted block. We studied sanitization of deduplicated filesystem, LessFS which is based on FUSE(Filesystem in USErspace). First, we show a vulnerability recover deleted data in the deduplicated filesystem. We implement sanitization of deduplicated filesystem considering the part of fingerprint DB with data blocks. It takes 60~70 times compared to without sanitization. Which means access time to fingerprint DB and overhead derived from increase of number of chunk have a critical impact on sanitization time. But in case of more than 65,536 Byte of chunksize, it is faster than normal filesystem without deduplication.

Keywords: deduplication, sanitization, FUSE, LessFS, filesystem

I. 서 론

LTE를 비롯한 고속 무선 데이터 전송 기술에 기반한 스마트 기기가 대중화되고, 사물인터넷(IoT)의 보급으로 인하여 필요한 저장 공간이 급격히 증가하고 있으나, 스토리지의 발전 속도는 둔화된 상황이다. 따라서 제한된 저장 공간 안에 많은 데이터를 저장하기 위하여 중복 제거 기술이 광범위하게 사용되고 있다. 최근 클라우드 및 IaaS(Infrastructure as a Service)와 같이 불특정 다수의 사용자가 스토리지를 공유하는 환경이 대중화되고 있는데, 이러한 환경에서 중복 제거가 특히 유용하다.

한편, 우리나라의 개인정보보호법 등 여러 법령과 보안규정에서 데이터의 복원이 불가능한 완전 삭제를 규정하고 있다. 하지만 중복 제거 파일시스템에서는

Received(07. 25. 2016), Modified(09. 20. 2016), Accepted(09. 20. 2016)
* 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No.NRF-2015R1A2A2A01004792). 또한, 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구 센터육성 지원사업의 연구결과로 수행되었음 (IITP-2016-H8501-16-1008)
† 주저자, emnuy@yonsei.ac.kr
‡ 교신저자, taekyoung@yonsei.ac.kr(Corresponding author)

하나의 데이터 블록이 여러 개체에 의하여 참조될 수 있기 때문에 기존의 완전 삭제 도구들을 이용할 수 없다. 이러한 이유로 중복 제거 스토리지에서의 완전 삭제와 관련한 연구는 거의 찾아볼 수 없는 실정이다.

본 논문에서는 FUSE(Filesystem in USErspace)를 이용하여 개발된 오픈 소스 중복 제거 파일시스템인 LessFS를 바탕으로 완전 삭제 기법을 연구하였다. 먼저 중복 제거 파일시스템에서 지워진 중복 데이터 블록을 중심으로 삭제 후 복구하는 취약점을 밝혔으며, 더 이상 사용되지 않는 데이터 블록을 덮어씌우고, fingerprint DB영역을 덮어씌워 중복 영역을 참조할 수 없도록 하는 완전 삭제 기법을 제안하고 구현 실험하였다. 성능 측정 결과 완전 삭제에 필요한 시간은 완전 삭제가 적용되지 않은 경우에 비해 60~70배 가량으로 나타났지만, chunk 크기가 65,536바이트를 넘는 경우에는 기존 파일시스템의 완전 삭제 기법보다 더 좋은 완전 삭제 성능을 보였다.

II. 이론적 배경 및 관련 연구

2.1 완전 삭제

2.1.1 완전 삭제의 의의

일반적인 파일 시스템에서 삭제 작업을 수행하게 되는 경우 파일 시스템은 삭제되는 파일의 할당 정보만을 제거하고 이후에 쓰기 작업이 수행되는 경우 해당 영역에 다시 쓸 수 있는 상태로 두게 된다. 따라서 실제 데이터가 저장되는 영역은 삭제 후에도 새로운 쓰기 작업으로 인해 덮어씌워지지 않는 이상 그대로 남아 있게 되고, 이를 이용하여 삭제된 파일의 내용을 복원할 수 있게 되는 것이다.

NIST는 복구 불가능하도록 데이터가 처음부터 존재하지 않았던 상태로 만들어 저장 매체로부터 삭제하고자 하는 부분을 완전히 제거하는 것으로 완전 삭제를 정의하였다[1]. 이러한 완전 삭제 작업은 삭제되는 파일이 저장되어 있던 데이터 영역을 식별할 수 없는 값으로 덮어씌우는 작업을 통해 이루어진다.

2.1.1 완전 삭제의 위협 모델

Wei외 등[2]은 완전 삭제되지 않은 데이터의 내용을 복원해낼 수 있는 위협 모델을 다음과 같은 3단계로 제시하였다.

1) Casual Attack
특별한 기술이 필요 없이 일반적인 파일 시스템을 통한 접근이다. 대표적인 예로 휴지통에서 삭제된 파일을 복원해내는 것을 들 수 있다.
2) Robust Keyboard Attack
일반적이지 않은 방법을 이용하여 파일시스템에 접근을 통한 공격이다. 허가되지 않은 루트 권한을 이용하여 데이터 영역에 접근하거나, 파일시스템 안의 비할당 영역에 접근하여 삭제된 파일의 잔존 데이터를 복원하는 것을 예로 들 수 있다.
3) Laboratory Attack
특별한 설비를 통해 데이터 영역에 접근하는 방법이다. 디스크가 덮어씌워졌음에도 불구하고 자성체 분석을 통해 미세한 변화가 남아있는 것을 감지하여 비트 단위로 삭제된 데이터를 복원할 수 있다.

2.1.2 완전 삭제의 요구 조건

Botelho 등은 완전 삭제가 요구하여야 하는 네 가지 조건을 다음과 같이 제시하였다[3].

P1) 삭제된 데이터는 완전히 제거되어야 한다.
P2) 삭제되지 않은 데이터는 정상적으로 사용 가능하여야 한다.
P3) 삭제 과정은 효율적이어야 한다.
P4) 삭제 과정 수행 중에도 파일시스템은 사용 가능하여야 한다.

2.2 중복 제거

Bhagwat 등은 파일을 스토리지에 저장할 때 일정한 크기의 블록으로 쪼개는 현대적인 의미의 중복 제거 방법론을 제시하였다[4]. 이렇게 쪼개진 부분을 기존에 저장된 스토리지의 내용과 비교하여 중복되지 않는 경우 해당 블록을 스토리지에 새로이 저장하게 되는데 이것을 chunk라고 한다. 또한 파일을 중복 제거 스토리지에 저장함에 있어서 중복을 비교하고 블록을 저장하는 이러한 과정을 chunking이라고 한다.

Henson은 chunk의 데이터를 이용하여 생성한 hash값을 바탕으로 효율적인 비교를 할 수 있는 방법론을 제시하였다[5]. 파일의 블록마다 사용되는

chunk가 존재하고, 이러한 chunk는 중복 참조될 수 있다. 또한 파일의 쪼개진 블록 순서대로 chunk의 실제 데이터가 저장된 주소를 참조하기 위한 list를 작성하게 된다. 이것을 hash index list또는 fingerprint라고 한다.

Harnik 등은 중복 제거와 압축의 공통점과 차이점을 연구하였다[6]. 중복 제거는 데이터 점유 공간을 줄인다는 점에서 넓은 의미의 압축이라고 볼 수 있지만, chunk간의 비교가 필요하다는 점에서 global한 속성을 가지는 반면, 압축은 해당 chunk만 압축하면 된다는 점에서 local한 속성을 가진다는 차이가 있다.

2.3 중복 제거 파일시스템에서의 완전 삭제

Ext, NTFS등 일반적인 파일시스템의 경우 완전 삭제를 할 수 있는 여러 가지 툴들이 이미 구현되어 있다. 그러나 중복 제거가 적용된 파일시스템에서는 제대로 동작하지 않는다.

중복 제거 파일시스템은 저장하고자 하는 파일의 블록이 기존에 저장된 chunk와 중복되는지 여부를 판단하여 중복되지 않는 경우는 새로이 저장하지만, 중복되는 경우 새로이 저장하지 않고 기존에 저장된 chunk를 참조하는 방식으로 작동하게 된다. 따라서 사용자가 파일을 삭제하여도 해당 chunk가 다른 파일에 의하여 사용되는 경우 완전 삭제 작업을 수행할 수 없다.

따라서 중복 제거 파일시스템에서의 파일의 완전 삭제를 위해서는 중복 제거 파일시스템의 구조를 파악하여 더 이상 사용되지 않는 chunk만을 가려내고 여기에 따른 작업을 수행하여야 하는데, 기존의 툴들은 단순히 해당 파일이 저장된 주소를 찾아 덮어쓰기 작업을 함에 그치고 있을 뿐이므로 중복 제거 파일시스템에 사용하는 경우 제대로 동작하지 않게 된다.

III. 중복 제거 파일시스템의 구조

3.1 LessFS 개요

대표적인 오픈 소스 중복 제거 파일시스템으로는 LessFS, SDFS, ZFS등이 있다[7]. 각각의 파일시스템들은 세부적인 기능에서 차이가 있으나, 핵심적인 중복 제거 기능은 유사한 구조를 가지고 있다. 여기서는 LessFS를 바탕으로 중복 제거 파일시스템의 구조를 파악하기로 한다.

LessFS는 Mark Ruijter가 개발하였으며, 최신 버전은 1.7.0이다. 주로 Fedora/RedHat기반의 Linux에서 잘 동작하도록 설계되어 있다.

chunk를 관리하는 DB를 선택할 수 있는데, 여기서는 default인 tokyocabinet을 사용하기로 한다. 또한 블록데이터 설정 역시 default인 file_io 방식을 사용하기로 한다.

3.2 LessFS의 작동 방식

LessFS 스토리지에 파일을 저장하는 경우 저장하기에 앞서 스토리지에 정해진 일정한 크기로 파일을 chunking하게 된다. 이렇게 나누어진 블록들마다 hash 값을 생성하여 fingerprint DB에 저장한다.

이렇게 생성된 hash 값을 chunk DB에 질의하여 동일한 hash 값이 chunk DB에 이미 존재하는 경우 새로이 저장하지 않고 inuse값만을 증가시킨다. 동일한 hash 값이 chunk DB에 존재하지 않는 경우 hash value를 새로운 key로, chunk가 저장될 위치의 주소값인 offset, chunk의 크기 등을 chunk DB에 저장하게 된다.

따라서 저장된 파일을 불러오기 위해서 fingerprint DB에서 hash 값을 얻을 수 있으며, 얻어낸 hash값을 이용하여 chunk가 저장된 실제 위치를 알아낼 수 있는 구조를 가지고 있다[8].

Fig. 1은 LessFS가 작동하는 개괄적인 구조, Fig. 2는 LessFS의 작동 예시를 도식화한 것이다.

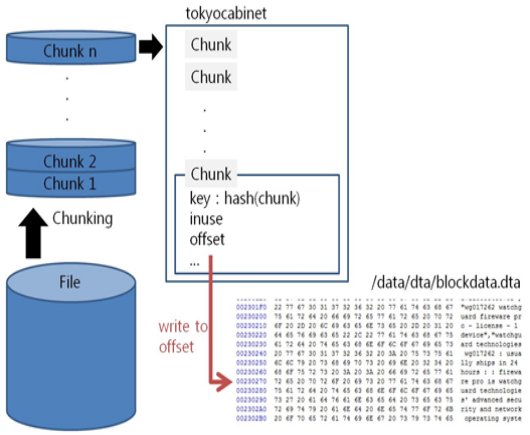

Fig. 1. Structure of LessFS

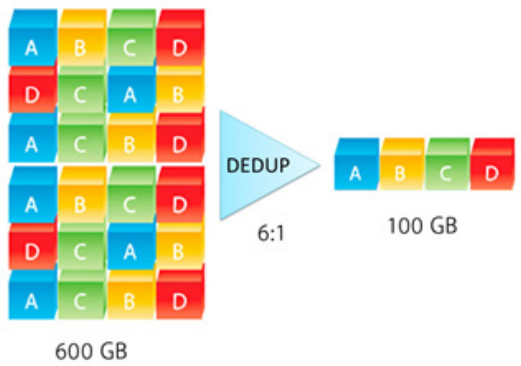

Fig. 2. Example of LessFS

3.3 fingerprint DB 구조

LessFS에서 fingerprint DB는 default의 경우 /data/mta/fileblock.tch 에 생성된다.

저장되는 파일의 정보를 나타내는 부분의 시작을 알리는 header는 'LESSFSTESTTRANSACTIONLESSFSTESTTRANSACTION(15 15 4C 45 53 53 46 53 54 45 53 54 54 52 41 4E 53 41 43 54 49 4F 4E 4C 45 53 53 46 53 54 45 53 54 54 52 41 4E 53 41 43 54 49 4F 4E B0 3E 00 00(16))'이다.

해당 부분부터 62Byte 간격으로 블록 정보가 기록된다. 하나의 블록에 대한 데이터가 저장되는 구조는 Fig. 3에 나타나 있다.

LessFS이 fingerprint DB는 파일 고유의 inode값(inode value)과 나누어진 블록 번호(blocknr)을 기본 키로 사용하고, hash value를 값으로 한다.

3.4 chunk DB 구조

Chunk DB는 default의 경우 /data/mta/blockusage.tch 에 생성된다.

저장되는 파일의 정보를 나타내는 부분의 시작을 알리는 header는 fingerprint DB의 경우와 같다.

chunk DB의 경우 78Byte 간격으로 chunk 정보가 기록된다. 하나의 chunk에 대한 데이터가 저장되는 구조는 Fig. 4에 나타나 있다.

LessFS의 chunk DB는 hash value를 기본 키로 하며, 해당 chunk의 데이터가 저장된 위치를 가르키는 offset값, 할당 크기(blocksize), 해당 chunk가 저장된 크기(chunksize), 해당 chunk가 사용된 횟수(inuse)를 값으로 한다.

IV. 완전 삭제의 구현

4.1 LessFS의 삭제 프로세스

앞서 살펴본 바와 같이 LessFS는 파일의 블록 정보와 chunk를 DB로 관리한다.

파일을 삭제하기 위해 fingerprint DB를 이용하여 블록별로 얻어낸 hash값을 chunk DB에 질의

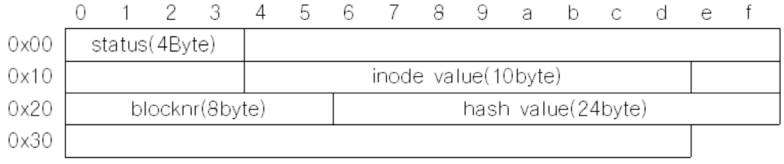

Fig. 3. Structure of fingerprint DB

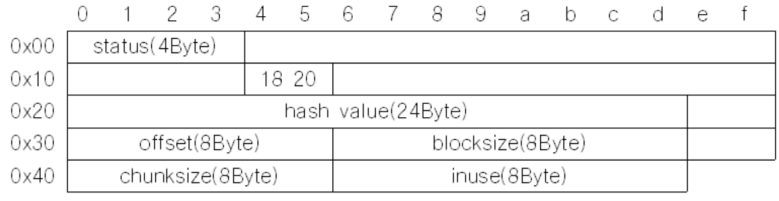

Fig. 4. Structure of chunk DB

한다. chunk의 inuse값이 1인 경우 해당 chunk는 더 이상 사용되지 않기 때문에 추후에 새로운 기록을 할 수 있도록 freelist로 추가한다. inuse값이 1 이상인 경우 해당 chunk는 계속 사용되어야 하기 때문에 inuse값을 감소시키는 데 그치게 된다.

fingerprint DB의 경우 삭제되는 파일의 key로 사용되는 inode값을 더 이상 질의할 수 없도록 제거한다.

4.2 삭제의 문제점

먼저 fingerprint DB의 경우 파일이 삭제되면 inode값을 DB상에서는 더 이상 질의할 수 없으나, 삭제되는 key를 사용하였던 개체들의 정보를 별도로 삭제하는 작업을 수행하지는 않는다. 따라서 DB 데이터가 저장되어 있는 fileblock.tch파일을 바이너리 분석하는 경우 파일의 inode값과 블록 번호를 여전히 찾을 수 있다. hash value역시 삭제되지 않았기 때문에 이를 이용하여 알아낼 수 있다.

chunk DB역시 key가 삭제되는 경우 DB상에서는 질의할 수 없으나, blockusage.tch파일을 바이너리 분석하는 경우 fingerprint DB에서 얻어낸 hash value를 검색할 수 있다. 이를 이용하여 chunk의 데이터가 저장된 offset을 알아낼 수 있다.

inuse값이 1인 경우 chunk를 최종적으로 삭제하여야 하는데, LessFS의 삭제 작업은 이를 freelist에 저장하는 것에 그치고 있을 뿐이다. 따라서 해당 chunk의 offset에는 삭제된 파일의 정보가 여전히 남아 있다.

inuse값이 2 이상인 경우는 DB 자체가 삭제되지 않기 때문에 역시 offset을 알아낼 수 있다. 또한 chunk역시 여전히 유효하게 사용되기 때문에 offset에 해당 내용이 저장되어 있다.

따라서 삭제된 파일의 inode값을 알고 있다면, 이를 fileblock.tch와 blockusage.tch를 차례로 검색하여 offset주소를 알아낼 수 있고 이를 block별로 완전히 복원해낼 수 있게 되는 것이다.

4.3 데이터 영역의 완전 삭제

삭제된 데이터를 복원할 수 없도록 완전 삭제를 구현하기 위하여 더 이상 사용되지 않는 chunk의 경우 이를 식별할 수 없는 값으로 덮어씌워야 한다. Alg. 1은 이러한 과정을 알고리즘으로 나타낸 것이다.

Alg. 1. Sanitization of chunk

```
(1) while(blocknr){
(2)    hash = fingerprintDB(inode,blocknr)
(3)    if (chunkDB(hash)->inuse==1)
(4)    overwrite(chunkDB(hash)->offset,
           chunkDB(hash)->chunksize)
(5)    else chunkDB(hash)->inuse--
(6) }
```

함수 fingerprintDB는 inode와 blocknr을 매개변수로 하여 hash값을 반환한다. 함수 chunkDB는 hash값을 매개변수로 하여 hash값을 key로 가지는 chunkDB의 포인터를 반환한다.

inuse값이 1인 경우 삭제 작업으로 인해 더 이상 사용되지 않는 chunk이므로 overwrite함수를 통해 offset위치의 chunksize만큼의 크기를 덮어씌운다. overwrite함수는 완전한 삭제를 위해 7번 덮어씌우도록 구현되었다[9].

4.4 fingerprint DB영역의 완전 삭제

앞서 4.3과 같이 완전 삭제가 구현되는 경우 inuse가 1인 경우는 식별할 수 없는 값으로 덮어씌워졌기 때문에 더 이상 내용을 복원할 수 없다. 그러나 inuse가 2이상인 경우는 다른 파일에서 사용되는 chunk이기 때문에 삭제되지 않고 여전히 내용이 보존되어 있다.

4.2에서 살펴본 바와 같이 fingerprint DB와 chunk DB에서 hash값과 이에 따른 offset등을 알아낼 수 있기 때문에 inuse가 2이상인 chunk의 경우 이를 이용하여 파일을 여전히 부분적으로 복원해낼 수 있다.

따라서 이러한 참조가 불가능하도록 fingerprint DB영역의 hash value영역을 식별할 수 없는 값으로 덮어씌워야 한다. Alg. 2는 이러한 과정을 알고리즘으로 나타낸 것이다.

Alg. 2. Sanitization of fingerprint DB

```
(1) while(blocknr){
(2)    overwriteDB(inode,blocknr)
(3) }
(4) delete_key(inode)
```

함수 overwriteDB는 inode와 blocknr을 매개

변수로 하여 fingerprint DB상에 있는 매개변수 값들을 키로 가지는 DB를 질의하여 hash value의 24Byte구간을 식별할 수 없도록 덮어씌운다. 역시 완전 삭제를 위해 7번 덮어씌우는 작업을 수행한다.

delete_key는 DB상에서 inode값을 삭제하는 작업으로 기존의 LessFS의 삭제 작업에서 존재하는 것과 동일하다.

fingerprint DB영역까지 완전 삭제가 적용되는 경우, inuse가 1인 chunk는 물론 inuse가 2 이상인 chunk 역시 fingerprint DB상에서 hash value구간이 식별할 수 없는 값으로 덮어씌워졌기 때문에 참조할 수 없어 복원이 불가능하다.

Fig. 5는 지금까지 구현된 완전 삭제 작업을 도식화한 것이다.

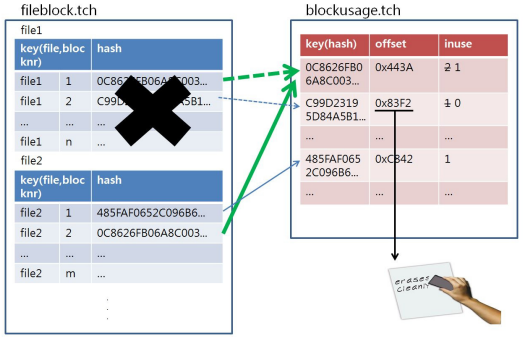

Fig. 5. Sanitization process

V. 성능 테스트

5.1 실험 환경

실험에 사용된 워크스테이션은 HP에서 제조한 마이크로서버인 'HP ProLiant Gen8'이다. CPU로는 Intel Pentium G2020T를, RAM은 4GB의 Unbuffered DDR3 메모리가 Dual-Channel로 구성되어 있다. 하드디스크는 2TB크기의 Serial-ATA 3규격을 사용하고 있다.

OS는 CentOS 6.7버전을 사용하였다. LessFS는 주로 Fedora/RedHat기반의 Linux에서 잘 작동하도록 설계되었다.

5.2 dataset의 구성

중복 제거와 관련하여 real-life user data의 구성을 경험적 연구의 방식으로 분석한 여러 선행 연구들이 존재한다[10,11].

선행 연구들을 바탕으로 생성한 dataset의 구성은 Table. 1과 같다.

Table 1. Organization of dataset

Type	Files	Capacity
jpg	2254(32.50%)	1044MB(23.36%)
etc	315(4.54%)	993MB(22.22%)
pdf	2173(31.33%)	503MB(11.26%)
xls	247(3.56%)	451MB(10.09%)
ppt	167(2.41%)	405MB(9.06%)
doc	322(4.64%)	403MB(9.02%)
avi	15(0.22%)	333MB(7.45%)
zip	2(0.03%)	110MB(2.46%)
mp3	11(0.16%)	108MB(2.41%)
exe	10(0.14%)	74MB(1.66%)
txt	1419(20.46%)	45MB(1.01%)
Total	6935	4469MB

5.3 실험 방법

5.2에서 다루었던 dataset을 바탕으로 삭제 작업을 수행한다.

실험군으로 완전 삭제가 적용된 LessFS 스토리지에서 dataset을 저장하고 삭제 작업을 수행한다.

대조군으로 완전 삭제가 적용되지 않은 LessFS 스토리지에서 dataset을 저장하고 삭제 작업을 수행한다.

chunk의 크기는 각각 4096, 16384, 65536, 131072 Byte로 달리하여 설정한다.

case마다 삭제에 소요된 시간을 각 5회씩 측정하여 평균치를 구한다.

비교를 위해 중복 제거가 없는 일반적인 리눅스 기반의 Ext4파일시스템에서 삭제 시간을 측정하였다.

실험 환경은 5.1에서 설명한 바와 같으며, 리눅스와 유닉스에 기본 내장된 shred명령어를 사용하여 dataset에 대한 완전 삭제를 수행하였다.

삭제에 소요된 시간을 각 5회 측정한 평균치로 14:47.364의 소요 시간을 얻을 수 있었다.

5.4 실험 수행 결과

dataset을 chunks 크기에 따라 각각 LessFS 스토리지에 저장한 결과는 Table. 2와 같다. chunk 크기를 작게 설정할수록 chunk의 수가 늘

Table 2. Saving dataset to storage of LessFS

chunksize (Byte)	data (MB)	DB(MB)	write time
4096	4996	218	4:26.484
16384	4582	103	2:22.062
66536	4479	75	2:18.415
131072	4462	70	2:12.677

어나기 때문에 이를 위한 DB영역의 크기가 증가하는 것은 당연하다. 반면, 데이터영역의 크기는 chunk 크기가 커질수록 감소하는데, 이는 LessFS의 할당 단위가 512 Byte이기 때문에 이를 위해 chunk마다 여유 공간을 확보하는 과정에서 점유하는 공간의 크기가 증가하는 것으로 보인다.

실험을 수행한 결과는 Table. 3와 같다. 완전 삭제가 적용된 경우를 SD로, 완전 삭제가 적용되지 않은 경우를 Non-SD로 나타내었다.

Fig. 5 는 Table. 3의 결과를 그래프로 나타낸 것이다. 완전 삭제가 적용된 경우와 적용되지 않은 경우의 차이가 크기 때문에 값을 log-scale로 변환하였다. 또한 중복 제거가 없는 경우의 소요 시간을 ND로 나타내었다.

Table 3. Elapsed time on deletion

chunksize (Byte)	Non-SD	SD
4096	1:17.415	94:58.738
16384	0:19.438	23:51.952
65536	0:05.871	7:59.789
131072	0.03.850	5:47.129

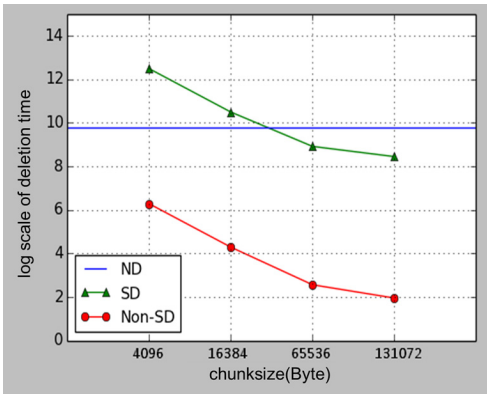

Fig. 6. Elapsed time on deletion

5.5 평가

완전 삭제가 적용된 경우의 삭제 작업은 완전 삭제가 적용되지 않은 경우에 비해 약 60~70배 가량의 시간을 소모하고 있다. LessFS는 background delete 모드가 지원되고 있으나 해당 실험의 측정을 위해 해당 옵션을 의도적으로 off하였다.

전반적으로 chunk 크기만큼 chunk의 수가 감소하기 때문에 그의 배수만큼 소요 시간이 감소하는 추세를 확인할 수 있다.

중복 제거 방식에 비추어 볼 때, 덮어씌워야 하는 데이터 영역의 크기는 chunk 크기와 상관없이 상대적으로 일정한 크기를 가진다. 따라서 chunk 크기 감소에 따른 소요시간 증가는 chunk수의 증가에 의한 DB영역의 덮어쓰기와 이에 따른 오버헤드 증가에 기인함을 알 수 있다.

그럼에도 불구하고 chunk 크기가 65,536Byte 이상으로 커지는 경우 완전 삭제를 하더라도 중복 제거가 없는 경우에 비하여 삭제 작업이 빠른 시간 안에 완료되는 것을 확인할 수 있다. 이러한 결과는 중복 제거 스토리지에서의 쓰기 성능을 측정한 여러 선행 연구들에서 밝혀진 바와 같다[7].

일반적으로 chunk 크기를 4,096Byte이하의 수준으로 작게 설정하는 것은 성능 측면 에서 권장되지 않는다[6]. 그럼에도 불구하고 fingerprint DB 와 관련한 작업이 수행 시간 증가의 대부분을 차지하는 현상은 중복 제거 스토리지를 도입함에 있어 커다란 장애물이 될 수밖에 없다. 보다 효과적인 방안을 모색할 필요가 있을 것으로 보인다.

VI. 결 론

본 연구에서는 중복 제거 파일시스템의 구조를 살펴봄으로써 완전 삭제를 적용하는 데 어려움이 있는 원인을 살펴보았다.

대표적인 오픈 소스 중복 제거 파일시스템 중 하나인 LessFS를 이용하여 파일의 삭제로 인해 chunk가 더 이상 사용되지 않게 되더라도 내용이 덮어씌워지기 전까지 그대로 남아있다는 점, 파일이 삭제되더라도 fingerprint DB 역시 바이너리 분석을 통해 알아낼 수 있다는 점을 이용하여 삭제된 파일의 일부 또는 전부를 복원할 수 있음을 보였다.

이러한 복원을 막기 위해 데이터 영역과 fingerprint DB영역에 7passes 삭제 알고리즘을

이용한 완전 삭제 방법을 제안하였다.

성능 분석 결과 완전 삭제가 적용되는 경우, 그렇지 않은 경우에 비해 60~70배 가량의 작업 시간을 필요로 하였으나, chunk 크기가 65,536Byte이상으로 커지는 경우 중복 제거가 없는 일반 파일시스템에 비해서 성능 향상이 있음을 보였다.

본 연구에서의 구현은 LessFS를 바탕으로 이루어졌지만, chunk의 완전 삭제가 이루어지지 않는다는 점과 hash index list를 이용하여 파일의 일부 내용을 복원할 수 있다는 점은 SDFS등 다른 오픈 소스 중복 제거 파일시스템에도 적용될 수 있는 문제점이기 때문에 이러한 복원을 방지할 수 있는 방법론을 제시하였다는 점에서 본 연구의 의의가 있다고 하겠다.

이러한 점을 바탕으로 향후 연구에서 다른 중복 제거 파일시스템에 유사한 방법론을 적용할 수 있을 것이다. 또한 완전 삭제가 아닌 경우와의 성능 차이가 크기 때문에 보다 효과적인 알고리즘을 모색할 필요가 있을 것으로 보인다.

References

〔1〕 R. Kissel, M. Scholl, S. Skolochenko, and X. Li, "Guidelines for Media Sanitization," September 2006, National Institute of Standards and Technology, 2006.

〔2〕 M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably Erasing Data From Flash-Based Solid State Drives", *9th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, pp. 105 - 117, 2011.

〔3〕 F. C. Botelho, P. Shilane, N. Garg, and W. Hsu, "Memory efficient Sanitization of a Deduplicated Storage System", *11th USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, pp. 81 - 94, 2013.

〔4〕 D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: scalable, parallel deduplication for chunk-based file backup", In Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 1-9, 2009.

〔5〕 Detecting duplicate and near-duplicate files, US Patent 6658423 Issued on December 2, 2003.

〔6〕 D. Harnik, E. Khaitzin, and D. Sotnikov (2016), "Estimating Unseen Deduplication - from Theory to Practice", 14th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, pp. 277 - 290, 2016.

〔7〕 Sung-ouk Jung, and, Hoon Choi, "Performance Analysis of Open Source BasedDistributed Deduplication File System", KIISE Transactions on Computing Practices, Vol.20, No. 12, pp. 623-631, 2014.

〔8〕 Val Henson, An analysis of compare-by-hash, 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003.

〔9〕 P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," *USENIX Security Symposium*, pp. 77 - 89, 1996,

〔10〕 D. Meister and A. Brinkmann, "Multi-level Comparison of Data Deduplication in a Backup Scenario", In Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR), pp. 8:1 - 8:12, (2009).

〔11〕 D. T. Meyer, and W. J. Bolosky (2011), "A Study of Practical Deduplication", 9th USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, pp. 1-14, 2011.

〈저 자 소 개〉

조 현 웅 (Hyeonwoong Cho) 학생회원
2014년 2월: 연세대학교 경제학과 졸업
2016년 8월: 연세대학교 정보대학원 석사
〈관심분야〉 디지털 포렌식, 딥 러닝, 분산 컴퓨팅

김 슬 기 (SeulGi Kim) 학생회원
2016년 2월: 순천향대학교 정보보호학과 학사
2016년 3월~현재: 연세대학교 정보대학원 석사과정
〈관심분야〉 HCI 보안, Usable Security, 머신러닝, 센서네트워크 보안 등

권 태 경 (Taekyoung Kwon) 종신회원
1992년 2월: 연세대학교 컴퓨터과학과 학사
1995년 2월: 연세대학교 컴퓨터과학과 석사
1999년 8월: 연세대학교 컴퓨터과학과 공학박사
1999년~2000년: U.C. Berkely Post-Doc.
2001년~2013년 8월: 세종대학교 컴퓨터공학과 교수
2007년~2008년: Univ. Maryland at College Park 교환교수
2013년 9월~현재: 연세대학교 정보대학원 교수
〈관심분야〉 암호프로토콜, IoT 보안, 소프트웨어 보안, HCI 보안 등