Vol.2015-DPS-162 No.40
Vol.2015-CSEC-68 No.40
2015/3/6

セクタハッシュと並列分散処理を用いた 大量の書き込み履歴データからの目的ファイルの高速な検出

平野学[†1]　吉田光輝[†1]　高瀬誉[†1]

コンピュータを使ったインシデントや犯罪に関係する外部記憶装置の容量が急速に増加している．検査者は複数のストレージ装置から得た数十テラバイトのデータから証拠となる電子データを高速に探し出す必要がある．本稿ではブロックデバイスへの出力からセクタ単位でハッシュ値を作ることで，ファイルを高速に探し出す手法を検討した．本稿ではセクタハッシュ方式に関する先行研究の調査結果を示し，方式の有効性を確かめるために Windows 8.1, MacOS 10.9, CentOS 6.5 のそれぞれに対して，セクタの一意性と境界配置に関する予備実験を行った．その後，我々が先行研究で開発した仮想計算機モニタを用いた履歴保全システムに対して，セクタハッシュ方式の検索を実装した結果を示した．MapReduce によって転置ファイルを作成し，MapReduce による単純な逐次探索の分散処理，MySQL の B tree インデックスを有効にした検索，の二通りにて性能評価を実施した．

Rapid Target File Detection in a Large Amount of Preserved Data by Using Sector Hashes and Parallel Distributed Processing

MANABU HIRANO[†1]　KOKI YOSHIDA[†1]
HAYATE TAKASE[†1]

The amount of data that is related to incidents or crimes with computers is growing rapidly. An investigator has to find digital evidence from several dozens of terabytes of data on seized storage devices. This paper discusses a method for finding a target file by using hash values generated from each sector's data. We show several related work of sector hashing. We present two experimental results of distinctness of sectors and data alignment on Windows 8.1, MacOS 10.9, and CentOS 6.5. This paper shows a prototype implementation of the sector hash method for our hypervisor-based data preservation and restoration system. The system employs MapReduce for making inverted indices. We show two prototype implementations, a simple sequential search program of MapReduce and a search program by using MySQL database with B tree indexing.

1. はじめに

コンピュータに関係する様々な事件や紛争が増え，それらの証拠となる電子データの取り扱いが重要な課題となっている．それらの紛争解決の手段としてデジタル・フォレンジック（Digital Forensics, DF）という用語が用いられており,「インシデント･レスポンスや法的紛争･訴訟に対し，電磁的記録の証拠保全及び調査・分析を行うとともに，電磁的記録の改ざん・毀損等についての分析・情報収集等を行う一連の科学的調査手法・技術」として定義されている[1].デジタル・フォレンジックの技術は 40 年以上の歴史があるが[3]，特に 2000 年に入ってからは電子機器が犯罪に広く使われるようになったため，方法論の標準化[2]が進んだ．デジタル・フォレンジックの方法論は法執行機関だけでなく，米国で訴訟される側が証拠を法廷へ提出する e-Discovery 手続きにも適用されている．本稿では法廷に関わる証拠だけでなく，現場の管理者が様々なインシデントの原因を探るためのより広義のデジタル・フォレンジックの技術について議論する．デジタル・フォレンジックの手順は文献[2]によると，（１）インシデントや事件の識別，（２）準備，（３）戦略の決定，（４）証拠の保全，（５）回収，（６）検査，（７）分析，（８）文書の作成，（９）証拠の返却，から構成される．本稿では特に（６）の検査を効率的に実施するための技術的課題に取り組む．具体的にはインシデントや犯罪の証拠となるファイルを高速に探す枠組みである．

ところで，Garfinkel は “Digital forensics research: The next 10 years” [3] の中で今後の研究課題をいくつか列挙したが，その中で “Visibility, Filter, and Report” モデルで作られている現在の多くのデジタル・フォレンジックのソフトウェアが抱える問題を指摘した．EnCase に代表されるツールは（１）回収されたメディアの全データを解析してツリー構造などで可視化し，（２）検査者はテーブルに表示された全データから個々のデータの詳細を見て，（３）フィルタ機能で表示内容を減らしながら，（４）キーワード検索でコンテンツを探し，（５）最後にレポートを作成する，という手順で利用されている．Garfinkel らはこの“Visibility, Filter, and Report” モデルを採用したソフトウェアの問題のひとつが並列処理の難しさであると指摘した．たしかに上記モデルのツールは人間が関与する手順が多いので機械的に並列処理するのは難しい．このモデルとは別に，ファイルの断片のみを見て該当ファイルを探し出すカービング (Carving) と呼ばれる手法も９０年代から研究されている[9]．この手法はファイルシステムを考慮しないため，単純にデータを分割できるので並列処理に向いている．しかしながら，ファイルシステムを解釈しないで意味のあるデー

†1 独立行政法人国立高等専門学校機構，豊田工業高等専門学校
National Institute of Technology, Toyota College

タを取り出すことには限界もあり，目的に応じてツールを使い分ける必要があると考えられる．本稿ではセクタハッシュというカービングの一手法の適用について議論する．

デジタル・フォレンジックを高速に行う必要性は外部記憶装置の容量の増加にも関係している．具体例を挙げて説明すると，SATA3 規格 (最大理論値 6Gbps, 実測値 400 MB/sec 程度) の SSD 2TB（本稿執筆時点で 1 万円程度で購入できる）を 5 台回収したと仮定した場合，解析のためにデータを複製するだけで 7 時間以上を要する．データ解析には “Visibility, Filter, and Report” モデルのツールを使うことになるが，フィルタやキーワード検索で個々のイメージを調査するには手間がかかる．ツールは転置インデックスを自動的に内部で作成し，ディスクアクセスを最小化する場合が多いが，仮に転置インデックスの対応していないキータイプに対して O(n) の逐次探索を実行すると 2TB 全体のデータを読み込むだけでも 1 時間 20 分ほどを要してしまう．以上のような課題を解決するため，様々な研究者が，分散処理で証拠データの発見を高速化できないかを検討してきた．Roussev らは研究[4]にて，解析対象の 6GB ディスクイメージをファイル単位で 8 台のコンピュータに振り分け，各コンピュータで AccessData 社の FTK を実行し 1GB の RAM に解析対象データを全てキャッシュさせ，管理用コンピュータから 1Gbps の LAN 経由で命令を送って，結果を集約する並列化手法を試作した．Roussev らはディスクアクセス速度に起因するツールの応答速度の問題を，分散コンピュータのメモリに乗せる方法で解決した．Marziale らは研究[5]において，ファイル・カービングを行うソフトウェアである Scalpel を GPU で高速化した．Scalpel は改良された Boyer-Moore アルゴリズムによってバイナリデータからファイルタイプなどの文字列を検索するが，その部分を GPU で並列処理した．

さて，以上で述べてきたようにデジタル・フォレンジックの研究は機能面，性能面，方法論について幅広く研究されてきている．我々は以前の研究報告[6]で，仮想計算機モニタのデバイスドライバを用いて外部記憶装置への書き込みをセクタ単位で時刻情報とともに逐次すべて記録していくシステムを開発した結果を報告した．我々の提案システムはインシデントや犯罪が起きたときに現場で証拠を回収，保全するものではなく，例えるなら監視カメラのように事前に設置して記録をとっておき，必要なときに過去の記録を効率的に探し出す仕組みに近い．同じような概念のシステムには手塚らのクラウドストレージに履歴追跡の機能を付与したファイルシステム[7]がある．本稿のシステムには書き込みの保存（保全）機能があるが，手塚らのシステムが持つ完全性や順序性を保証する仕組みは含んでいない．本稿では，デジタル・フォレンジックで重要な作業となるイベントを時系列で再構成していくタイムライン構築[8]に焦点をあて，その支援を並列処理で高速化する方法を検討する．最終的には “Visibility, Filter, and Report” モデルで作られている EnCase や TSK のような実績あるフォレンジック・ソフトウェアと相互補完しながら，効率的な検査を実現するシステムの構築を目指す．

本稿では，まずセクタハッシュ方式に関するファイル検出の先行研究を示した．次に，セクタハッシュの有効性を検証するために実施したセクタの一意性に関する実験と，境界配置に関する実験の結果を示した．そして，先行研究の仮想計算機モニタを用いた履歴保全システムからの検索に，セクタハッシュ方式を実装した結果を示した．転置インデックスの作成には MapReduce を使った．転置インデックスからの検索には MapReduce による逐次探索と B tree インデックスを有効にした MySQL からの検索の二通りで実装した結果を示した．最後に性能評価をおこない，大規模データからの検索の見通しについて議論した．

2. セクタハッシュ方式の先行研究

前述のようにファイルの断片のみを見て該当ファイルを探し出す手法をカービング (Carving) と呼び，この分野は９０年代から研究されている[9]．本節では特にセクタハッシュに関する関連研究をまとめる．「セクタハッシュ」とはブロック単位でデータを読み書きする装置において，読み書きの単位となるセクタ（512 バイトが主流であるが，新しい装置は 4096 バイトになる）をハッシュ関数の入力として与えた際の出力の値を指す用語として用いる．なお，セクタとブロック，クラスタ，アロケーションユニットサイズなどの用語はどれも一定サイズのデータ単位を指す言葉であるが，混同して使うとまぎらわしいため，本稿では「セクタ」はドライブイメージから得られたデータを扱う場合に，「ブロック」はファイルシステムまたはファイルから得られたデータを扱う場合に，それぞれ用いるように区別した．なお，本稿では特に断らない限りセクタの大きさは 512 バイトとした．

Garfinkel らは研究[9]でディスクのセクタやファイルから得たブロックの断片を使って特定のデータを高速に識別する手法を提案した．彼らは “distinct” disk sector の概念を導入し，その定義をオリジナルのコピー以外には存在し得ないセクタと述べた．彼らの仮説は以下のとおりである．[仮説１] もしファイルのあるブロックが distinct であれば，別な外部記憶装置に見つかった同じデータセクタの存在は，該当ファイルが一度は存在していたことを示している．[仮説２] もしあるファイルが高いエントロピーを発生させる処理で作られていると分かっている場合，かつそのファイルのブロックが大きなコーパス全体を通しても distinct であれば，それらのブロックは distinct として扱える．彼らは Windows XP SP3 のディスクイメージである nps-2009-domexusers データセットを使って，セクタサイズを 512, 1024, 4096 と変化させたときのセクタの 一意性

（distinctness）を調査した．彼らはセクタ全体が同じ定数で埋まっているセクタは除外した．その結果，セクタサイズを 512 バイトとした場合に 56%，セクタサイズを 4096 バイトとした場合に，ディスクイメージの約 60%のセクタが distinct（同じ内容のセクタが存在しない）であることを示した．さらに，彼らは National Software Reference Library から取得した約 770 万個のファイルについてブロック単位に分割したときの distinct なブロックの割合を調査し，87%のブロックが distinct であるという結果を示した．

Young らは研究[10]において，米国政府サイトから取得したファイルのコーパスである Govdocs（ファイル数 974,741），OpenMalware2012（ファイル数 2,998,898），2009 National Software Reference Library の Reference Data Set (RDS)（4096 バイト単位で 12,236,979 ブロック） の３つのデータセットに対して，ブロックを 512 バイト，4096 バイトにしたときのそれぞれで distinct なブロックの割合を調査した結果を示した．その結果，一回しか現れないブロックの割合は，上記のすべてのデータセットとブロックサイズについて約 88%以上であることを示した．彼らの研究によると，何度も現れるブロックには，同じバイトデータ（0x00）が連続するブロックのほかに，PDF や Office の内部データ構造について，特徴の同じブロックが複数発見されたことを示した．Young らは適切なブロックの大きさについて，NTFS のような多くのファイルシステムは 4096 バイトのブロックサイズを採用しているものの，ドライブ自体がまだ 512 バイトセクタの仕様が多いため，多くの場合にデータが 4096 バイト境界に配置されず，ハッシュ値によって検出できなくなることがあったと述べた．彼らはこの問題を 4096 バイトのブロックサイズを採用しながら，15 個の 512 バイトブロックを読み込んで，ブロックをひとつ毎にずらしながら合計 8 個の 4096 バイト単位のハッシュ値を作成することで回避した．

Young らは研究[10]で計算量が少ないことを理由に，セクタハッシュを作るアルゴリズムとして MD5 を採用した．ファイルを探す目的でハッシュデータベースを用いる場合には MD5 の衝突耐性の脆弱性は影響しないが，もし，ファイル検出を既知ファイルの除去に使う場合には，攻撃者が既知ファイルと同じファイルを作成して検出を回避する危険がある．サイズに関しては，MD5 は 128bit（16 バイト）のデータであるため 512 バイトのブロックに対して 1/32 のサイズでハッシュ値のデータベースを構築できる．4096 バイトのブロックに対して同様に MD5 を使うと 1/256 のサイズでハッシュ値のデータベースを構築できる．本稿の執筆時点で日本の政府推奨暗号 CRYPTREC の推奨暗号リストに含まれている SHA-256 を採用すると，MD5 と比べてハッシュ値のデータベースが２倍の大きさになる．安全性を高めればデータ量が大きくなり，検索性能が悪化するため，目的に応じた効率の良い設計をすることが必要である．

Garfinkel, Young, Farrell らの過去の研究[9][10][11]では大規模なハッシュ値のデータベースから検査者が短時間で効率的にファイルを検索するために様々な工夫をおこなっている．基本的には以下のような共通点がある．まず事前にハッシュ値をキーとする転置インデックスを作成する．インデックスは物理メモリに全て乗る場合にはハッシュ探索のようなアルゴリズムで性能が得られるが，そうでない場合には B tree を使ってインデックスを作成しディスクアクセスを最小限にする．転置インデックスだけで実用的な性能が出ない場合には，物理メモリにのるサイズの Bloom フィルタ [12] をあらかじめ作成しておく．Bloom フィルタとはデータが存在しているかを確認するために用いるデータ構造である．あらかじめ存在確認をしたいデータのハッシュ値を Bloom フィルタにセットしておき，Bloom フィルタ全体を物理メモリにのせておけば，高速に存在確認をおこなえる．Bloom フィルタは，偽陰性（False Negative）が原理上ゼロであるため，Bloom フィルタを使ってデータが存在しないことがわかってしまえば転置インデックスを検索する処理を省くことができる．偽陽性（False Positive）は Bloom フィルタのサイズ及び適用するハッシュ関数の数によって増減するため，物理メモリのサイズと偽陽性率のバランスを考えてフィルタを作成しておく必要がある．もし Bloom フィルタが CPU のキャッシュに乗れば非常に高速に存在確認を行えることが研究[11]の実験で示されている. 研究[9][10][11]では上記の方針を独自に実装して高速処理を実現しているが，機能性や汎用性を考慮すると実績のあるリレーショナルデータベースの採用も選択肢のひとつである. たとえば MySQL は B tree でインデックスを作成して高速化することが可能である[13].

3. セクタハッシュの一意性に関する実験

セクタハッシュ方式でのファイル検索を採用するには，同じハッシュ値を使って検索したときの誤検出が少ないかを確かめる必要がある. 2 節で述べたように Garfinkel らの先行研究[9] において人間が作ったファイルから得たセクタハッシュが distinct である割合はおおむね 87%以上であったことが示された．一方，彼らは別の先行研究[10]で Windows XP ディスクイメージのセクタハッシュ（512 バイト単位）が distinct な割合は 56% であったことを報告した．後者のデータは実験で使った OS が古くなっているため，本節では本稿執筆時点で主に利用されている３種類の OS についてディスクイメージを作成しセクタハッシュが distinct である割合を調査した．さらに，調査の結果 distinct なセクタの割合が低かった Windows 8.1 と CentOS に関して原因を調査するため，ファイル内容の重複に関する追加の実験をおこなった．

3.1 実験方法

著者らは前の報告[14]において Windows 8.1 (x64) と Microsoft Office 2013 をインストールしたディスクイメージ，MacOS 10.9 と Microsoft Office 2011 をインストールしたディスクイメージ，CentOS 6.5 (x64)を Basic Server でインストールしたディスクイメージの３種類に対してセクタごとの distinct なセクタの割合を調査した結果を示した．その結果を表 1 に再掲する. これは文献[9]と同様の方法で実施した結果であるが，以下に方法を説明する．まず，対象とするディスクイメージから 0x00 や 0xFF などの同一バイトが連続して 512 回出現するセクタ（全部で 256 種類ある）を除外する．その後，残ったセクタ（512 バイト毎）のハッシュ値を計算して，同じディスクイメージ中の他のセクタと比較をおこなうことで distinct なセクタの割合を求めた.

表 1　OS 毎のセクタの distinctness に関する実験結果

セクタ数	Windows 8.1 (NTFS)	MacOS 10.9 (HFS+)	CentOS 6.5 Linux (Ext4)
合計セクタ数	195,371,568	156,301,488	195,371,568
除外セクタ数	170,738,500	136,997,813	169,699,709
対象セクタ数	24,633,068	19,303,675	25,671,859
Singleton（１回）	12,562,375 (51.0%)	17,090,144 (88.5%)	15,718,575 (61.2%)
Twin（2 回）	8,067,594 (32.7%)	1,141,746 (5.9%)	6,767,062 (26.3%)
Triplet（３回）	2,160,450 (8.7%)	121,257 (0.6%)	1,542,480 (6.0%)
４回以上	1,842,649 (7.5%)	950,528 (4.9%)	1,643,742 (6.4%)

表 1 をみると完全にユニークなセクタ（Singleton，一人っ子）は Windows 8.1 で 51%，MacOS 10.9 で 88.5%，CentOS 6.5 (Linux) で 61.2 %であった．２回出現している Twin（双子），３回出現している Triplet（三つ子）のパーセンテージが Windows 8.1 と CentOS 6.5 で高かったのはシステムファイルで同一のファイルが多いからと推定した.

本稿では上述の同一ファイルの多さが distinct なセクタ数に与える影響を確認するために, 以下の追実験を行った. まず, 表 1 の実験と同じ３種類のディスクイメージをファイルシステムにマウントして，ファイル毎の MD5 のハッシュ値を計算して，同一ハッシュ値かどうかを調べることでファイル内容が同一かどうかを判定した．そのファイルサイズから占有セクタ数をもとめ，n 個の同一ファイルがある場合にはその占有セクタ数を n 倍した結果の合計を求めた．占有セクタ数はファイルサイズから 512 バイトの倍数となるように求めた．例えば，512 バイト未満のファイルは１セクタ，720 バイトの場合は 1024 バイト未満なので２セクタとした.

3.2 実験結果

表 2 に同一ファイルの占めるセクタ数の割合を求めた結果を OS ごとにまとめた．この結果では合計セクタ数が表 1 の対象セクタ数よりも大きな数になっている．これは同じバイトが連続するセクタを除外していないことが原因であると考えられる．しかし，同一ファイルがどれくらいの割合で存在しているかを把握することは可能であると思われる．Windows 8.1 の結果をみると推測どおり，セクタ全体の 56.3%が，初期インストール状態で２つ以上の同一内容のファイルを保存するのに使われていることがわかった．この同一内容のファイルの占める割合の多さは，表 1 の Singleton（１回だけ出現するセクタ）の割合の少なさの要因のひとつであると考えられる．一方，MacOS 10.9 では同一ファイルはほとんど存在せず，セクタ全体の 96.5%がユニークなファイル内容のもので占められていることが判明した. この結果は表 1 に示したとおり MacOS 10.9 で Singleton のセクタが 88.5%と他 OS に比べて多かったことと一致している．最後に CentOS 6.5 であるが，同一内容のファイルが 94.3%を占めており，同一内容のファイルが２個以上あるセクタの占める割合は 5.7% と予想よりも少なかった.

表 2　ファイルコンテンツから求めた重複セクタ数

ファイルが占有するセクタ数	Windows 8.1 (NTFS)	MacOS 10.9 (HFS+)	CentOS 6.5 Linux (Ext4)
同一内容のファイルが他にない	16,708,798 (43.7%)	23,389,715 (96.5%)	3,203,920 (94.3%)
同一内容のファイルが２個存在	16,996,756 (44.5%)	378,948 (1.6%)	159,408 (4.7%)
同一内容のファイルが３個存在	2,383,044 (6.2%)	66,819 (0.3%)	15,150 (0.5%)
同一内容のファイルが４個以上存在	2,147,244 (5.6%)	401,032 (1.7%)	20,320 (0.6%)
合計	38,235,842	24,236,514	3,398,798

3.3 考察

Linux の場合ファイル単位では完全に同一のファイルの数は少なかったが，ファイルをセクタ単位で分割した場合には同一セクタが多く出現するという特性があることが分かった．オープンソースのシステム特有の特徴ではないかと推測できるが, 今回の実験結果で得た CentOS 6.5 の初期インストール状態のディスクイメージで distinct なセクタが少ない理由については，更なる検証が必要である.

本節の実験結果をまとめると，Windows 8.1 と MacOS 10.9 の初期インストール状態のディスクではセクタハッシュ方式によるファイル特定方法がある程度は有用と考えられる．他方，CentOS 6.5 のファイルは，セクタハッシュを使って同一内容のファイルを探す場合にも，同一セクタを含む別なファイルを誤検出する可能性が高いことがわかった．本節で示した結果はシステムファイルや設定ファイルなどの，初期状態で存在するファイルに対して，セクタハッシュを適用した場合に誤検出がどれくらい発生するかを見積もるために利用できる．初期状態のシステムファイルに対するセクタハッシュの代表的な適用例は，既知の known-good （良性）ファイルを探索対象から除外する場合である．Windows 8.1 や MacOS 10.9 では known-good を除外するのに，セクタハッシュを利用できる可能性があることが確認できた．ただし，known-good のリストを作る場合には衝突耐性の問題のないハッシュアルゴリズムを採用する必要があることには留意する必要がある．

4. セクタハッシュの境界配置に関する実験

セクタハッシュを使った検索方式の前提条件に「データがセクタの境界に整列して配置され，決して境界をまたがって配置されない」ことがあげられる．この条件を満たさないと，ファイルのセクタハッシュをキーとして，ディスクイメージのセクタハッシュを探索しても見つからないデータがでてきてしまう．本節では 3 節で使ったのと同じテストデータを使い，セクタハッシュの境界配置に関する実験を実施した結果を示す．

4.1 実験方法

Windows 8.1 (x64) と Microsoft Office 2013 をインストールしたディスクイメージ，MacOS 10.9 と Microsoft Office 2011 をインストールしたディスクイメージ，CentOS 6.5 (x64)を Basic Server でインストールしたディスクイメージの３種類について，ファイルシステムにマウントし，すべてのファイルに対してセクタハッシュを計算し，ファイルのセクタハッシュ値のリストを作成した（例えば 700 バイトのファイルなら 512 バイトのデータと残り 188 バイトの末尾をゼロで埋めたデータの２つに対してハッシュ値を計算する）．次に，ディスクイメージを先頭から読み込み，ディスク全体のセクタハッシュのリストを作成した．最後に両者のリストを照合し，見つからなかったセクタハッシュの数を数えた．

4.2 実験結果

表 3 に OS ごとの実験結果を示す. Windows 8.1 は NTFS をファイルシステムとして採用しているが，512 バイト未満のファイルの 85.84%が境界にデータが配置されておらず，セクタハッシュ方式で検索できないことが分かった．対して，MacOS 10.9 で採用されている HFS+ ファイルシステムと Linux で採用されている ext4 ファイルシステムはセクタ境界にデータが必ず配置されており，検索できないファイルは存在しなかった．

表 3　境界配置に関する実験結果

ディスク全体に対する見つからなかったセクタの割合	Windows 8.1 (NTFS)	MacOS 10.9 (HFS+)	CentOS 6.5 Linux (Ext4)
512 バイト以下のファイル	85.84%	0%	0%
513 バイト以上 1KiB 未満のファイル	1.76%	0%	0%
1 KiB 以上 1 MiB 未満のファイル	0.0001% (20 ファイル)	0%	0%
1MiB 以上のファイル	0%	0%	0%

4.3 考察

Windows 8.1 で採用されている NTFS ファイルシステムでは Master File Table (MFT) と呼ばれる 1KiB のファイルレコードでファイルを管理している[15]．表 3 で見つからなかったファイルのサイズを調査した結果，見つからなかったファイルのサイズは一部の例外（表 3 の 0.0001%, 20 個のファイル）を除き，全て 720 バイト以下であった．ファイルサイズが小さい場合には，1KiB の MFT ファイルレコードにメタデータとともにファイルのコンテンツも格納されるため，データがセクタ境界に揃わなくなったと推定できる．ただし，上記の条件を満たさない 1KiB 以上のファイルで 0.0001%が発見できなかった点については，今回の実験結果の正確さも含めて追加の調査が必要である．

NTFS はクライアント向け PC で最も一般的なファイルシステムであるためセクタハッシュ方式にも対応させる必要がある．ちなみに，我々の調査結果によると，実験に用いた Windows 8.1 と Office 2013 をインストールした初期状態のイメージで MFT に収まる可能性のある（つまりセクタハッシュ方式で検索できない可能性の高い）1KiB 未満のファイルの数は，実に全体のファイル数の 70% を占めていた．対策として，MFT ゾーンはボリュームの先頭の 12.5 パーセントとして予約されているため[16]，MFT ゾーンの該当セクタのみ，境界配置の問題を考慮してセクタハッシュを別な方法で作ることが考えられる．

5. 履歴保全システム

本研究の目的は，セクタハッシュ方式を使ったファイル検索を，先行研究[6]で我々が開発した履歴保全システムに適用することである．提案するシステムの全体構成を図 1 に示す．先行研究で開発した履歴保全システムは，仮想計算機モニタ Xen（準仮想化）の Blktap 機構を利用して，外部記憶装置への書き込みデータを 4096 バイトのブロック

単位でブロック番号，時刻情報とともに保全する仮想デバイスドライバと，解析時に任意時刻のディスク状態を復元してマウントするための仮想デバイスドライバの二つから構成されている．

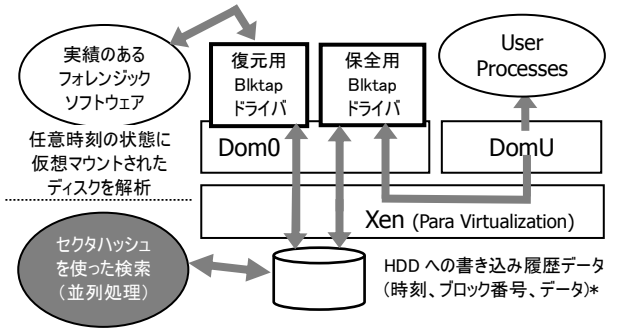

図 1　履歴保全システム

復元用の仮想デバイスドライバを使えば指定した時刻のディスクイメージを読み込み専用でマウントして読み込ませることができるので，実績のある TSK や EnCase のようなフォレンジック・ソフトウェアで解析できる．しかしながら，どの時刻のディスクを復元すればよいのかを探し出す機能や，時系列で問題となるファイルを絞り込んでいくといった機能は，時系列の書き込みデータを解析の対象としていない既存ツールの守備範囲外である．さらに，時系列での書き込みデータはブロック単位であるためファイルシステムの情報を利用できない．そこで，本稿ではセクタハッシュ方式によるファイル検出方式を採用することで，この監視システムから得られたデータから，特定ファイル（漏えいしたファイル等）の書き込み時刻を推定し，その後は既存の実績のあるツールで解析する枠組みを検討する．

我々の履歴保全システムは前述のとおり 4096 バイト単位でブロックデータを時刻情報とともに保存する．しかし，ファイルシステムが 4096 バイトブロックで処理できる機能をもっていても，ハードウェア側の仕様によってデータ境界が 512 バイト毎になってしまうことが多いことが判明した．そのため，本稿ではハッシュ値を 512 バイトのセクタ単位で作成して境界問題を回避することとした．

6. セクタハッシュによる分散並列処理を用いたファイル検出システムの試作

本節では，2 節で示したセクタハッシュ方式を，5 節で示した履歴保全システムから得たデータからのファイル検索に適用した試作について報告する．

6.1 設計

5 節の履歴保全システムで出力した履歴データは（セクタ，セクタのメタデータ）から構成される連続したバイナリデータである．セクタのメタデータは，書き込み時の時刻情報（UNIX の timespec 構造体）と Logical Block Addressing 方式で表現されたセクタ番号から構成される．まず，前処理として全てのデータを読み込み，セクタのハッシュ値を MD5 アルゴリズムで計算して，（i 番目のセクタのハッシュ値，i 番目のセクタのメタデータ）からなる転置インデックスを作成する．転置インデックスの作成は，キーバリューの単純な逐次読み込み処理であるため，実績があり動作も安定している並列分散処理フレームワークの Hadoop を採用する．転置インデックスからの検索については，検索キーにファイルを与えると，ファイルを 512 バイト単位で区切り，512 バイト毎にハッシュ値を計算して，転置インデックスのハッシュ値と照合する方式とした（最後の 512 バイトの末尾が足りない場合はゼロを埋めてハッシュ値を求める）．最後に，適合したハッシュ値のメタデータ（時刻情報とセクタ番号）の一覧を出力する．転置ファイルからの検索は（１）MapReduce で逐次検索アルゴリズムを分散処理，（２）B-tree インデックスを有効にした MySQL を使った探索を 1 台で実行，の二通りで実現した．

6.2 実装

転置インデックスの作成処理は Apache Hadoop 2.2.0 の MapReduce プログラムとして Java 言語で実装した．履歴保全システムの出力データ（つまり転置インデックスのもとになるデータ）は固定長レコードが連続するバイナリデータであるため，MapReduce ではそのまま処理できない．このため，事前処理として全データをバイナリ形式のキーバリューレコード（SequenceFile 形式）に変換してから転置インデックスの作成をおこなった．Map ではバイナリレコードのキーバリューをひとつずつ読み込み，512 バイトのデータを入力として MD5 ハッシュアルゴリズムの出力をキーとして出力し，時刻情報とセクタ番号を一緒にバリューとして出力した．Reduce で受信したキーバリューを転置インデックスとして書き出した．

転置インデックスからの検索は以下の二通りの方法で実装した．（１）MapReduce プログラムとして処理させるため，Map で転置インデックスを一行ずつ読みこみ，逐次探索を n 台のクラスタで並列処理させ，Reduce で結果を出力した．（２）MySQL に転置インデックスを読み込ませ，MySQL に内蔵されている B-tree インデックスで検索をおこなわせた．後者は Java Database Connectivity (JDBC)ライブラリ経由で SQL を呼び出すスタンドアロンの Java プログラムとして実装した．

7. 性能評価

仮想ディスクのイメージを CentOS 6.5 (x64) 上で ext4 ファイルシステムとしてフォーマットし，Govdocs データセット[17] に含まれる 98,562 個のファイル（約 55.2GB）を履歴保全システム[6]を用いて書き込むことで，テスト用の履歴データ（約 58.4GB）を作成した．

Vol.2015-DPS-162 No.40
Vol.2015-CSEC-68 No.40
2015/3/6

表 4 に実験に用いたサーバの仕様を示す．マスタサーバで約 58.4GB の履歴データから SequenceFile 形式ファイル（37.7GB）へ変換するのに 25 分，さらにマスタ 1 台，スレーブ 3 台の Hadoop クラスタを用いて SequenceFile 形式ファイルからセクタハッシュ値をキーとする転置インデックス(4.7GB)を作成するのに 31 分を要した．最後に転置インデックスから Hadoop クラスタ合計 4 台を用いて MapReduce の単純な逐次探索プログラムでファイルを検索した際の時間を計測した（図 2）．検索対象のファイルは Govdocs データセットに含まれる 050434.jpg ファイル（39,004 バイト，セクタハッシュ数 77）とした．検索対象データ（Govdocs のファイル約 55.2GB）を 2 倍，5 倍，10 倍と増加させたときの検索時間の変化を計測した．

MySQL では，前述の SequenceFile 形式ファイル（37.7GB）から，Hadoop クラスタ計 4 台を用いて MapReduce で転置インデックス（8.6GB）を作成した．MySQL へデータをロードして B-tree インデックスを作るのに要した時間は，マスタサーバ 1 台を使った場合で，対象データ 55.2GB（SQL へ入力した転置インデックス 8.6GB）で 27 分，110.4GB（SQL へ入力した転置インデックス 17.2GB）で 60 分，対象データ 276.0GB（SQL へ入力した転置インデックス 43.1GB）で 6 時間 14 分であった．MySQL のインデックスを含めたデータ使用量は，履歴データ 55.2GB（転置インデックス 8.6GB）の場合に 15.4GB，履歴データ 110.4GB（転置インデックス 17.2GB）の場合に 44.6GB，履歴データ 276.0GB（転置インデックス 43.1GB）の場合に 100.3GB に増加した．最後に，マスタサーバ 1 台で動作する MySQL に読み込ませ，B-tree インデックスを有効にした際に 050434.jpg を検索するのに要した時間を図 3 に示す．

8. 考察

性能評価実験から得られた実験データをもとに，市販サーバを使ってどの程度までのデータ量を処理できるかを考察する．まず，単純な逐次探索用の転置インデックスの作成時間は 55.2GB 程度のデータに対して SequenceFile 形式への変換も含めて 1 時間程度を要した．転置インデックスの作成は全てのセクタを読み込み，ハッシュ値を計算して出力する O(n) の処理であるため，データ量に比例した時間がかかるはずである．履歴保全システムと連動させることを想定とすると，マスタサーバでバッチ処理により一定期間ごとに履歴データから SequenceFile 変換をおこない，定期的に分散ファイルシステム上の SequenceFile を MapReduce プログラムで転置インデックス化する設計が考えられる．

続いて転置ファイルからの検索時間であるが，MapReduce でナイーブな逐次探索アルゴリズムを実行した場合，55.2GB からの検索であっても 66 秒を要した．O(n) の探索アルゴリズムを改良しない限り，Hadoop クラスタの台数を線形的に増加させるだけでは実用的な性能は得られないことが確認できた．改善策として後述する MySQL のように B tree やハッシュ探索のインデックスをあらかじめ作成しておき，各 Map または Reduce 処理で B tree やハッシュ探索のインデックスを使いながら並列処理で探索する方法が考えられる．どちらにしても MapReduce はキーバリューを 1 行ずつ読み込む O(n) の転置ファイル作成のような処理に適しており，Hadoop 分散ファイルシステムの Locality の利点を最大限に活かしてノード間のデータのコピーを減らすためには，インデックスの領域を分割して分散処理させなければいけないと考えられる．

MySQL を使った転置ファイルからの検索時間については 1 台のサーバを使っているにも関わらず検索対象データ 55.2GB からの探索には 342.6 [ms]しか要しなかった．実験結果から B tree を有効にした MySQL の探索時間はおおよ

表 4　性能評価に使ったサーバの仕様

	マスタサーバ	スレーブサーバ
CPU (コア数)	Xeon E5-2630v3 x2 (16 core)	Core i7 5820K (6 core)
キャッシュ	20MiB	15MiB
メモリ	64GiB	48GiB
NIC	10GBASE-T	10GBASE-T
台数	1	3

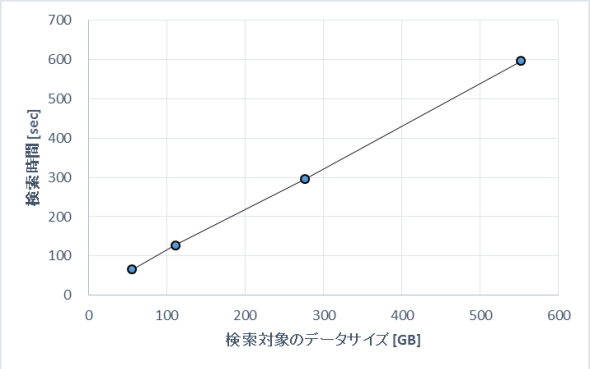

図 2　Hadoop クラスタによる逐次探索に要した時間

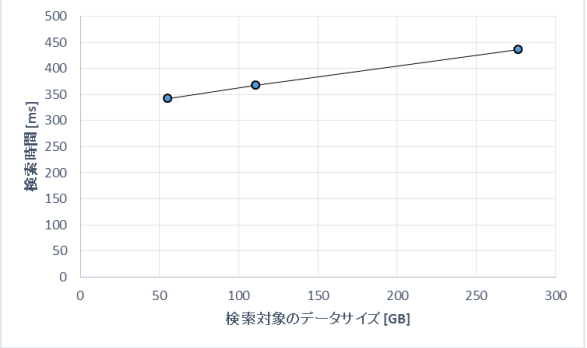

図 3　MySQL の B tree インデックス検索に要した時間

そ log(n)で増加したことが確認できた．B tree アルゴリズムはインデックスをページ単位で必要な部分だけを読み書きしてディスクアクセスを最小化する．このため，主メモリの容量の違いを単純に無視すると，実験と同じファイルを検索する場合，仮に 6.4TB からデータを探索すると仮定した場合で約 1.1 [s]，640 TB からデータを探索することを仮定した場合で約 1.8 [s] で検索が完了すると推定できた．しかしながら，大規模なデータからのセクタハッシュを用いた検索には以下の２つの課題がある．まず，インデックスを作成するのに要する時間はサイズが大きくなるごとに増加した．例えば対象データ 276.0GB の時にインデックスサイズが 100.3 GB となって物理メモリの 64GB を超えてしまった．これにより極度にインデックス作成の性能が低下した．この結果から，常に物理メモリに収まる程度のインデックスサイズになるように，入力データ量を調整しなければ最良の性能は得られないことが分かった．もうひとつの問題はインデックスのサイズである．履歴データ 276.0GB に対して B tree 有効時のデータベースの使用量が 100.3GB であったので，元のサイズの 1/3 程度の容量を追加で占有することになった．インデックス作成は物理メモリ量とインデックスの作成時間，元データに対するサイズのバランスをとって検討する必要があることが分かった．

本稿で示した実験結果から，セクタハッシュ方式で数十テラバイト規模のデータを処理するには，少なくとも log(n) の探索アルゴリズムが必要であり，事前のインデックス作成が必要であることを確認できた．MapReduce は O(n)が必要となる転置インデックスの作成には適しているが，B tree などのインデックスを使わない限り，50GB 程度の小規模データでさえも実用的な探索時間は得られないことが分かった．MySQL と B tree インデックスを使った場合にはインデックスの作成に時間がかかるが，実用的な速度を得ることができた．Garfinkel らが研究[9]で指摘しているようにセクタハッシュ方式の場合はキーとなるハッシュ値がランダムにばらつくため，Prefix routing 方式[18]であらかじめキー範囲毎に n 台のサーバに分散させて，インデックス作成と検索を並列処理でおこなわせることで，更に大規模なデータを処理できる可能性があると考えられる．

9. おわりに

本稿ではセクタハッシュ方式について紹介し，関連研究を示した．さらに本稿執筆時点で広く使われている 3 種類の OS についてセクタハッシュの有効性に関する実験をおこない，Windows 8.1 と MacOS 10.9 では初期状態のシステムファイルに対してもセクタハッシュでファイル特定をおこなう方法が有効であることを示した．さらに，Windows で採用されている NTFS では 720 バイト以下のファイルがセクタ境界に並ばずに検出できない課題があることが実験により確認できた．最後にセクタハッシュ方式を我々の先行研究で開発した履歴保全システムに適用し，並列分散処理で転置ファイルを作成し，ファイルを高速に検索する手法について検討した結果を報告した．

謝辞 本研究は JSPS 科研費 26330168 の助成を受けたものです．

参考文献

1) 辻井重男監修, 佐々木良一他著: デジタル・フォレンジック事典,日科技連出版社 (2006)
2) Reith, Mark, Clint Carr, and Gregg Gunsch: An examination of digital forensic models., International Journal of Digital Evidence 1.3 (2002)
3) Simson L. Garfinkel: Digital forensics research: The next 10 years, Digital Investigation, Volume 7, Supplement, August 2010, Pages S64-S73 (2010)
4) Roussev, Vassil, and Golden G. Richard III: Breaking the performance wall: The case for distributed digital forensics, Proceedings of the 2004 digital forensics research workshop. Vol. 94. (2004)
5) Lodovico Marziale, Golden G. Richard III, Vassil Roussev: Massive threading: Using GPUs to increase the performance of digital forensics tools, Digital Investigation, Volume 4, Pages 73-81 (2007)
6) 小川拡, 平野学: 仮想計算機モニタを利用したコンピュータフォレンジックスのための補助記憶装置のデータの保全と回復のシステム, 研究報告コンピュータセキュリティ, 2013-CSEC-60(1), 1-6 (2013)
7) 手塚伸, 宇田隆哉, 岡田謙一: 監査と削除保証を考慮したクラウド仮想ファイルシステム, 情報処理学会論文誌 55.2, pp. 695-706 (2014)
8) Carrier, Brian, and Eugene H. Spafford: An event-based digital forensic investigation framework, Digital forensic research workshop (2004)
9) Simson Garfinkel, Alex Nelson, Douglas White, Vassil Roussev: Using purpose-built functions and block hashes to enable small block and sub-file forensics, Digital Investigation, Volume 7, Supplement, pp. S13-S23 (2010)
10) Joel Young, Kristina Foster, Simson Garfinkel, Kevin Fairbanks: Distinct Sector Hashes for Target File Detection, IEEE Computer, vol. 45, no. 12, pp. 28-35 (2012)
11) Farrell, P.; Garfinkel, S.L.; White, D.: Practical Applications of Bloom Filters to the NIST RDS and Hard Drive Triage, Computer Security Applications Conference, 2008. ACSAC 2008. Annual , vol., no., pp.13,22, 8-12 (2008)
12) Burton H. Bloom: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13, pp. 422-426 (1970)
13) Oracle, How MySQL Uses Indexes, http://dev.mysql.com/doc/refman/5.7/en/mysql-indexes.html (2015 年 2 月 8 日閲覧)
14) 吉田光輝, 高瀬誉, 平野学: 仮想計算機モニタを用いた外部記憶装置の監視分析システム～ブロックハッシュを用いた分析機能の試作と評価, 情報処理学会第 77 回全国大会 (to appear) (2015)
15) David Solomon, Mark Russinovich 著: インサイド Windows 第 4 版下巻, 日経 BP ソフトプレス (2005)
16) Microsoft, How NTFS Works, https://technet.microsoft.com/, (2003) (2015 年 2 月 6 日閲覧)
17) Garfinkel, S., Paul Farrell, Vassil Roussev, George Dinolt: Bringing science to digital forensics with standardized forensic corpora DFRWS (2009)
18) Bakker, Erwin M., Jan van Leeuwen, and Richard B. Tan: Prefix routing schemes in dynamic networks, Computer Networks and ISDN Systems 26.4, pp. 403-421 (1993)