Android端末のメモリ暗号化によるコールドブート攻撃対策

福田直人[1]　光来健一[1]

概要：スマートフォンやタブレットなどの Android 端末は盗難にあうリスクが高いため，ディスク暗号化により端末が盗まれた場合の情報漏洩を防いでいる．しかし，近年，コールドブート攻撃と呼ばれる攻撃手法が報告され，メモリからの情報漏洩が問題となってきている．Android OS ではディスク上のデータをメモリ上にページキャッシュとして保持しているため，コールドブート攻撃が行われるとディスクを暗号化していてもページキャッシュ上のデータを盗み見られてしまう．本稿では，メモリ上のページキャッシュを暗号化することで Android 端末の盗難時における情報漏洩を防ぐためのシステム *Cache-Crypt* を提案する．Cache-Crypt はディスク上のファイルを読み込む際などにページキャッシュの内容を暗号化する．アプリがファイルにアクセスする時だけ OS がページキャッシュを復号し，アクセスが終わったら再び暗号化する．Cache-Crypt では，暗号化のオーバヘッドを削減するためにこの再暗号化を遅延する．Cache-Crypt を用いることで情報漏洩の対象をアクセス中のページキャッシュのみに限定することができる．我々は Android OS に Cache-Crypt を実装し，オーバヘッドが許容範囲内であることを確認した．

キーワード：Android，盗難対策，コールドブート攻撃，メモリ暗号化

A Countermeasure against Cold-boot Attacks Using Memory Encryption of Android Devices

Naoto Fukuda[1]　Kenichi Kourai[1]

Abstract: Since Android devices such as smartphones and tablets are at high risk of theft, they prevent information leakage from disks by disk encryption in case of device theft. However, in recent years, cold boot attacks have been reported and information leakage from memory is being a major problem. Since the Android operating system maintains data on disks in memory as the page cache, data on the page cache can be stolen by cold boot attacks even if disks are encrypted. In this paper, we propose a system called *Cache-Crypt* for preventing information leakage in case of device theft by encrypting the page cache in memory. Cache-Crypt encrypts the page cache, e.g., when the operating system reads files from disks. The operating system decrypts the page cache only when apps access files and encrypts it again when the access is done. To reduce the overhead of encryption, Cache-Crypt delays this re-encryption. Cache-Crypt can limit the range of information leakage only to the page cache being accessed. We have implemented Cache-Crypt in the Android operating system and confirmed that the overhead is allowable.

Keywords: Android, anti-theft, cold boot attack, memory encryption

1. はじめに

近年，スマートフォンやタブレットなどの Android 端末が急速に普及している．Android 端末のアクティベート数は 2013 年で 10 億台以上になり，現在も増加し続けている．特に，新興国向けに安価な Android 端末が数多く提供されているため，普及率も他の OS を搭載した端末に比べて非常に高い．Android 端末の普及や多機能化に伴って，Android 向けに数多くのアプリが提供されるようになっている．そのため，Android 端末は従来の携帯電話より多くの情報を保持しており，端末が盗難にあった際のリスクは高まっている．例えば Android 端末には，クレジットカー

[1] 九州工業大学
Kyushu Institute of Technology

ド番号，ID やパスワード，その他の多くの個人情報が格納されていることが多い．加えて，Android 端末はノート PC より小型軽量であるため，盗難にあうリスクも高い．盗難対策として，Android ではディスク暗号化の機能が提供されており，端末が盗まれた場合でも画面ロックを解除されなければディスクからの情報漏洩を防ぐことができる．

しかし，近年，コールドブート攻撃 [3] と呼ばれる攻撃手法が報告され，メモリからの情報漏洩が問題となってきている．コールドブート攻撃は，メモリを冷却した状態で端末をリセットし，別の OS を起動してメモリ上のデータを盗む攻撃である．Android 端末においてもコールドブート攻撃が報告されている [13]．Android OS では，ディスク上のデータをページキャッシュとしてメモリ上に保持することで，ファイルアクセスを高速化している．そのため，端末を盗まれてコールドブート攻撃が行われると，ページキャッシュ上のデータを盗み見られてしまう．その結果，ディスクが暗号化されていたとしてもディスク上のデータの一部がページキャッシュを通して漏洩してしまう危険がある．

本稿では，メモリ上のページキャッシュを暗号化することで Android 端末の盗難時における情報漏洩を防ぐためのシステム *Cache-Crypt* を提案する．Cache-Crypt はディスク上のファイルをメモリに読み込む際に暗号化されたデータをページキャッシュに格納する．アプリが OS 経由でファイルにアクセスする時だけページキャッシュを復号し，アクセスが終わったら再び暗号化する．Cache-Crypt では，端末の盗難時には画面がロックされていることを想定して，画面ロックが解除されている間だけ再暗号化を遅延することにより安全に暗号化のオーバヘッドを削減する．また，ページキャッシュの暗号化・復号化に用いる暗号鍵はメモリ上に置かず，既存研究 [8] を用いてデバック用の CPU レジスタ上に置くことで保護する．Cache-Crypt を用いることで，情報漏洩の対象をアクセス中のページキャッシュのみに限定することができる．

我々は Cache-Crypt を Android OS に実装し，ページキャッシュに対して AES のブロック単位で暗号化されているかどうかを管理するようにした．アプリが read システムコールを発行した時に，Cache-Crypt はページキャッシュ上のデータを復号してアプリに返す．write システムコールを発行した時には，Cache-Crypt は必要に応じてページキャッシュを復号してからデータを書き込む．最後にアクセスしてから一定時間が経過したらページキャッシュを再暗号化する．一方，アプリが mmap システムコールを発行してファイルをメモリマップした時には，Cache-Crypt は最初のアクセス時にページキャッシュのデータを復号する．Nexus 7 上でベンチマークを動作させて Cache-Crypt を用いた場合のファイルアクセス性能を測定し，オーバヘッドが許容範囲内であることを確認した．

以下，2 章では，ページキャッシュからの情報漏洩の危険性について述べる．3 章ではこの問題を解決する Cache-Crypt について述べ，4 章でその実装の詳細について述べる．5 章で Cache-Crypt を用いて行った実験について述べる．6 章で関連研究について述べ，7 章で本稿をまとめる．

2. ページキャッシュからの情報漏洩

Android では端末の盗難対策としてフルディスク暗号化が提供されている．フルディスク暗号化とは，ディスクのパーティション全体を暗号化することでデータを保護する仕組みである．Android ではアプリのデータが置かれたパーティションのみが暗号化される．Android のフルディスク暗号化には dm-crypt が用いられており，ディスクからの読み込み時にデータの復号が行われ，ディスクへの書き込み時にデータの暗号化が行われる．暗号化されたパーティションを復号するには，OS 起動時に暗証番号（PIN）を入力する必要がある．OS は入力された PIN から暗号鍵を生成し，dm-crypt による暗号化・復号化に用いる．そのため，PIN を知られない限り，ディスクを取り外して別の端末に取り付けたとしてもディスク上の情報を盗み見られることはない．

しかし，近年，メモリから比較的容易に情報を盗み出せるコールドブート攻撃 [3] が報告されている．コールドブート攻撃は，メモリを冷却した状態で端末を強制リセットし，攻撃者自身の OS を用いて端末を起動してメモリ上に残っているデータを盗み見る攻撃である．揮発性メモリ上のデータは通常，端末のリセットにより電源供給が途絶えている間に次第に破壊されていくが，データが完全に消えるまでには少し時間がかかる．メモリを冷却することでメモリ上のデータが破壊されるのを遅らせることができる．この攻撃手法は端末を強制リセットするため，通常のシャットダウンとは異なり，OS がメモリ上の機密情報を消去することはできない．

Android 端末においてもコールドブート攻撃が報告されている [13]．報告された攻撃手法では，端末全体を冷却した状態でバッテリを抜き差しして端末のリセットを行う．次に，USB 経由で PC から攻撃者のリカバリイメージをインストールし，そのリカバリイメージから起動する．リカバリイメージをインストールする際にブートローダをアンロックする必要がある場合でも，ディスク上のデータは消去されるが，メモリ上のデータは消去されない．リカバリイメージの OS はメモリ上のデータをダンプし，ディスク暗号化の鍵などを取り出すことができる．

近年，盛んに開発されている STT-MRAM などの次世代不揮発性メモリが実用化されると，コールドブート攻撃をさらに容易に行うことができるようになる．不揮発性メモリは従来の揮発性メモリと異なり，電源供給が途絶えた場合でも，メモリ上のデータは消えずにそのまま残る．不揮

発性メモリを用いると，システムがクラッシュした時のメモリの状態を容易に保存できるのでクラッシュの原因の調査などが容易になる．また，電源を切った状態からの起動を行った場合でも，OS やアプリケーションが前回使っていたデータがメモリ上に保存されているため，起動が速くなるなどの利点が多い．しかし，端末が盗難にあった場合に強制リセットが行われてもメモリ上の情報が壊れることがなく，コールドブート攻撃に成功される危険性が高まる．

Android OS は他の多くの OS と同様に，ファイルアクセスの高速化のためにディスク上のデータをメモリ上にキャッシュとして保持している．特に，ファイルの内容はページキャッシュと呼ばれるキャッシュに保持される．アプリがファイルを読み込む際には，まず，OS がディスク上のデータをメモリ上のページキャッシュに読み込む．そして，ページキャッシュ上のデータがアプリに返される．アプリが同じファイルにアクセスする時には，ディスクへのアクセスを行わず，ページキャッシュ上のデータを即座に返すことができる．また，アプリがディスクにデータを書き込む際には，まず，ページキャッシュに書き込まれてからディスクに書き戻される．

そのため，Android 端末がコールドブート攻撃を受けると，メモリ上のページキャッシュを盗み見られることになる．ページキャッシュ上にはディスクと同じデータ，もしくは，ディスクより新しいデータが置かれている．その結果，フルディスク暗号化を行っていたとしてもディスクのデータの一部は漏洩してしまう．Android OS では空きメモリの多くはページキャッシュとして使われ，また，Android 端末のメモリ容量も増大してきているため，漏洩するデータ量も増大する傾向にある．

3. Cache-Crypt

3.1 脅威モデル

Cache-Crypt では端末を盗まれ，コールドブート攻撃によりページキャッシュや暗号鍵を盗み見られる攻撃を想定する．ディスクからの情報漏洩はフルディスク暗号化で防ぎ，アプリのメモリからの情報漏洩は既存研究の Cryptkeeper [9] で防ぐ．Cache-Crypt では端末の盗難時には画面がロックされており，PIN を知られて画面ロックを解除されないことを仮定する．また，端末にインストールされた不正なアプリ経由で復号後のディスク上のデータを取得されないものとする．

3.2 Cache-Crypt

Cache-Crypt は OS がディスク上のファイルをメモリに読み込む際などにページキャッシュの内容を暗号化する．これを**暗号化ページキャッシュ**と呼ぶ．アプリが OS 経由でファイルにアクセスする時だけ暗号化ページキャッシュの必要な領域を復号し，アクセスが終わったら再び暗号化する．Cache-Crypt を用いることでコールドブート攻撃による情報漏洩の対象を，端末がリセットされた瞬間にアプリがアクセス中であったページキャッシュのみに限定することができる．その他のページキャッシュについては暗号化されているため，メモリ全体を盗み見られても情報は漏洩しない．

アプリが OS のシステムコールを用いてファイルからデータを読み込む際には，Cache-Crypt が暗号化ページキャッシュを復号し，そのデータをアプリのバッファにコピーする．その後で，ページキャッシュの再暗号化を行う．アプリはバッファに格納された復号済みのデータを Cache-Crypt による暗号化を意識せずに扱うことができる．読み込もうとしたファイルに対応するページキャッシュが存在しない時は，暗号化ディスクからページキャッシュ上にデータを読み込む．この時に，フルディスク暗号化と Cache-Crypt の暗号方式および暗号鍵を共通にしておくことにより，暗号化ディスク上のデータをそのままページキャッシュに読み込むことができる．これにより，暗号化ディスク上のデータを一旦復号して，再度，Cache-Crypt 用に暗号化し直す必要がなくなり，性能を向上させることができる．また，一時的に復号されたデータがメモリ上に置かれることもなくなり，セキュリティを向上させることができる．

アプリがファイルにデータを書き込む際には，Cache-Crypt が必要に応じて暗号化ページキャッシュを復号し，アプリのバッファのデータをページキャッシュに書き込む．その後で，ページキャッシュに書き込まれたデータを暗号化する．アプリは Cache-Crypt による暗号化を意識せずにバッファのデータを書き込むことができる．既存のファイルの一部を書き換える場合は，まず，暗号化ディスクから暗号化ページキャッシュにデータを読み込んでから上書きを行う．書き換えられたページキャッシュ上のデータは，既存の OS の機構により，適切なタイミングで暗号化ディスクに書き戻される．この際に，フルディスク暗号化と Cache-Crypt を連携させることにより，暗号化ページキャッシュ上のデータをそのままディスクに書き込むことができる．

ページキャッシュにアクセスした後の再暗号化のオーバヘッドを削減するために，Cache-Crypt は暗号化の遅延を行い，一旦，復号したページキャッシュはしばらく復号したままにする．そして，一定時間アクセスがなければページキャッシュを再び暗号化する．これにより，ファイルの読み書きの際には必要に応じてページキャッシュの復号のみを行えばよくなり，Cache-Crypt の性能を向上させることができる．遅延時間を長くすればするほど，性能を向上させられる可能性があるが，暗号化されていないページキャッシュが増えるため，安全性とのトレードオフになる．ただし，端末を盗まれてコールドブート攻撃を行われるま

でには一定の時間がかかるため，それまでの間に暗号化が完了すれば安全性は低下しない．

アプリがファイルをメモリマップしてアクセスする際には，Cache-Crypt がそのファイルに対応するページキャッシュを復号する．ファイルをメモリマップすると，ページキャッシュがアプリのアドレス空間にマップされ，アプリが OS を介さずに直接ページキャッシュにアクセスできるようになる．そのため，アプリがファイルをアンマップするまではページキャッシュを復号したままにする．メモリマップされたファイルが書き換えられた場合，Cache-Crypt はページキャッシュ上のデータの暗号化を行った上で暗号化ディスクに書き戻す．ファイルのアンマップ時には再びページキャッシュを暗号化する．

ページキャッシュの暗号化を行うための暗号鍵は既存研究の ARMORED [8] を用いて保護する．暗号鍵は通常，メモリ上に置かれるが，コールドブート攻撃が行われるとメモリ上の暗号鍵も盗まれてしまう．その結果，暗号化したページキャッシュも復号されてしまうことになる．ARMORED では，CPU レジスタ上に暗号鍵を保持することによって，コールドブート攻撃を受けても暗号鍵が漏洩するのを防ぐことができる．さらに，CPU の SIMD 拡張命令セットが提供するレジスタをメモリの代わりに用い，暗号処理の途中結果が漏洩することも防ぐことができる．

4. 実装

我々は Cache-Crypt を Android カーネル 3.4 に実装した．ファイルシステムに依存しないようにするため，Cache-Crypt を VFS 層に実装した．これにより，Android 標準の ext4 以外のファイルシステムもサポートする．現在の実装では，フルディスク暗号化の dm-crypt との連携はまだ行っておらず，個別に暗号化・復号化を行っている．

4.1 暗号化ページキャッシュの管理

Cache-Crypt ではページ構造体に暗号化フラグを追加することで，ページキャッシュの状態を管理する．ページ構造体はページキャッシュとして使われているメモリページを含め，すべてのページを管理するためのカーネルデータ構造である．Cache-Crypt は暗号化フラグを用いて，ページキャッシュとして使われているか，ページキャッシュが暗号化されているか，および，プロセスにメモリマップされているかを判断する．暗号化フラグの値は表 1 のいずれかである．

暗号化フラグが PAGE_NON であるページをページキャッシュに追加する際に，その暗号化フラグを PAGE_DEC に変更する．逆に，ページキャッシュからページを削除する際には，その暗号化フラグが PAGE_DEC，PAGE_ENC，PAGE_MAP のいずれであっても PAGE_NON に変更する．

表 1　ページキャッシュの状態を表すための暗号化フラグ

暗号化フラグ	ページキャッシュの状態
PAGE_NON	ページキャッシュではない
PAGE_ENC	ページ全体が暗号化されている
PAGE_DEC	一部または全体が復号されている
PAGE_MAP	メモリマップされている

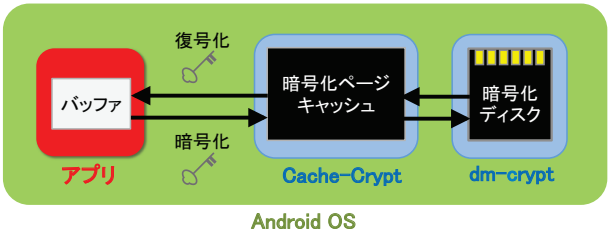

図 1　ファイルの読み書きの流れ

AES のブロック単位でのきめ細かい暗号化・復号化を行えるようにするために，Cache-Crypt は**暗号化ビットマップ**を用いて各ブロックが暗号化されているかどうかを管理する．ビットが 1 の時にはブロックが暗号化されており，ビットが 0 の時には復号されていることを表す．AES のブロック長は 128 ビットであるため，4KB のページには 256 個のブロックが含まれる．Cache-Crypt は，暗号化・復号化を行う際に暗号化ビットマップを確認して，ブロック単位で暗号化・復号化を行う．これにより，ページキャッシュの中のほぼ必要なデータのみを暗号化・復号化できるようになり．オーバヘッドを削減することができる．

4.2 ファイルの読み込み

図 1 に示すように，アプリ（プロセス）が read システムコール等を発行してファイルを読み込む時に，必要に応じて対応するページキャッシュを復号する．この処理はファイルシステムに共通の file_read_actor 関数で行われる．指定されたファイルオフセットから指定されたサイズのデータを含むページキャッシュ中の AES ブロックに対して，Cache-Crypt はまず，暗号化ビットマップを調べる．暗号化ビットマップのビットが 1 であれば，対応するブロックを復号し，そのビットをクリアする．そして，そのページの暗号化フラグを PAGE_DEC に変更する．

読み込むファイルに対応するページキャッシュが存在しない場合は，ディスクからページキャッシュに読み込まれるデータを暗号化する．この処理は読み込み完了時に必ず呼ばれる bio_endio 関数にて行う．読み込み先のページキャッシュは bio 構造体に格納されており，読み込んだデータを暗号化して，対応する暗号化ビットマップのビットを 1 にする．ページ全体が暗号化された場合には暗号化フラグを PAGE_ENC に変更し，そうでない場合には PAGE_DEC に変更する．dm-crypt と連携して暗号化されたデータを直接ページキャッシュに読み込むことができれ

ば，この暗号化は不要となる．

ページキャッシュからプロセスのバッファへのデータコピーが終わるとページキャッシュを再び暗号化する．その際に，暗号化ビットマップのビットを 1 にし，ページ全体が暗号化された場合は暗号化フラグを PAGE_ENC に変更する．ただし，この暗号化は，暗号化フラグが PAGE_MAP の場合には行わない．後述するように，アクセスしたファイルがプロセスにメモリマップされている場合には，ページを復号化したままにしておく必要があるためである．また，ページキャッシュがファイルシステムのメタデータを保持している場合にはページの暗号化を行わない．メタデータかどうかは，ページキャッシュに対応するファイルの inode 番号が 0 かどうかで判断する．メタデータはカーネル内でも頻繁に参照されるため，暗号化するとカーネル内での処理を行うたびに復号化・暗号化を繰り返すことになる．その上，メタデータに機密情報が含まれることは少ない．

4.3 ファイルへの書き込み

プロセスが write システムコール等を発行してファイルに書き込む時は，必要に応じてページキャッシュのデータを復号する．この処理は generic_perform_write 関数で行われる．指定されたファイルオフセットが AES ブロックの先頭でない場合には，そのブロックを復号してから部分的な上書きを行う．また，ファイルオフセットと指定されたサイズの和が AES ブロックの先頭でない場合にも，最後のブロックを復号してから部分的な上書きを行う．その他のブロックについては復号を行わずに，プロセスから渡されたバッファのデータで上書きすることができる．

書き込み時にページキャッシュが存在しない場合，ディスク上に対応するファイルブロックがあれば 4.2 節と同様にしてファイルを読み込む．この際に，Cache-Crypt がページキャッシュを暗号化し，必要に応じて復号しながら書き込みを行う．一方，ディスク上に読み込むべきデータがなければ，確保されたページキャッシュを PAGE_DEC の状態にし，直接書き込みを行う．

ページキャッシュにデータを書き込んだ後，Cache-Crypt はページキャッシュを暗号化し，対応する暗号化ビットマップおよび暗号化フラグを変更する．この暗号化はファイルの読み込みと同様に，ページの暗号化フラグが PAGE_MAP ではない，かつ，ページキャッシュがメタデータを保持していないという条件の下で行う．

4.4 ディスクへの書き戻し

書き換えられたページキャッシュはディスクに書き戻される際に Cache-Crypt が復号を行う．まず，ファイルの最後のブロックを含むページキャッシュの場合には，block_write_full_page_endio 関数においてファイルサイズを超える部分を復号してから 0 で埋める．次に，submit_bio 関数においてディスクに書き戻されるブロックを復号する．そして，ディスクへの書き戻し時に呼ばれる bio_endio 関数において，ページキャッシュを再び暗号化する．現在の実装では，Cache-Crypt によって復号されたデータが dm-crypt によって再び暗号化される．dm-crypt と連携することで暗号化ページキャッシュをそのままディスクに書き戻すことが可能になる．

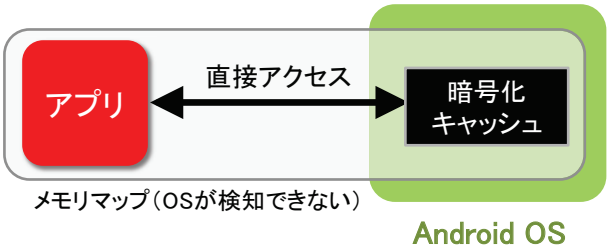

図 2 メモリマップされたファイルのアクセス

4.5 ファイルのメモリマップ

プロセスが mmap システムコールを発行してファイルをメモリマップした場合，ページキャッシュを暗号化していると正常な動作が行えなくなる．ファイルをメモリマップすると，ディスク上のファイルがページキャッシュに読み込まれ，そのページキャッシュがプロセスのアドレス空間に直接マップされる．これにより，プロセスはメモリへのアクセスと同様にしてファイルにアクセスすることができ，高速なファイルアクセスやプロセス間でのデータ共有などが行える．しかし，メモリマップされたページキャッシュは図 2 のように OS を介さずに直接アクセスされるため，アクセス時だけ復号するのは難しい．

そこで，Cache-Crypt では，メモリマップされたページキャッシュにプロセスが初めてアクセスした時にページキャッシュを復号する．この処理は filemap_fault 関数で行われる．mmap システムコールを実行した時点ではページキャッシュはまだマップされておらず，プロセスが最初にアクセスした時にページフォールトが発生する．その際に，まだページキャッシュが存在しなければディスクからファイルブロックを読み込む．そして，ページキャッシュをプロセスのアドレス空間にマップする．ページフォールトの際にページの暗号化ビットマップが 1 のブロックを復号し，暗号化フラグを PAGE_MAP にする．このページキャッシュはメモリマップが行われている間は復号されたままにする．

ファイルがアンマップされた時にページキャッシュを再び暗号化し，暗号化フラグを PAGE_ENC にする．ただし，同一のファイルが複数のプロセスからマップされる可能性があるため，ページのマップカウントを調べて，どのプロセスにもマップされていない場合にだけ暗号化を行う．

4.6 暗号化の遅延

暗号化の遅延を行う際には，一定時間後に暗号化を行うページキャッシュを**遅延リスト**の末尾に追加する．遅延リストには再暗号化を行うまでの時間が短い順にページキャッシュがつながれている．追加するページキャッシュが既に遅延リストにつながれている場合には，遅延リストの末尾に移動する．再暗号化を行う時刻はページ構造体に格納される．ページキャッシュを遅延リストに追加した後，遅延リストの先頭のページキャッシュを再暗号化する時刻をタイマに設定し直す．

タイマに設定した時刻になったらタイムアウト関数が呼ばれ暗号化の処理を開始する．まず，暗号化を行う時刻になったページキャッシュを遅延リストから取り出し，**暗号処理リスト**に追加する．そして，実際にページキャッシュの暗号化を行う**暗号処理スレッド**を起こす．これは，タイムアウト関数において時間のかかる処理を行うべきではないためである．暗号処理スレッドでは，暗号処理リストからページキャッシュを取り出し，暗号化を行う．

4.7 tmpfs

tmpfs は起動時に用いられる initramfs やルートファイルシステムなどで用いられているメモリベースのファイルシステムである．ファイルデータをページキャッシュに保持し続けられるようにするために，tmpfs では独自のページキャッシュ管理を行う．そのため，Cache-Crypt では新しいページが tmpfs のページキャッシュに追加される時にページの暗号化フラグを PAGE_DEC に変更する．一方，tmpfs のページキャッシュからページを削除する時に暗号化フラグを PAGE_NON に変更する．

5. 実験

Android 端末の Nexus 7 (2013) を用いて実験を行った．Android のカーネルとして，Android 4.4.2 の標準カーネルである Linux カーネル 3.4 に Cache-Crypt を実装したものを用いた．

5.1 オーバーヘッド

Benchmark というベンチマークアプリを用いて、ファイルの読み書きの性能を測定した。測定対象として、Android の標準カーネルおよび、暗号化の遅延時間を様々に変化させた Cache-Crypt を用いた。実験結果を図 3 に示す。読み書きともに、暗号化の遅延を行わなかった場合には非常に性能が悪いことが分かる。読み込みについては、遅延時間を長くすると段階的に性能が改善し、1000 ミリ秒以上ではほぼ一定の性能となった。この時の性能は標準カーネルの 58%程度であった。一方、書き込みについては、暗号化の遅延を少しでも行うと性能が改善し、遅延時間を長くしてもそれ以上の改善は見られなかった。この時の性能は標準カーネルの 88%程度であった。

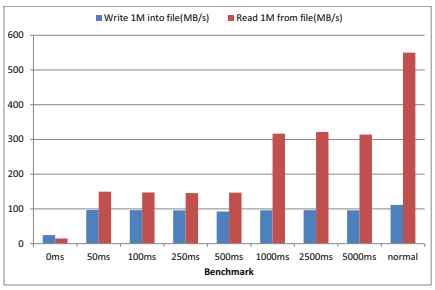

図 3 ファイル読み書きの性能

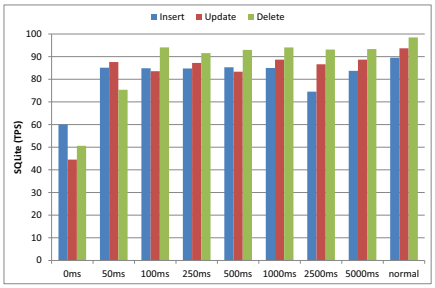

図 4 SQLite の性能

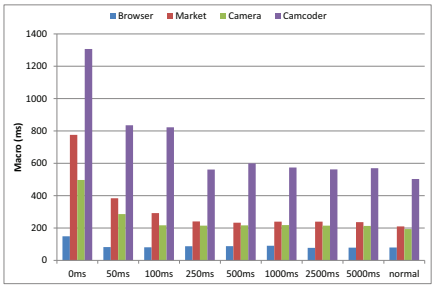

図 5 実アプリの性能

次に、AndroBench というベンチマークアプリを用いて、SQLite の性能測定および実アプリに近い環境での性能測定を行った。実験結果を図 4 と図 5 に示す。SQLite の性能については、少しでも暗号化の遅延を行えば、標準カーネルと Cache-Crypt でほぼ差がないことが分かった。実アプリについても、250 ミリ秒以上の遅延時間にすれば Cache-Crypt による性能低下は小さいことが分かった。

5.2 CPU 使用率

top コマンドを用いて，AndroBench を実行した時の CPU 使用率を測定した．測定対象として，標準カーネル，暗号化の遅延なしの Cache-Crypt，遅延時間を 1 秒とした Cache-Crypt を用いた．実験結果は図 6 のようになった．ccrypt は暗号処理スレッドによって使われた CPU を示し

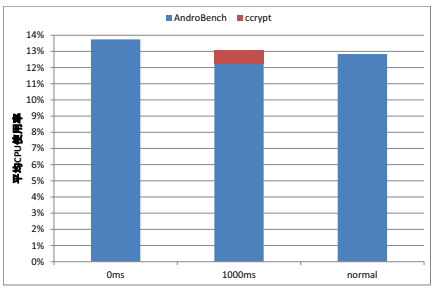

図 6　CPU 使用率

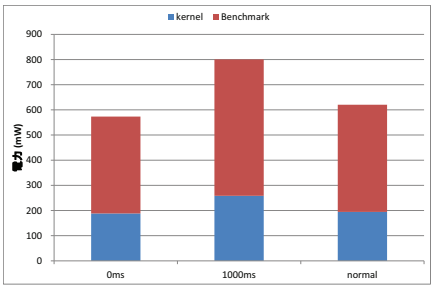

図 7　消費電力

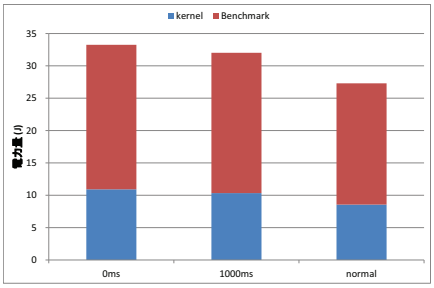

図 8　電力量

ている．この結果より，平均の CPU 使用率には大きな差がないことが分かる．

5.3　消費電力

PowerTutor という消費電力測定アプリを用いて、Benchmark 実行時の消費電力を測定した。測定対象には、標準カーネルおよび Cache-Crypt（暗号化の遅延なしと 1 秒の場合）を用いた。PowerTutor の制限により、カーネル内の暗号処理スレッドだけの消費電力を測定できなかったため、カーネル全体の消費電力を測定した。実験結果を図 7 に示す。この結果より、遅延を行った場合の消費電力がかなり大きくなることが分かった。この原因は現在調査中である。また、この結果から計算した電力量を図 8 に示す。Cache-Crypt では暗号化の遅延を行った場合でも、電力量が少し増大していることが分かる。

6.　関連研究

暗号化ファイルシステムはディスクからの読み込み時 に暗号化されたファイルを復号し，ディスクへの書き込 み時にファイルを暗号化する．そのため，一般的にページキャッシュは暗号化されない．

eCryptfs [6] は既 存のファイルシステムの上に重ねて利用する暗号化ファイ ルシステムであり，既存のファイルシステムが用いるペー ジキャッシュは暗号化される．しかし，eCryptfs 自体が用いる ページキャッシュは暗号化されない．

EncFS [7] ではファ イルをオープンしている間だけ独自のキャッシュが作られ るが，それは暗号化されない．

TransCrypt [11] では Cache-Crypt と同様に，ページキャッシュとプロセスのバッファ間でデータをコピーする時にページキャッシュの暗号化・復号化を行うことができる．しかし，メモリマップには対応しておらず，オーバヘッドの削減も行われていない．また，専用ファイルシステムの機能の一部であるため，ext4 を用いる Android OS では利用することができない．

ZIA [1] はハードウェアトークンが近くにある時だけファイルを復号するファイルシステムである．ZIA ではファイルを暗号化するのに用いた鍵をトークンと通信することで復号し，復号した鍵を用いてファイルを復号する．端末が盗まれるなどしてトークンが近くにない状態になると，ZIA は復号された暗号鍵を破棄し，ページキャッシュを暗号化する．しかし，トークンを持ったユーザが端末から離れている間はメール受信などファイルシステムを使う処理をバックグラウンドで行うことができない．また，端末とトークンの両方が盗まれた場合，ZIA はファイルを保護できない．

Keypad [4] は盗まれやすい端末のためのファイルシステムである．Keypad では，端末が盗まれた後でアクセスされたファイルが分かり，また，それ以降のファイルアクセスを禁止することができる．Keypad はファイル単位で暗号化を行い，暗号鍵をサーバに保存する．そのため，ネットワークに接続されていなければファイルを復号できず，安全に暗号化を行うこともできない．もう一台の端末を用いる解決法も提案されているが，対象がスマートフォンの場合にはそれ以外の端末を携帯しなければならないのは実用上の障害となる．Keypad ではページキャッシュも暗号化する設計となっているが，FUSE を用いて実装されているため，ページキャッシュの扱いの詳細については不明である．

スワップ暗号化 [10] はディスクにスワップアウトされるページを暗号化し，スワップインされる時に復号する．短い期間だけ有効な暗号鍵を用いて暗号化を行うことにより，プロセスが終了した後はスワップアウトによってディ

スクに書き込まれたデータを復号することができなくなる．Cache-Crypt ではページキャッシュを暗号化しているため，スワップアウトされるページは暗号化済みである．

Cryptkeeper [9] はソフトウェアによる暗号化を行う仮想メモリマネージャである．Cryptkeeper はメモリを小さなワーキングセットとそれ以外の部分に分け，後者を暗号化する．暗号化されたページはアクセス禁止にされ，アクセスされてページフォールトが発生した時に復号される．暗号化されないページ数が上限を超えた場合には，その中で最初に復号されたページが暗号化される．Cryptkeeper はプロセスのメモリのみを暗号化し，カーネルのメモリは暗号化しないため，ページキャッシュは暗号化されない．また，暗号鍵はメモリ上に置かれ，コールドブート攻撃対策が行われていない．

Lacuna [2] は，プライベートセッションでアプリケーションを実行し，セッションが終わると実行に関する全てのメモリを消去するシステムである．Lacuna は，プロセス間通信が制限された VM 上でアプリケーションを実行する．外部デバイスとはエフェメラルチャネル経由で接続される．ハードウェアチャネルを用いる場合はゲスト OS のデバイスドライバが直接アクセスし，QEMU がエミュレーションする場合はホスト OS 内を通るデータを暗号化する．これにより，アプリケーションの終了時に消去すべきデータが存在する範囲を限定する．

Vanish [5] や CleanOS [14] は一定時間たつと機密データを暗号化して，暗号化に用いた鍵をインターネット上に格納する．CleanOS [14] は Android OS を拡張し，SDO と呼ばれる Java オブジェクトに機密データを格納する．独自の GC を用いて SDO のデータが頻繁に使われているかどうかを判定し，使われていなければ暗号化して鍵をクラウド上に格納する．そのため，ネットワークに接続されていない時は SDO を暗号化するまでの時間を長くしたり，暗号鍵を端末に保持させたりする必要があり，情報漏洩の危険性が増す．

7. まとめ

本稿では，メモリ上のページキャッシュを暗号化することでコールドブート攻撃を行われたとしても情報漏洩を防ぐシステム Cache-Crypt を提案した．Cache-Crypt では，OS がディスク上のデータを読み書きする際に作られるページキャッシュを暗号化し，アプリがアクセスする際にだけ復号する．これにより，コールドブート攻撃による情報漏洩の対象をアプリがアクセス中のページキャッシュに限定することができる．また，復号したページキャッシュの再暗号化のオーバヘッドを削減するために，Cache-Crypt は暗号化の遅延を行う．Cache-Crypt を Android OS に実装し，ファイルの読み書きにおける Cache-Crypt のオーバヘッドが 28%程度であることを確認した．

今後の課題は，Cache-Crypt とフルディスク暗号化の連携を行えるようにすることである．同じ暗号方式および同じ暗号鍵を用いてディスクの読み書きの際の復号化と暗号化の回数を減らすことで，Cache-Crypt のオーバヘッドの削減をする．また，ARMORED [8] を Android OS に実装し，暗号鍵をデバッグレジスタ上に保持して性能測定を行うことも今後の課題である．

参考文献

[1] M. D. Corner and B. D. Noble. Zero Interaction Authentication. In *Proc. of the ACM Annual International Conference on Mobile Computing and Networking*, 2002.

[2] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein. Y. Xu, V. Shmatikow, and E. Witchel. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[3] J. Halderman et al. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. USENIX Security Symposium*, 2008.

[4] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[5] R. Geambasu, T. Kohno, A. levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security*, 2009.

[6] M. A. Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proc. of the Linux Symposium*, pages 2495–2503, 2005.

[7] Taylor Hornby. EncFS Encrypted Filesystem. `https://defuse.ca/audits/encfs.htm`, 2 2014.

[8] J. Götzfried and T. Müller. ARMORED CPU-bound Encryption for Android-driven ARM Devices. In *Proc. of the 8th ARES Conference (ARES 2013)*, 2013.

[9] P. A. H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Proc. of the IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[10] N. Provos. Encrypting Virtual Memory. In *Proc. of the 9th USENIX Security Symposium*, 2000.

[11] S. Sharma. TransCrypt: Design of a Secure and Transparent Encrypting File System. Master's thesis, Indian Institute of Technology Kanpur, 2006.

[12] Joe Sylve. LiME - Linux Memory Extractor. `https://code.google.com/p/lime-forensics/`, 2012.

[13] T. Müller and M. Spreitzenbarth and F. C. Freiling. FROST – Forensic Recovery of Scrambled Telephones. In *Proc. of The 11th International Conference on Applied Cryptography and Network Security (ACNS)*, 2013.

[14] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, 2012.