Vol.2022-CSEC-99 No.2
Vol.2022-SPT-49 No.2
Vol.2022-EIP-98 No.2
2022/12/22

実行ファイル改変による キャッシュ攻撃への対策手法の検討

島田祐希[1]　楫勇一[1]

概要：Flush+Reload や Cache Template Attack といったプロセッサキャッシュに対するサイドチャネル攻撃への新たな対策手法を提案する．提案手法では攻撃対象となる実行ファイルのコンパイル段階で，表面上の動作を変えずに異なる実行ファイルを得られるようアセンブリコードレベルで処理を行い，改変された実行ファイルへの他者のアクセスを制限することを考える．これにより，攻撃実施プログラムによる攻撃対象プログラムのメモリライン共有を阻害し，Flush+Reload が実行される前提条件の成立を妨げる．本研究では，提案手法の概要について述べ，提案手法を実行する試作システムについて紹介する．簡単な Hello World プログラムや AES の暗号化プログラムに対し提案手法を適用したところ，コンパイル毎に異なる機械語コード列からなる実行ファイルを生成することが確認できた．また，ファイルサイズやプログラム実行時間への影響も限定的であることを確認し，提案する手法のオーバヘッドが非常に小さいことが明らかとなった．

キーワード：Flush+Reload，Cache Template Attack，プロセッサキャッシュ，サイドチャネル攻撃

Investigation of countermeasures against cache attacks by modifying executable files

YUKI SHIMADA[†1]　YUICHI KAJI[†1]

Abstract: We propose a new countermeasure against side-channel attacks such as Flush+Reload and Cache Template Attack that target processor caches. The proposed method prevents the sharing of memory lines between an attacking spy program and a targeted victim program. This is realized by randomly modifying assembly codes during the compilation step while the modification is made in such a way that it does not change the behavior of the program. This manuscript outlines our approach and introduces an experimental implementation of the proposed method. Through a preliminary investigation over a sample Hello World Program and an AES encrypting program, it is confirmed that different executable files with the same functionality are indeed composed and the overhead of the modification is small and feasible.

Keywords: Flush+Reload, Cache Template Attack, processor caches, side-channel attacks

1. はじめに

装置や情報システムの動作状況を直接的または間接的に外部から観察し，装置内・システム内の情報の取得を試みる攻撃をサイドチャネル攻撃[8]という．サイドチャネル攻撃には様々なバリエーションが存在し，本研究で扱う Flush+Reload[10]もサイドチャネル攻撃の一種である．サイドチャネル攻撃は何を観察対象とするかで分類され，Flush+Reload はプロセッサキャッシュを観察対象とするサイドチャネル攻撃に分類される．

Flush+Reload は，攻撃対象プログラムと攻撃実施プログラムを同一プロセッサにおいて並列実行させ，攻撃実施プログラムにプロセッサキャッシュの内容を間接的に観測させることで，攻撃対象プログラムの特定の箇所（ターゲットコード）がいつ実行されたか知ることを目的とする攻撃手法である．これにより，プログラムの実行フローを外部から追跡し，条件分岐に関与する変数値の推測等を行うことで，暗号鍵等の機密情報の特定を試みる．一連の攻撃を実現するため，攻撃実施プログラムは攻撃対象プログラムと同一のメモリライン，同一のプロセッサを共有するよう動作する．攻撃実施プログラムは，ターゲットコードの内容をキャッシュから掃き出し（Flush），一定時間待機した後に，共有されたメモリラインからターゲットコードの内容を再ロード（Reload）する．再ロードに要する時間が短ければ，ターゲットコードはキャッシュに存在していたと推測されるが，一度掃き出したはずのターゲットコードがキャッシュに存在しているのは攻撃対象プログラムがターゲットコードにアクセスしたため，したがって，ターゲットコードを実行したためだと類推することができる．

Flush+Reload が他のキャッシュ攻撃と比較して特徴的なのは観察対象を LLC(Last Level Cache) としている点である．Flush+Reload 以前に考案されたキャッシュを対象とするサイドチャネル攻撃[4]では，攻撃に際し攻撃実施プログラムと攻撃対象プログラムが同一のコアで実行される必要があり，攻撃の実施には高い技術力が必要であった．一方，Flush+Reload ではコアレベルでの並列実行は必要なく，プロセッサレベルでの並列実行だけで攻撃を実施することが可能であるため，技術力の低い攻撃者であっても攻撃に成

1 名古屋大学情報学研究科
Nagoya University, Graduate School of Informatics

Vol.2022-CSEC-99 No.2
Vol.2022-SPT-49 No.2
Vol.2022-EIP-98 No.2
2022/12/22

功する可能性が高くなっている．加えて，コアレベルでの並列実行が必要ないという特徴は，同一プロセッサ上で動作している仮想環境に対する攻撃も可能としている[6][9].

Flush+Reload から派生した攻撃手法も報告されており，Cache Template Attack[3]は, Flush+Reload を利用した攻撃の実行に必要となる情報の取得を自動化した手法である．この特徴により，攻撃対象プログラムの詳細な分析等を行わなくてもターゲットコードを特定することが容易となり，専門性の低い攻撃者であっても攻撃を行うことが可能となる．また，攻撃対象プログラムのソースコード等が入手できない場合であっても，入力されたパスワード等の秘密情報を特定するような攻撃も可能となる．Cache Template Attack を防止するには Flush+Reload の実行を阻害する必要があるため，Flsuh+Reload への対策は関係する他の攻撃方法の抑止にもつながる．

Flush+Reload の対策手法は複数提案されているが，ハードウェアの大きな変更を必要としたり，パフォーマンスの低下を招いたりする等の理由から，実用性や有用性に問題のある手法が多い[7][11]．また，Cache Template Attack への対策手法に関しては十分な議論がされておらず，根本的な Flush+Reload の無効化以外に明確な対策手法は提案されていない．本研究では，Flush+Reload 実行の前提となる「メモリラインの共有」を人為的に困難にすることで，Flush+Reload や Cache Template Attack に対処することを考える．提案手法はハードウェアの変更やオペレーティングシステムの改変等の大掛かりな対応を必要とせず，利用者個人のレベルで実現可能な軽量な仕組みとなっている．本稿では，提案手法の概要を示し，検討するアプローチの有用性について検討を行う．

2. キャッシュに対するサイドチャネル攻撃

2.1 プロセッサキャッシュ

プロセッサキャッシュは，プロセッサ内に装備された高速・小容量な記憶装置である．データの局所性に基づき，使用したデータを一時的にキャッシュに保存しておくことで，そのデータの再利用時にかかるアクセス時間を短縮させることが可能となる．プロセッサキャッシュには，プロセッサの高速な処理とメモリアクセスという低速な処理のギャップを埋めることで，全体としての処理スピードを向上させる役割が与えられている．

プロセッサキャッシュは階層構造を持ち，コアに近い側から L1 キャッシュ，L2 キャッシュ，L3 キャッシュというように記憶階層を形成する．ここで最もコアから離れた位置にあるキャッシュ階層を LLC(Last Level Cache) と呼ぶ. 図 1 は本研究で使用した Intel Core i3-4160 のキャッシュ構造である[5]. この場合は L3 キャッシュが LLC となる．図からも分かるように，マルチコアのプロセッサの場合，

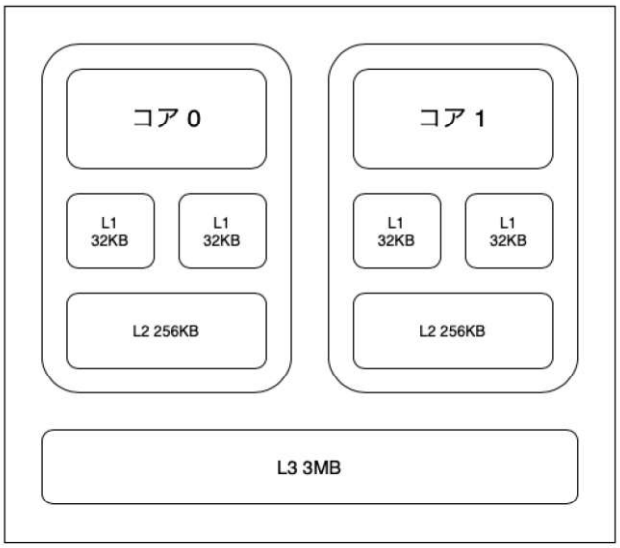

図 1　Intel Core i3-4160 のキャッシュ構造

L1 キャッシュや L2 キャッシュが各コアにそれぞれ備わっているのに対し，LLC は全てのコア間をまたがる包括的な階層となっている．Intel Core i3-4160 は各コアが 64KB の L1 キャッシュ（うち 32KB はデータキャッシュ，残り 32KB は命令キャッシュ）と 256KB の L2 キャッシュを持ち, 2 つのコアが 3MB の L3 キャッシュを共有している．

LLC は上位のキャッシュ階層が持つデータのコピーを全て保持しているため, LLC からデータを取り除く（Flush する）と上位階層のキャッシュからも自動的にデータが取り除かれることになる．Flush+Reload はこの振る舞いを利用している．また，コアの近くに位置するキャッシュ階層からデータを読むよりもコアから離れたキャッシュ階層やメモリからデータを読む方が長い時間がかかる．この読み込みにかかる時間の差を利用しサイドチャネル攻撃を行うことが可能となる．

2.2 Flush+Reload

Flush+Reload はメモリページの共有を利用した，プロセッサキャッシュに対するサイドチャネル攻撃の一種である．Flush+Reload を用いた攻撃実施プログラムの目的は，同一のシステム上で並列して実行される攻撃対象プログラムにおいて, 特定の箇所（ターゲットコード）がいつ実行されるかを知ることにある．これを実現するには，攻撃実施プログラムと攻撃対象プログラムが同一のメモリラインを共有する必要がある．したがって，Flush+Reload を実行するにあたって，まず攻撃実施プログラムは攻撃対象プログラムの実行ファイルを攻撃実施プログラムの仮想アドレス空間にマッピングし，メモリラインの共有を行う．メモリラインの共有が行えたら，いつ実行されるかを知りたいターゲットコードを含むページ(ターゲットページ)を設定し，以下のように 3 段階からなる攻撃を行う．

1. ターゲットページをキャッシュから排除(Flush)する．

Vol.2022-CSEC-99 No.2
Vol.2022-SPT-49 No.2
Vol.2022-EIP-98 No.2
2022/12/22

2. 一定時間待機する．

3. ターゲットページへアクセス（Reload）して，アクセス完了までの時間を観測する．

2 の待機時間の間に攻撃対象プログラムからターゲットページへのアクセスがあった場合，ターゲットページはキャッシュに読み込まれて保存されるため，攻撃実施プログラムによるリロードにおいても，その読み込み時間は短くなる．逆に，攻撃対象プログラムによるターゲットページへのアクセスがなかった場合，攻撃実施プログラムのリロード操作ではターゲットページをメモリから読み込むことになるため，リロードにより長い時間がかかってしまうことになる．このリロードにかかる時間の差異から攻撃対象の挙動の情報を推測するのが Flush+Reload という手法である．

Flush+Reload を利用した攻撃例としては，暗号アルゴリズムで行われる各演算に対応するコードをターゲットコードに設定し，Flush+Reload を実行することで計算過程を追跡し，条件分岐等で参照される鍵情報を推測するものなどがある．

2.3 Cache Template Attack

Cache Template Attack は Flush+Reload を利用した派生攻撃手法である．Flush+Reload を攻撃に利用するためには，攻撃対象の挙動を推測できるような適切なターゲットページを選択する必要があり，攻撃対象となるソフトウェアやアルゴリズムに関する高度な知識が必要とされた．Cache Template Attack は，ターゲットページの選択を自動化した攻撃手法であるため，専門性の低い攻撃者でも攻撃を実行することが可能となっている．

Cache Template Attack は，ターゲットページの選択を行うためのプロファイリング段階と，攻撃を実行するための悪用段階の 2 段階で構成されている．この 2 つの段階を実現するにあたり，ターゲットページを盲目的・網羅的に選択する形で Flush+Reload が実行される．

プロファイリング段階では，攻撃対象プログラム上での，キー入力などの特定のイベントの実行中に，特定のメモリページにどれだけのキャッシュヒットが発生したかを測定する．検知したいイベントを繰り返し実行し，その間複数のメモリページに対して繰り返し Flush+Reload を実行する．これにより，特定のイベントにどのメモリページが対応しているかを確認し，イベントを検知するためのターゲットページを特定することが可能となる．

悪用段階ではプロファイリング段階の結果をもとに設定したターゲットページに対して Flush+Reload を実行する．実行結果を分析することで，特定のイベントの発生を検知することができる．

Cache Template Attack を実行するにあたって必要となるのは，Flush+Reload を実行できることと，攻撃対象プログラムと同一の実行ファイルを実行できることである．

2.4 既存の対策手法

Flush+Reload の対策手法としては，Flush+Reload を実現させている clflush 等の命令使用に制限をかけることや，メモリ共有の無効化などが挙げられる．しかし，これら手法はハードウェアの大きな変更を必要とし，パフォーマンスの低下を招く．ターゲットページの特定を困難にするように，プログラム難読化等の手段[2] を利用して攻撃対象プログラムのソースコードを改変することで Flush+Reload による攻撃を防ぐことができる場合もあるが，これは攻撃対象と同一の実行ファイルを実行できる環境さえあれば攻撃を成功させ得る Cache Template Attack に対しては不十分である．

3. 提案手法

3.1 アプローチと前提条件

本研究では，攻撃実施プログラムによるメモリラインの共有を阻害することで，Flush+Reload の実施を困難にするアプローチについて検討を行う．

環境や設定に依存する部分は多いが，多くのマルチユーザシステムにおいて，実行ファイルは複数のユーザにより共有される資源として扱われることが多い．もちろん，実行ファイルに対して特定ユーザ以外のファイルアクセスを禁止するよう設定することも可能であるが，多くのユーザが利用するプログラムへのアクセスを禁止すると，利便性を大きく損なったりシステム稼働に支障が生じたりする可能性もある．

実行ファイルを複製し，1 ユーザのみにアクセスを許すよう設定すれば，攻撃対象プログラムが直接的に攻撃実施プログラムから参照されないようすることは可能である．しかし，文脈ベースページ共有（メモリ重複排除）を採用したシステム（Linux や Windows 等もこれに該当する[1]）においては，実行ファイルがメモリ展開された段階で内容の重複するページが検知され，1 つのページに集約されるため，たとえファイルレベルでの共有を禁止したとしても，メモリライン共有を禁止する効果は生じない．

運用上の弊害を甘受した上で，システム内で唯一の実行ファイルへのアクセスを禁止すれば，上述のファイル複製で生じるような問題を回避することは可能である．しかしその場合であっても，攻撃者が攻撃対象プログラムのソースコードを入手し，システム内でアクセス制限下に置かれた攻撃対象プログラムと同一の実行ファイルを構成できてしまう可能性は排除できない．

以上からわかるとおり，実行ファイルへのアクセス禁止だけでは，メモリラインの共有を阻害することはできない．本研究では，実行ファイルへのアクセス禁止に加え，もうひと工夫を行うことでメモリラインの共有阻害を実現する．必要となる前提条件は，実行ファイルへのアクセス禁止だけであり，cflush 命令の使用制限やメモリ共有等の無効化

Vol.2022-CSEC-99 No.2
Vol.2022-SPT-49 No.2
Vol.2022-EIP-98 No.2
2022/12/22

処理といったシステム全体にかかる措置は必要ないため汎用性が高く，たとえば Flush+Reload の性質を利用した仮想環境への攻撃などに対しても対応できると考えられる．

3.2 提案手法

提案手法では，ソースコードから生成されたアセンブリコードのランダムな行に nop 命令を挿入する．nop 命令は何もしないという命令であるため，挿入したとしても，コンパイルの結果得られる実行ファイルの動作には影響しないはずである．この nop 命令の特徴を利用し，コンパイルごとにランダムな位置に nop 命令を挿入することで，表面上の動作を変えることなく異なる実行ファイルを得られるようにする．生成された実行ファイルは，正当な利用者のみがアクセスできるようファイル設定を行い，他者（攻撃者）が実行ファイルを読み込むことを防止する．たとえ攻撃者が攻撃対象プログラムのソースコードを入手し，自身でコンパイルを行ったとしても，挿入される nop 命令の位置が攻撃対象プログラムとは異なるものとなっているはずである．これにより，攻撃実施プログラムと攻撃対象プログラムのメモリライン共有を阻害し，Flush+Reload の実行を防ぐことが可能となる．提案手法の動作の様子を示すため，C 言語で記述された Hello World プログラムのオリジナルのアセンブリコードを図 2 に，これに対しランダムに nop 命令を挿入したものを図 3 に示す．図 3 のコードにおいて，マーカーで色付けされた nop 命令が，事後にランダムに挿入されたものとなる．

nop 命令の挿入を行いコンパイルした実行ファイルと通常のコンパイルで生成した実行ファイルを比較すると，機械語コード列の変化を確認することができる．攻撃者が自らコンパイルした実行ファイルを解析したとしても，提案手法により加工された攻撃対象プログラムの実行ファイルとは機械語コード列が異なるため，解析結果を利用して Flush+Reload を実行することはできない．

```
        .file "hello.c"
        .section    .rodata
.LC0:
        .string     "Hello World"
        .text
        .globl      main
        .type main, @function
main:
.LFB0:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq  %rsp, %rbp
        .cfi_def_cfa_register 6
        movl  $.LC0, %edi
        call  puts
        popq  %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident      "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
        .section    .note.GNU-stack,"",@progbits
```

図 2　Hello World プログラムのアセンブリコード

```
        .file "hello.c"
        nop
        .section    .rodata
        nop
.LC0:
        .string     "Hello World"
        .text
        nop
        .globl      main
        nop
        .type main, @function
main:
.LFB0:
        nop
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq  %rsp, %rbp
        nop
        .cfi_def_cfa_register 6
        movl  $.LC0, %edi
        nop
        call  puts
        popq  %rbp
        nop
        nop
        .cfi_def_cfa 7, 8
        ret
        nop
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident      "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
        .section    .note.GNU-stack,"",@progbits
```

図 3　Hello World プログラムに対する nop 命令挿入例

以上のようにして，アセンブリコードへの nop 命令挿入によって Flush+Reload を防ぐことができると考えられる．本手法の実行にハードウェアの変更は必要としないため，容易に導入することができる．

3.3 実装

前節では nop 命令をランダムに挿入すると記載したが，完全にランダムに nop 命令を挿入してしまうとアセンブリコード内のデータ領域等が損なわれ，実行ファイルの動作に不具合が生じる可能性がある．そのため，提案手法の実装にあたっては，エラーを引き起こしてしまうと考えられる行は，ランダム挿入の選択肢から除外するように設定する必要がある．たとえば図 3 に示したアセンブリコードの場合では，6 行目にある.string の直前の行に nop 命令を挿入してしまうと，プログラムの実行結果として得られるはずの Hello World という文字列が文字化けして表示されてしまう．複数の実行結果から検証し，文字列や数字のデータ定義を行う命令が書かれた行の直前に nop 命令を挿入するとプログラムが正常に動作しなくなってしまうことがわかったため，そのような行をランダム挿入の選択肢から除外している．

作成した nop 挿入プログラムでは，アセンブリコード内の，挿入しても動作に支障のない行をランダムに選択し，任意の個数の nop 命令を挿入する．

3.4 実行例

図 4 は，前節で述べた Hello World プログラムを通常の gcc でコンパイルして得られた実行ファイルのバイナリダンプである．これに対し図 5 は，挿入 nop 数を 100 に設定し，提案手法によりアセンブリコードを改変した実行ファ

Vol.2022-CSEC-99 No.2
Vol.2022-SPT-49 No.2
Vol.2022-EIP-98 No.2
2022/12/22

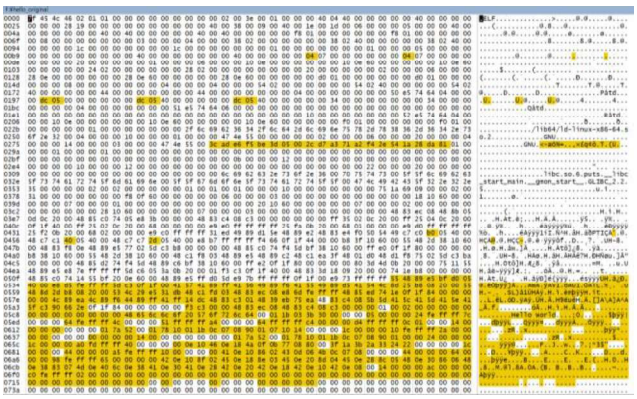

図 4　通常のコンパイルで生成した実行ファイル

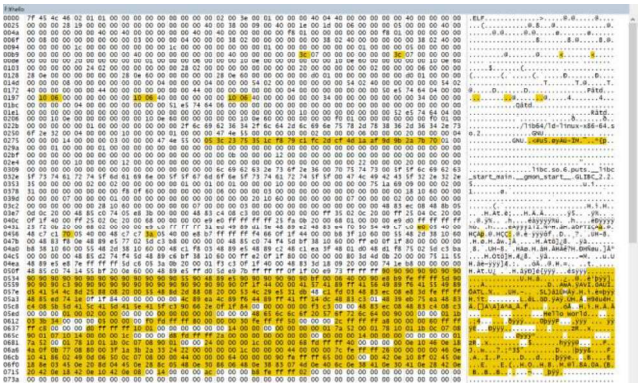

図 5　提案手法コンパイルで生成した実行ファイル

イルのバイナリダンプである．色のつけられた箇所は両ファイルで異なっているバイトデータであり，この比較より，異なる機械語コード列が生成できていることがわかる．

3.5　パフォーマンス低下検証

提案手法では元々のアセンブリコードに nop 命令の記述を追加するため，ファイルサイズ増加や実行速度の低下が懸念される．パフォーマンスの低下度合いを確認するために，AES 暗号プログラム[12]に対し提案手法を用いてコンパイルを行い，通常のコンパイルで生成した実行ファイルとパフォーマンスの比較を行った．本検証で使用したプログラムのオリジナルのアセンブリコードは 1960 行である．

まず，挿入する nop 命令の数を 250，500，750，1000，10000 とした実行ファイルを生成し，それぞれについてファイルサイズを比較した．結果を表 1 に示す．

表 1 より，挿入数 500 までは比較的緩やかにファイルサイズが増加しているが，nop 数が 750 を超えるとファイルサイズが急激に増加し，その後しばらくは漸増していることがわかる．挿入する nop 数とファイルサイズの増加量は線形的ではなく，離散的に振る舞うことが確認できたが，この原因については本稿執筆時点で明らかになっていない．

次に，実行速度の比較を行った．それぞれの実行ファイルにおいて，暗号化と復号の処理を 100 回ずつ行い，合計の実行時間を比較した結果を表 2 に示す．

表 1　nop 命令挿入数別実行ファイルサイズ

nop 命令挿入数	ファイルサイズ
0	15360 バイト
250	15424 バイト
500	15488 バイト
750	19624 バイト
1000	19744 バイト
10000	24152 バイト

表 2　nop 命令挿入数別実行時間

nop 命令挿入数	実行時間
0	0.146s
250	0.149s
500	0.154s
750	0.162s
1000	0.163s
10000	0.194s

表 2 より，挿入数 1000 までは実行時間には誤差程度の差しか生まれず，挿入数 10000 であっても，実行時間に与える影響は小さいことがわかる．

検証により，提案手法によるコンパイルが引き起こす実行速度の低下はわずかであることがわかった．実行ファイルサイズの比較においては，nop 命令の挿入数を多くするほど影響は大きくなっていくことが確認されたが，挿入数 500 まではファイルサイズへの影響は小さいことがわかった．図 3，4 の例のように，挿入数 100 程度であっても，十分に異なる機械語コード列の生成は行えているため，実用の場面において，ファイルサイズへの影響はほとんどないと考えられる．nop 命令挿入数を多くしすぎなければ，提案手法によるコンパイルはパフォーマンスの低下につながらないといえる．

4.　ソフトウェア難読化との関係

本稿では，アセンブリコードの改変によるキャッシュ攻撃の対策手法の概要を示し，提案手法の有用性について検討を行った．

アセンブリコードの改変ではなく，ソフトウェア難読化の技術を利用しても，本研究と類似する効果を生み出せる可能性はある．ただし，ソフトウェアの難読化は，プログラムの開発者等がソースコードのレベルで実施することが一般的であり，プログラムの内容について熟知していない第三者（一般的なソフトウェア利用者）が実行ファイルからソースコードを復元し，ソースコードの内容を理解したうえで適切な難読化処理を行うことは容易でないと考えられる．また，ソースコードレベルで難読化を行っても，コンパイルの際の最適化処理により難読化の効果が除去され，

結果として実行ファイルに変化が生じない可能性も考えられる．一方，攻撃対象プログラムのソースコードにアクセスできない場合や実行ファイルからソースコードの復元が困難な場合であっても，逆アセンブリによりアセンブリコードを入手することは比較的容易である．また，アセンブリコードのレベルで行われた改変は最適化処理等の影響を受けず，実行ファイルに直接的な変化をもたらす．このことから，アセンブリコードレベルで処理を行うことについて，一定の優位性があると考えられる．

5. おわりに

本研究では，プログラムのアセンブリコードに nop 命令を挿入することで実行ファイルを改変し，Flush+Reload 等における攻撃対象プログラムのメモリライン共有を阻止する手法を提案した．また，提案手法を実装し，ファイルサイズや実行時間に与える影響が比較的小さいことを実験的に確認した．本稿執筆時点においては，提案方式も試作実装も基本的なレベルに留まっており，これから更に精密化を図ることも可能であると考えている．また，現時点では C 言語で記述されたプログラムにしか対応できていないが，今後，同様のアプローチにより対応可能な範囲を拡大していくことを目指したい．

参考文献

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM, 2009 Linux Symposium, pp. 19–28, 2009.
[2] Ernie Brickell, Gary Graunke, Michael Neve and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities., IACR Cryptology ePrint Archive, vol. 2006, pp. 52, 2006.
[3] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard Graz. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches, 24th USENIX Security Symposium, pp. 897–912, 2015.
[4] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games –Bringing Access-Based Cache Attacks on AES to Practice, 2011 IEEE Symposiumon Security and Privacy, pp. 490–505, 2011.
[5] Intel. インテル Core i3-4160 プロセッサー(3M キャッシュ, 3.60 GHz) 製品仕様, https://ark.intel.com/content/www/jp/ja/ark/products/77488/intel-corei3-4160-processor-3m-cache-3-60-ghz.html, (参照 2022-11-28).
[6] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES, Research in Attacks, Intrusions and Defenses, pp. 299–319, 2014.
[7] Minwoo Jang, Seungkyu Lee, Jaeha Kung, and Daehoon Kim. Defending Against Flush+Reload Attack With DRAM Cache by Bypassing Shared SRAM Cache, IEEE Access, pp.179837–179844, 2020.
[8] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems, Advances in Cryptology — CRYPTO '96, pp.104–113, 1996.
[9] Danny Philippe-Jankovic, and Tanveer A Zia. Breaking VM Isolation – An In-Depth Look into the Cross VM Flush Reload Cache Timing Attack, IJCSNS International Journal of Computer Science and Network Security, **17**, *2*, February 2017, pp.181–193, 2017.
[10] Yuval Yarom, and Katrina Falkner. FLUSH+RELOAD: a High Resolution,Low Noise, L3 Cache Side-Channel Attack, 23rd USENIX Security Symposium, pp. 719–732, 2013.
[11] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches, 23rd ACM Conference on Computer and Communications Security, pp.871–882, 2016.
[12] ピイジェイ. AES 暗号, https://free.pjc.co.jp/AES/index.html, (参照 2022-11-28).