暗号化によるプログラムの保護機能をサポートする プロセッサ・アーキテクチャ

１７５０００４４

平成１７年度～平成１８年度科学研究費補助金

（基盤研究（C））研究成果報告書

平成１９年５月

研究代表者　北 村　俊 明

広島市立大学 情報科学部 教授

《はしがき》

本研究報告書は、科学研究費補助金基盤研究(C)「暗号化によるプログラムの保護機能をサポートするプロセッサ・アーキテクチャ」（平成１７年度～１８年度　課題番号　１７５０００４４）の研究成果をまとめたものである。

本研究の基本的な課題は、プログラムのアルゴリズムを悪意のある第三者の逆アセンブルなどのリバースエンジニアリングから保護しようというものである。これは、プログラムやデジタルコンテンツの複製による不正使用を防止しようと言う試みは各種行われているが、最近重要度を増している組み込みシステムで使用されるプログラムは、直接、制御対象となる装置を制御するプログラム論理を持っており、そのアルゴリズムは、対象装置の仕様から簡単に導き出せる論理や手順だけでなく、より良い制御を行うために各種のノウハウが使われている。このようなノウハウは、公表すること自体が権利の消滅につながり、特許で保護することが難しい。また、格納されているプログラムを逆アセンブル等で解析することで容易に知ることができる。

本研究では、組み込みシステムなどで、独立したチップの主記憶上に格納されるプログラムを、公開暗号鍵による暗号化を行って格納し、プロセッサは暗号化された機械語プログラムを直接取り込んで復号化し実行するという機構で、プログラムの逆アセンブルを阻止する方式を提案し、その実現可能性を検討する。特に、このために必要となるハードウェア量、性能に対する影響、コンパイラやOSなどに対する影響を明らかにし、この方式の安全性について評価を行うものである。

研究組織

研究代表者　：　北村　俊明（広島市立大学情報科学部教授）
研究協力者　：　城本　正尋（広島市立大学大学院 情報科学研究科 在学時）
酒井　智也（広島市立大学大学院 情報科学研究科 在学時）
田端　猛一（広島市立大学大学院 情報科学研究科 学生）
山下　純一（広島市立大学 情報科学部 在学時）

交付決定額（配分額）　（金額単位：円）

	直接経費	間接経費	合計
平成１７年度	1,700,000	0	1,700,000
平成１８年度	1,500,000	0	1,500,000
総　計	3,200,000	0	3,200,000

研究発表

（1）学会誌

・城本正尋, 田端猛一, 酒井智也, 島田貴史, 窪田昌史, 北村俊明. 公開鍵暗号を用いてプログラムの保護を行うプロセッサの提案. 情報処理学会論文誌, Vol. 47, No. SIG 18 (ACS 16),

pp. 55–64，１１月 2006年.

（２）口頭発表

・城本正尋, 田端猛一, 酒井智也, 島田貴史, 北村俊明. 公開鍵暗号を用いてプロ グラムの保護を行うプロセッサの開発. 情報処理学会研究報告, ARC-165, pp. 21–26，１１月 2005年.

・酒井智也, 田端猛一, 北村俊明. プログラム保護を行うプロセッサの公開鍵暗号ハードウェアの評価. 情報処理学会研究報告, ARC-170, pp. 19--24，１１月 2006年.

•Junichi Yamashita, Takekazu Tabata, Toshiaki Kitamura, Reverse Engineering on the Processor with Program Protection Feature, IEEE Symposium on Low-Power and High-Speed Chips (Cool Chips X), Poster 9, p.143, April 2007

（３）出版物

・なし

研究成果による工業所有権の出願・取得状況

・なし

目 次

第1章　はじめに

近年，家電製品や通信機器などでは，組み込みプロセッサによって装置制御を行う場合が増えている．制御をハードウェアで実現する場合に比べて，組み込みプロセッサを用いた制御プログラムによる実現では，機能の変更・追加・修正は容易になり，製品の開発サイクルを大幅に短縮できるメリットがある．しかし，システム開発者にとってプログラムによる制御を行うことは，悪意を持った第三者のリバースエンジニアリングによって重要なプログラムを不正に使用される危険性を含んでいる．

一部の特殊なアルゴリズムについては，特許などの手段で保護することも可能である．しかし，組み込みシステムで使用されるプログラムには，制御対象となる装置の仕様から簡単に導き出せる論理・手順だけでなく，より良い制御を行うために各種のノウハウが使われている．たとえば，プリンタの発色管理では，単に出力する画素の色情報そのもので制御するのではなく，印刷する紙の種類になどに応じて微妙にカラーバランスを変化させるなどの工夫がなされている．このようなノウハウは，公表すること自体が権利の消滅につながり，特許で保護することが難しい．

悪意のあるユーザーがプログラムの不正な入手をしようとした場合，システム上のプログラムへの攻撃方法としては，次のようなものが考えられる．

- 二次記憶に存在する実行プログラムに対してソフトウェアアクセスする方法
- OS の特権を利用して，通常のユーザーが知り得ない情報を取得する方法 (OS を改変できる場合)
- メモリバスなどの命令が転送されるバスを観測するハードウェア的な攻撃方法

などである．

プログラムを保護する手段の一例として，プログラムを難読化することによる耐タンパー性向上 [25] が考えられる．耐タンパー性とは，不正利用に対する耐性のことである．難読化によってリバースエンジニアリングに要するコストを増大させる手法である．また，プログラムを暗号化することで逆アセンブルを不可能にする手法もある．復号処理をソフトウェアで実装した場合は，復号ルーチンを解析，改ざんされる恐れがあり，危険である．また，システム上のプログラムに対するハードウェア的な攻撃も存在するので，ソフトウェアだけの保護では十分で

はない．そこで，プロセッサ外部ではプログラムを暗号化しておき，プロセッサ内部でプログラムの復号を行うことで，プロセッサチップ内にのみ復号された命令が存在し，命令が転送されるバスをロジックアナライザなどでハードウェア的に観測するような攻撃からもプログラムを保護する機能を有したプロセッサであるセキュアプロセッサが提案されている [7, 1, 2, 3, 4, 5, 6].

セキュアプロセッサでは，処理されるプログラムはそれぞれ異なるベンダによって開発される可能性が高く（マルチベンダ環境），各ベンダで異なる鍵を用いて暗号化できるという自由度を持つことが望ましい．そのため，公開鍵暗号と共通鍵暗号を組み合わせたハイブリッド方式を採用している．そこで，セキュアプロセッサでは，実行プログラムを復号する際に，プログラムに対応した鍵を用意する機構が必要となるが，従来，実行プログラムと鍵との対応付けをプロセス単位で行っていたのに対し，我々は，仮想記憶の枠組を利用し，ページ単位で実行プログラムと鍵とを対応付けるプログラム保護システムを提案している [7]．提案システムでは，プログラムはページ単位で復号する鍵と対応付けられるため，1 つのプロセスの中に，異なる鍵で暗号化されたプログラムを存在させることができる．

提案システムでは，プログラムのアルゴリズム保護に重点を置いて設計を行なっており，プログラム本体のみを暗号化の対象としている．プログラムのオペランドデータや割込み時のコンテキスト退避処理におけるデータの暗号化は行なわずに主記憶へ格納される．これらの暗号化されていないデータの実行時の変化から，保護するプログラムの命令列を同定される危険性が存在する．保護するプログラムのアルゴリズムが充分に複雑であれば，データによる命令列の同定は困難だと考えていたが，どの程度困難であるのかの調査は行なっていなかった．そこで，データの変化による命令列の同定難度がどの程度であるかを，同定アルゴリズムを考案し実際にデータを基に解析を試みることで，プログラムを保護する能力の評価も行なった．

以下本報告書では，次章で，関連研究のまとめとして，システム上のプログラムに対する攻撃法を述べ，それに対するセキュアプロセッサのプログラム保護の機構を説明する．また，現在までに研究されているセキュアプロセッサを紹介する．3 章では，セキュアプロセッサにおいて実行プログラムとプログラム鍵の対応付けに仮想記憶機構を利用したプロセッサを提案し，それを用いたプログラム保護システムについて説明する．4 章では，本提案システムで用いる共通鍵暗号と公開鍵暗号について調査し，実際的なインプリメントを想定しての評価を行う．5 章では，提案システムのフィージビリティ・スタディとして本提案システムを実現するために必要となるチップ面積や性能のトレードオフを調査し考察を行う．6 章では，データによる命令同定アルゴリズムについて述べる，アルゴリズムの一部である，命令種類別の解析方法について述べたのち，同定アルゴリズムによる解析結果を述べ，その結果を基に現実的に実現可能であるかの評価を行う．最後に，7 章では，まとめと今後の課題を述べる．

第2章　セキュアプロセッサによるプログラム保護

本章では，システム上に存在するプログラムに対して，悪意のある第三者が行う代表的な攻撃を挙げ，それらの攻撃に対してセキュアプロセッサがどのようにプログラムを保護しているかを述べる．そして，現在までに研究されているセキュアプロセッサを紹介する．

2.1　システム上のプログラムへの攻撃法

プログラムの不正な入手を目指す悪意のある第三者としては，商用のプログラムに対してリバースエンジニアリングを行い，無料での使用を企むような一般ユーザや，競合他社の製品から制御プログラム内の重要なアルゴリズムの取得を狙うような企業などが考えられる．そのため，システム上のプログラムに対する攻撃には，簡単に行えるものから専用の装置を必要とするものまで多種多様な攻撃が存在する（図 2.1）．以下で，図 2.1 内の攻撃の簡単な説明と，それに対する実行プログラムの保護手段を述べる．

図 2.1 の (1) のような，二次記憶に存在する実行プログラムに対してソフトウェアアクセスを行うような単純な攻撃には，実行プログラムに難読化を施すことによって，リバースエンジニアリングにかかるコストを増大させるなどの対応策が考えられている．また，難読化では完全にプログラムを秘密にできているわけではないので，特に秘密にしたい重要なプログラムについては，実行プログラムを暗号化する方法もある．この場合，復号をソフトウェアで実現すると，そのソフトウェアが攻撃され，プログラムが解読される危険があるため，プログラムの復号方法が問題となる．

次に，図 2.1 の (2) のような，実行中のプログラムに対して，システム上で同時に実行中である他のプログラムが覗き見を行うような攻撃が挙げられる．この攻撃に対しては，OS によって保護すべき実行プログラムのアドレス領域は，他のプログラムからアクセスできないように制限する方法が存在する．しかし，OS が攻撃され乗っ取られた場合には，この手法のような OS によるプログラムの保護は全く無力となる．また，OS が悪意のある第三者に乗っ取られた場合には，OS の特権を利用して様々な攻撃が可能となる．例えば図 2.1 の (3) のように，特権命令を

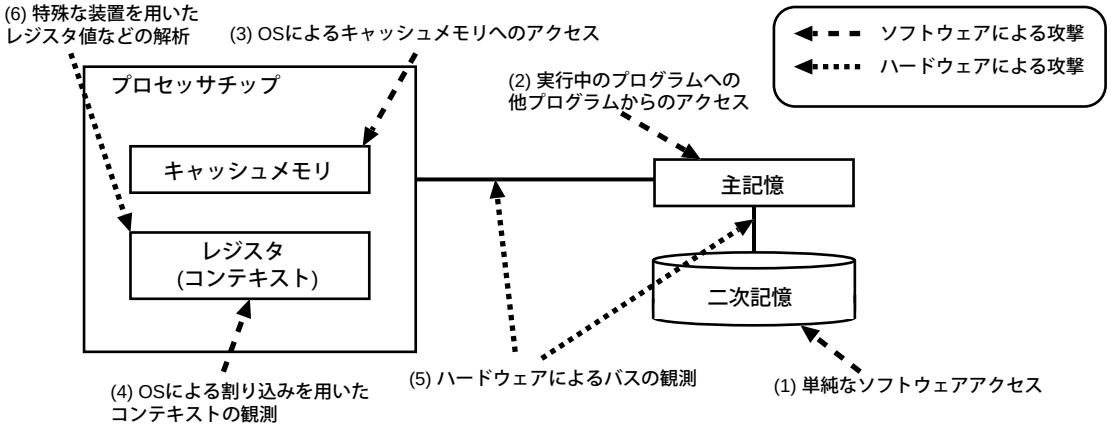

図 2.1: システム上のプログラムに対する攻撃

用いてプロセッサチップ内のキャッシュメモリにアクセスしたり，図 2.1 の (4) のように，重要なアルゴリズムを含むプログラムの実行中に，1 機械語命令毎に割り込みを発生させ，命令実行前と後に退避されたコンテキストの組から機械語命令を同定していき，重要なアルゴリズムを入手するような方法も考えられる．

以上までは，システムに対するソフトウェア的な攻撃を挙げたが，図 2.1 の (5) のように，ロジックアナライザや ICE (In Circuit Emulator) などの機器を用いて，メモリバスなどの命令が転送されるバスを観測し，その結果を逆アセンブルすることで機械語命令を得るようなハードウェア的な攻撃も存在する．その他ハードウェア的な攻撃としては，図 2.1 の (6) のように，プロセッサチップ内部のレジスタを対象とし，LSI のテストなどに使用されるビームプローブを用いて命令レジスタや汎用レジスタの値を解析する方法が挙げられる．ただし，この攻撃方法では，チップ表面を露出させる前処理や，目的とするレジスタの位置を正確に把握する必要があるため，実現可能性は高くはないと考えられる．

2.2 セキュアプロセッサ

2.2.1 基本的な仕組み

2.1 節で述べた実行プログラムを暗号化することでプログラムを保護する手法において，プロセッサチップ内部に復号回路を有し，チップ内部で暗号化されたプログラムの復号を行うプロセッサであるセキュアプロセッサが提案されている．セキュアプロセッサにおいて，プログラムは主記憶や二次記憶では暗号化された状態で存在するため，プログラムを復号する鍵を持たない悪意のある第三者による主記憶や二次記憶に対するソフトウェア的な攻撃のみならず，ハードウェア的な攻撃からもプログラムを保護することが可能となっている．

セキュアプロセッサのシステム上で暗号化された実行プログラムが起動した時には，プロセス単位で OS が管理するプロセス ID とは別に特別な ID(以降，SPID: Secure Process ID) が割り当てられる．SPID の生成・削除は，プロセッサ内の特別なハードウェアによって行われ，SPID の操作のために追加された特殊な命令を用いて，OS が生成・削除の指示のみを行う．セキュアプロセッサでは，暗号化された実行プログラムだけでなく平文[1]の実行プログラムの実行も可能となっており，平文の実行プログラムのプロセスには，SPID=#0 が割り当てられる．

また，セキュアプロセッサでは，プロセッサチップ内のキャッシュメモリの内容を保護するため，キャッシュメモリにアドレスに加えて SPID を共に格納している．キャッシュメモリのヒット判定を，アドレスと SPID 両方が一致した場合のみヒットとみなすため，セキュアプロセッサ上のシステムでは，実行プログラムは他の実行プログラムのキャッシュメモリの内容にアクセスできないようになっている．この機構は特権を持った OS のプログラムについても適用されているので，OS が乗っ取られた場合でも実行プログラムを秘密にすることができる．

さらに，セキュアプロセッサでは，実行プログラムを暗号化するだけでなく，プログラム実行時に処理しているデータも主記憶に格納する際には，暗号化を行ってから格納している．このため，割り込みが発生した際に主記憶に退避されるコンテキストも暗号化の後に格納されるので，割り込みを利用した攻撃からもプログラムを保護している．

セキュアプロセッサによるプログラムの保護機能をまとめると，プロセッサチップ外の主記憶や二次記憶への攻撃に対してはプログラム及びデータを暗号化することで対応し，プロセッサチップ内のキャッシュメモリへの攻撃に対しては SPID を付加し，他プロセスからのキャッシュメモリへのアクセスを制限することで対応している．セキュアプロセッサにおける図 2.1 の各攻撃に対するプログラムの保護を表現した図を図 2.2 に示す．

2.2.2 暗号化と復号の機構

セキュアプロセッサにおけるのプログラムの暗号化と復号の機構を図 2.3 に示す．以下，図 2.3 にしたがって説明する．

セキュアプロセッサを用いたシステムでは，ソフトウェア開発者は作成したプログラムを保護するため，誰でも実行プログラムを暗号化できる必要がある．しかし，その暗号化されたプログラムを復号できるのは，プログラムを実行するセキュアプロセッサのみでなければならない．この実現のため，セキュアプロセッサにおいて実行プログラムの暗号化には，公開鍵暗号[2]が用いられる．2 つの鍵のうち 1 つはプログラムを暗号化するための鍵としてソフトウェア開発者やベンダに公開する．もう 1 つの鍵は，復号のための鍵としてプロセッサ内に秘密に保持す

[1]暗号化されていない文章．この場合，暗号化されていない普通の実行プログラムである．
[2]非対称鍵暗号とも呼ばれ，暗号化に用いる鍵と復号に用いる鍵が異なる

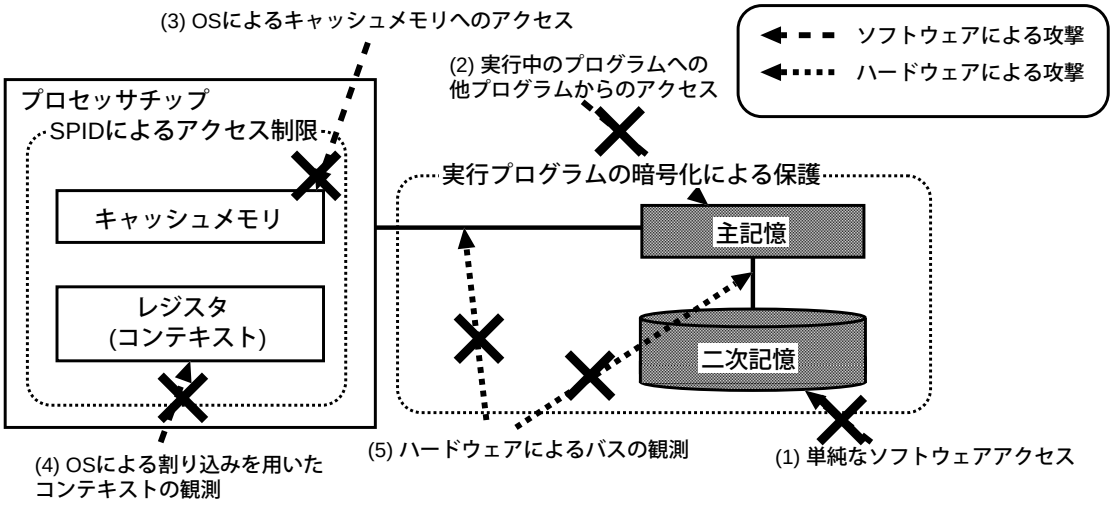

図 2.2: セキュアプロセッサによるプログラム保護

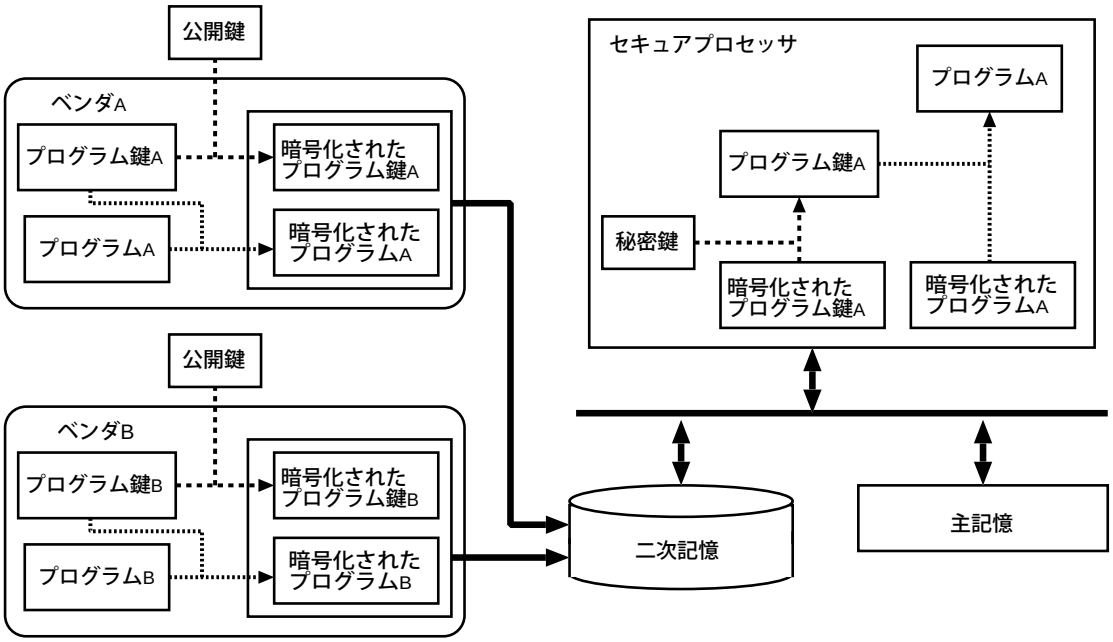

図 2.3: プログラムの暗号化と復号の概略図

る．以上のようにして，プログラムの暗号化は誰でも行えるが，復号は秘密鍵を持ったプロセッサのみが可能となり，プログラム保護機能を有したまま自由にソフトウェア開発が可能な環境が構築できる．

しかし，公開鍵暗号の暗号化及び復号処理は非常に計算量が多く，実行プログラム全域にわたって暗号化/復号を行うと，膨大な処理時間を要する．したがって，セ

キュアプロセッサでは公開鍵暗号よりも大幅に処理時間が短い共通鍵暗号[3]によって実行プログラムを暗号化し，そして，実行プログラムを暗号化した鍵（以下，プログラム鍵）は公開鍵暗号によって暗号化している．暗号化されたプログラム鍵は，実行プログラムが起動される際にプロセッサ内部の秘密鍵を用いて復号する．暗号化された実行プログラムの復号は，復号したプログラム鍵を用いてキャッシュミス時に実行される．復号した実行プログラムは通常通りキャッシュメモリに格納されるため，キャッシュにヒットすれば，復号処理を行うことなくプログラムを処理できる．プログラム鍵を公開鍵暗号で暗号化/復号することでソフトウェア開発者から安全にプロセッサ内部にプログラム鍵を渡すことができ，プログラムの暗号化は共通鍵暗号を用いるのでキャッシュミス時の復号の処理時間を抑えることができる．

2.3 関連研究

この章では，提案システムと関係がある研究についての紹介を行う．

2.3.1 プログラムの難読化

門田，高木らはプログラムを解析の困難なプログラムに変換（難読化）することで，解析にかかるコストを増大させ，事実上解析を困難にする手法を提案している [22]．難読化とは等価変換であり，復号のような逆変換を行わなくても実行することができる．[22] の論文で難読化は以下のように定義されている．

- 仕様の保存
  任意の入力に対して，難読化前と後で出力は変化しない．
- 実行速度を落とさない
  任意の入力に対して，難読化前と比べて難読化後はプログラムの実行速度は変化しない，または少ない．
- 解析の困難さの増加
  難読化前と比べて難読化後は解析にかかる時間が増大する．

なお，難読化前と後でのプログラムサイズの変化についてはある程度増加が許されるものと考えている．変換後のサイズは変換に制限を加えると変換前の 2 倍以下になるとしている．

具体的には，プログラム中のループ部分を見つけてフローチャートへ変換し，制御構造を複雑に変形しているくことでループ自身を発見しにくくするという方法

[3]対象鍵暗号とも呼ばれ，暗号化に用いる鍵と復号に用いる鍵が共通である

とループの制御変数の増減を複雑にするという方法で難読化を行っている．論文ではこの 2 つの方法を組み合わせることでより解析が困難になるとしている．

ただ，難読化したプログラムの中には解析は困難であるが，元のプログラムの情報が全て含まれているため，十分な時間をかければ解析できるはずである．そこで，実際には現実的時間内に解析するのを困難にすることが重要であるが，難読化によって，どの程度解析が困難になるかは定量的な評価がなされていない．論文では学生に 20 行程度の難読化しないプログラムと難読化したプログラムを与え，解析にかかる時間を計測することで評価をとっている．

難読化はプログラムのアルゴリズムを保護することを目的としており，難読化の 1 つの方法として暗号化が存在している．以下に紹介する研究は暗号化を利用したものである．

2.3.2 TPM

TPM(Trusted Platform Module) とは，非営利の標準化団体 TCG(Trusted Computing Group) が策定を進めているハードウェアセキュリティチップである [23]．TPM を信頼の要としてプラットフォームにおいて信頼できるコンピューティング環境を構築することを目的としている．TPM は通常，PC のマザーボードに取り付けられ，CPU からアクセスできるコプロセッサとして動作する．TPM の提供する機能の特徴として以下の点が挙げられる．

- プラットホームの認証

  TPM を搭載したプラットホームの構成を管理，また改ざんを検出し，安全なプラットホームであることを保証する仕組みである．TPM 内部には PCR(Platform Configuration Register) と呼ばれるレジスタがあり，このレジスタにソフトウェアの観測値[4]を格納，比較することで改ざんを検知する．例えば，PC の起動には，BIOS からブートローダ，ブートローダから OS という制御の移り変わりが存在するが，TPM はまず，BIOS のコードを調べ，改ざんがされていないかどうかを調査する．改ざんされていなければ，BIOS に制御を移す，そして BIOS からブートローダに制御を移す前にブートローダのコードを調べ，改ざんの有無を調査する．以降，同じように信頼できる範囲を拡張していくことで，プラットホームの認証を行う．

- プログラムの署名鍵/暗号鍵などの秘密情報の保護

  TPM では SRK(Storage Root Key) と呼ばれる公開鍵暗号の鍵ペアを TPM 内部に持っており，プログラムが利用する署名鍵/暗号鍵などの秘密情報を SRK を用いて暗号化し[5]，これを二次記憶装置に保管する．この方法により，TPM

[4]TPM 内にハッシュ値を計算する演算器があり，それを利用して観測値を計算する．
[5]TPM 内に RSA 暗号化回路があり，それを利用して暗号化を行う．

の容量を気にせずにプログラムの署名鍵/暗号鍵などの秘密情報の保護が可能となる．

TPM ではプログラムは主記憶に復号された状態で置かれることになり，ソフトウェア的な攻撃にはプラットホームの認証により危険性がないことが保証されるため安全であるが，ロジックアナライザなどで主記憶を覗き見するようなハードウェア的な攻撃は防ぐことができない．セキュアプロセッサではこの点が考慮され，主記憶上でも暗号化された状態でプログラムが置かれている．

この TPM を利用するシステムとして，Windwos Vista に搭載される BitLocker がある．これは，OS がインストールされている HDD のパーティションを OS ごと暗号化するというシステムで，暗号化に使用した暗号鍵[6]を TPM で管理するというものである．我々の提案システムとの違いは，BitLocker は盗難，紛失による HDD からのデータの盗難を防ぐことを目的としており，提案システムではプログラムの保護を目的としているという点である．BitLocker ではユーザーが暗号化を行うかどうかを決定することができ，ユーザーは自由にプログラムを見ることができる．提案システムではプログラムは常に暗号化されているため，ユーザーであっても見ることはできない．

2.3.3 XOM

David Lie らは，セキュアプロセッサの一種として XOM(eXecute-Only Memory) アーキテクチャを提案している [1, 2, 3]．我々のシステムでは重要なアルゴリズムの保護に重点を置きプログラム本体のみを暗号化しているが，彼らの XOM ではプログラムに加えオペランドデータも暗号化し，第 3 者によるデータの改竄を検知できる機能を有することで，アルゴリズムの保護に加えて耐タンパー性の向上も成し得ている．そのトレードオフとしてデータの暗号化処理が必要になり，プログラムの実行性能に影響を与えることとなる．また，プログラム鍵の割付けをプロセス単位で行っているため，様々なプロセスから共有されるライブラリの暗号化は行えない．プログラム実行中に通常モード (平文実行) と XOM モード (暗号文実行) の切り換えが可能であり，XOM モードの場合はストア命令を実行するとデータが暗号化されて主記憶に保存される．

XOM ではコンパートメントという概念に基づき，プログラムごとに扱えるデータを区分けしている．このプロセッサ内に存在する各リソース (レジスタやキャッシュ) には，そのリソースの所有者であるプログラムを指し示す情報を付与することができる．これを XOM ID と呼び，プログラムごとにユニークな XOM ID が生成されてリソースに付与される．プログラムの XOM ID とリソースの XOM ID が異なる場合，つまりリソースの所有者ではないプログラムはそのリソースにアクセスすることができず，データを読み出すことも書き換えることもできない制約

[6]AES で暗号化を行う．鍵長は 128 ビットと 256 ビットのどちらかを選択できる．

がある．加えて，各プログラムはたとえ自らの XOM ID でさえも知り得ることができず，XOM ID はハードウェアによってのみ管理される．以上の制約により，プログラムは他のプログラムのリソースにはアクセスできないという，耐タンパー性を保証する環境を構築する．OS も例外なくこの制約受けるため，このままでは割込み発生時のコンテキスト退避を完遂することができなくなっている．この問題には OS 特権となる特殊な方法を用いて対処しているが，この方法はリソースのデータ内容や XOM ID を知り得るものではない．よって OS でさえも他のプログラムに干渉することはできなくなっている．また，プログラム間でデータ通信を行いたい場合にデータの受渡しができない問題が発生するが，例外コンパートメントとして，ある特殊な領域を設けて全てのプログラムからアクセス可能としている．これをヌルコンパートメントと呼び，ここにデータを配置することでデータの受渡しを実現する．

割込みなどによって，プロセスの処理が中断すると，その時点でのプロセス状態を保存するために，レジスタ値が主記憶に退避されるが（コンテキスト切り替え），これは OS の役割である．XOM では OS が改変されている可能性があるとして，信頼をしていない．レジスタは XOM ID によってタグ付けされており，たとえ OS でも ID が異なるため，レジスタ値を読むことはできない．そのため，XOM では OS がレジスタを退避，復帰できるように特別な OS 命令を追加している．具体的には，退避時にはレジスタの内容を暗号化し，タグと一緒にハッシュ値を計算し，暗号化データ，ハッシュ値，XOM ID を OS に所有権のある特別なレジスタに保存することで，OS に内容を知られることなく主記憶に退避できる．復帰はその逆の操作で行うことができる．

また，一般に，主記憶に暗号化してストアする際には，暗合前のデータのハッシュを計算して暗号データと共に主記憶に格納する．ロード動作の際にハッシュをベリファイすることでデータの正確性を確認する．ベリファイに失敗すればロードデータはジャンクデータとして破棄される．主記憶上でデータが改竄されていればベリファイに失敗することになり，改竄されたことが検知可能となる．暗号鍵はプログラム毎に異なり，他のプログラムの暗号データをロードしても平文データを得ることはできない．

2.3.4 AEGIS

Suh らの提案する AEGIS では，XOM と同等の機能を持ち，さらに，ハッシュ値をツリー構造で効率的に管理している [4]．また，AEGIS ではプロセッサチップ内部に物理的に乱数を発生させる機能を持っており，それをアプリケーションプログラムの中で秘密の保護に利用することができる [5].

2.3.5 L-MSP

L-MSP[6] は橋本，春木らが提案しているセキュアプロセッサアーキテクチャで，ハイブリッド方式を採用しており，命令だけでなくデータの暗号化も行う．特徴として，プロセス管理の機能をプロセッサ内部に持たせており，従来，OS で行われていた割込み時のコンテキスト切り替えをハードウェアで実現している．L-MSP はソフトウェアやデジタルコンテンツの不正コピー，及びプログラムのリバースエンジニアリングを防ぐことを目的としたものである．

L-MSP ではプロセッサの認識するプロセスを ECU(Execution Control Unit) と呼び，EUC ごとに EUC ID を割り当て，プロセスを識別する．プロセッサは EUC 保護機能を備えており，EUC の持つ情報を管理している．OS は特殊な命令で EUC の生成や削除，コンテキストの切り替えを指示できるが，ECU の情報を OS が直接操作することはできないようになっている．EUC ID が OS により指定され，EUC が生成されると，暗号化されたプログラム鍵が主記憶から取り出され，EUC 保護機能内の公開鍵復号回路で平文のプログラム鍵に復号される．そのプログラム鍵はプログラム鍵テーブルの EUC ID エントリに格納される．また，乱数発生器を備えており，EUC 生成と同時に乱数が生成され，コンテキストを暗号化するコンテキスト鍵として専用の鍵テーブルの EUC ID エントリに格納される．プロセッサのキャッシュには EUC ID タグが追加されており，キャッシュへ命令，データの読み込み時に現在の EUC ID が書き込まれる．アクセス時に現在実行中の ECU ID とタグが比較され，キャッシュに対するアクセス制御がなされる．

前述の XOM では割り込みが発生するとレジスタごとに個別に暗号化していたが，L-MSP ではコンテキストをまとめてキャッシュに一時退避させ，まとめて暗号化することで遅延を緩和している．

XOM や L-MSP のようにデータの暗号化を行う場合，復号したデータを処理し，また暗号化して主記憶に書き戻す経路が存在することになる．プログラムをデータとして読み込むことができれば，この経路を利用して解読される危険性がある．

2.3.6 Security Enhanced MeP

Kawabata らは東芝が開発している 32 ビット CPU コア MeP(Media embedded Processor) にセキュリティ機能を追加する方法について提案している [24]．これは我々の提案システムと同様に命令のみの暗号化を行うもので，ソフトウェアのリバースエンジニアリングを防ぐことを目的としている．

特徴として，プロセッサの内部に命令とデータ用のローカルメモリを持っており，DMA コントローラーが主記憶からプログラムをローカルメモリにローディングする際に復号を行う．DMA コントローラーの制御レジスタには属性ビットフィールドと Key ID フィールドが追加されており，もし，属性ビットフィールドのビットが 1 であれば，プログラムは暗号化されている状態であるとして，DMA コントロー

ラーは，復号に用いる鍵の Key ID とプログラムを復号回路に送り，メモリコントローラーが復号結果をローカルメモリに格納する．ローカルメモリへのアクセスを制御するために，メモリコントローラーを追加しており，ローカルメモリが暗号化状態の場合は，命令フェッチユニットからのアクセスを除いて，DMA コントローラーのアクセスやロード/ストア命令によるアクセスができないようになっている．ただ，ローカルメモリにプログラムを転送する指示はユーザーが明示的に行うことになり，ユーザーの負担が大きいといえる．今後は，プロセッサ内部のデータ用のローカルメモリや，コンテキストを含むデータの暗号化にも対応していく予定としている．

第3章　提案システムの概要

本章では，仮想記憶を利用してページ単位で実行プログラムとプログラム鍵を対応付けるプロセッサアーキテクチャを提案する．3.2 節以降では，提案するプロセッサを用いたプログラム保護について説明する．

3.1　提案するプロセッサ

セキュアプロセッサでは，実行プログラムの暗号化を公開鍵暗号と共通鍵暗号を組み合わせて実現することで，ソフトウェア開発者は任意のプログラム鍵で実行プログラムを暗号化できる．したがって，マルチタスク環境下ではそれぞれ異なるプログラム鍵で暗号化された実行プログラムが起動されることになるため，プロセッサは起動中の実行プログラムのプログラム鍵はプロセッサ内部のレジスタスタック（以下，鍵テーブル）に保存し，実行プログラムを復号する際には鍵テーブルの中から対応するプログラム鍵を用意する必要がある．プログラム鍵の復号は実行プログラムの起動時に行われ，プログラムが終了するまでプログラム鍵は鍵テーブルに保存される．また，処理中の実行プログラムと，それを復号する鍵テーブル内のプログラム鍵との対応付けは，2.2.1 項で述べた SPID によって行われる．つまり，プロセッサは現在処理しているプロセスの SPID をインデックスとし，そのインデックスを参照して鍵テーブルから必要となるプログラム鍵を取り出す．

しかし， プログラム鍵を SPID と対応付けを行う場合，1 つのプロセスに対し 1 つの SPID が付加されているので，必然的に 1 つのプロセスは 1 つのプログラム鍵と対応付けられることになる．ここで，プロセスはシェアードライブラリのライブラリプログラムや，OS プログラムの一部など複数の実行プログラムを同一プロセス内に持つことがあるが，SPID だけの対応付けでは 1 つのプロセス内に複数のプログラム鍵を対応付けることができない．そこで本研究では，仮想記憶機構を利用してページ単位で実行プログラムとプログラム鍵を対応付けるプロセッサを提案する．プログラム鍵はページ単位で設定できるため，1 つのプロセスに複数のプログラム鍵を対応付けることができる．

本提案において，プロセッサ内部の鍵テーブルの管理には，実行プログラムの起動と終了の情報が必要となるため，OS が鍵テーブルの管理を行うが，OS が知ることができるのは，システム上に存在する実行プログラムのプログラム鍵がテー

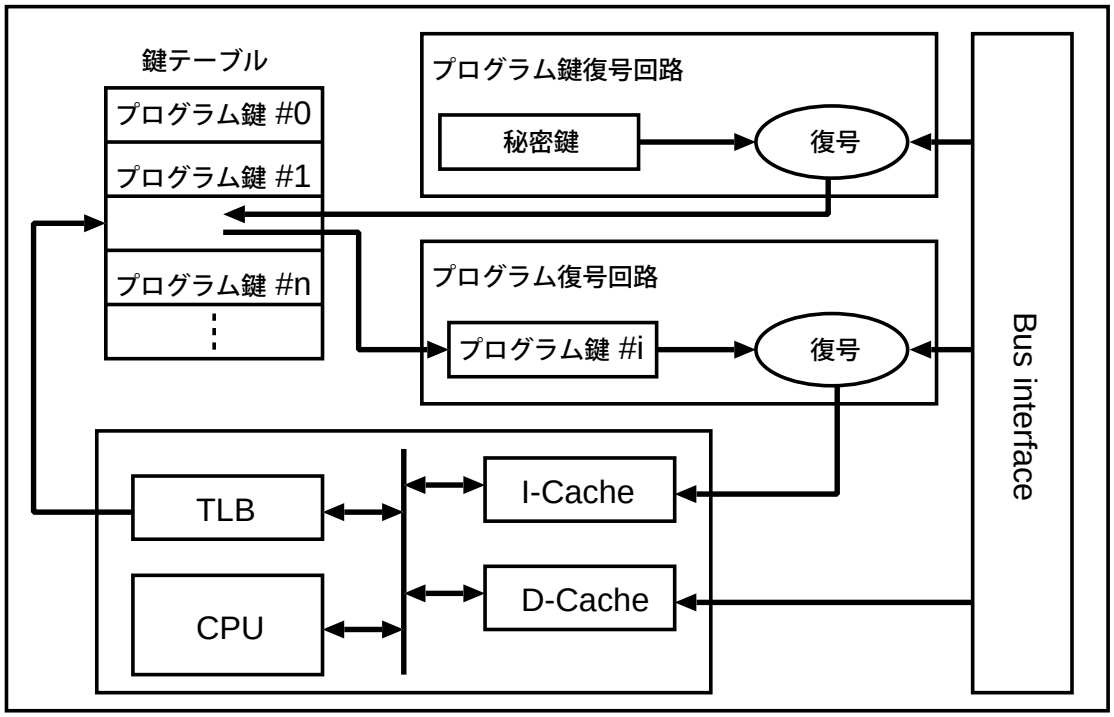

図 3.1: プロセッサのブロック図

ブルのどのインデックスに格納されているかという情報だけとなっている．詳しくは 3.2 節以降で説明するが，OS は鍵テーブル内のプログラム鍵の値自体は知ることができないため，OS が悪意のある第三者に乗っ取られた場合でも，保護すべきプログラムの秘密は守ることができる．本研究で提案するプロセッサのブロック図を図 3.1 に示す．

先行研究では，実行プログラムだけでなく，プログラムの処理対象となるデータの暗号化も行っている．本論文では，プログラム内の重要なアルゴリズムを保護するという観点に重点をおいて，実行プログラムのみの暗号化を行っており，プログラムの処理対象となるデータの暗号化は対象としていない．ただし，本提案システムでは，暗号化されたデータの復号をソフトウェアで処理することによってデータを秘密にしたい場合にも対応できると考えている．たとえば，有料のディジタルコンテンツを秘密のため暗号化してあるとし，その復号はプログラムで処理する．ここで，復号を行うプログラムを暗号化しておくことで，不正利用を狙うユーザからもディジタルコンテンツを保護することができると考えられる．また，2.1 節で紹介した OS による割り込みを用いて主記憶に退避されるコンテキストに対する攻撃において，入力データと出力データが観測されることで内部アルゴリズムを推定される危険性はあるが，保護するアルゴリズムが十分複雑であれば，入力データと出力データの組合せから内部アルゴリズムを同定することは十分に困難であろうと考えている．

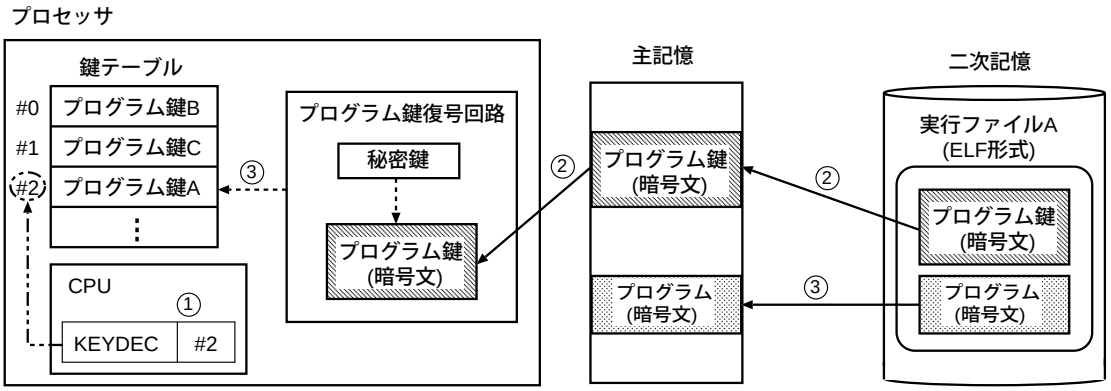

図 3.2: プログラム鍵の復号

3.2 プログラム鍵の復号

本節では，プログラム起動時に行われるプログラム鍵の復号について説明する．公開鍵暗号によって暗号化されたプログラム鍵は，ELF (Executable and Linking Format) 形式のような実行ファイルのヘッダ情報内の新たなセグメントに加えられるものとし，プログラム鍵によって暗号化された実行プログラムと共に 1 つの実行ファイルとして二次記憶上に格納されている．

プログラムが起動されると，実行プログラムが二次記憶から主記憶へローディングされるが，OS は最初にプログラム鍵をローディングし，暗号化されたプログラム鍵の復号を行う命令 (KEYDEC) を発行する．ここで，OS は KEYDEC 命令によって，復号したプログラム鍵を格納する鍵テーブルのインデックスを指定する．その後，実行プログラムのローディング処理に並行して，プロセッサはプロセッサ内部の復号回路でプログラム鍵の復号を行い，KEYDEC 命令によって指定された鍵テーブルのインデックスへプログラム鍵を格納する．鍵テーブルとその管理については，3.3.1 で詳しく説明する．プログラム鍵を復号する様子を図 3.2 に示す．

3.3 実行プログラムの復号

3.3.1 鍵テーブルについて

3.2 節で，復号したプログラム鍵はプロセッサ内の鍵テーブルに格納されると述べたが，鍵テーブルとその管理には，以下に示す点について考慮する必要がある．

1. 鍵テーブルのサイズ
2. 実行プログラムを復号する時に，どのプログラム鍵を用いて復号すべきか選択する手段

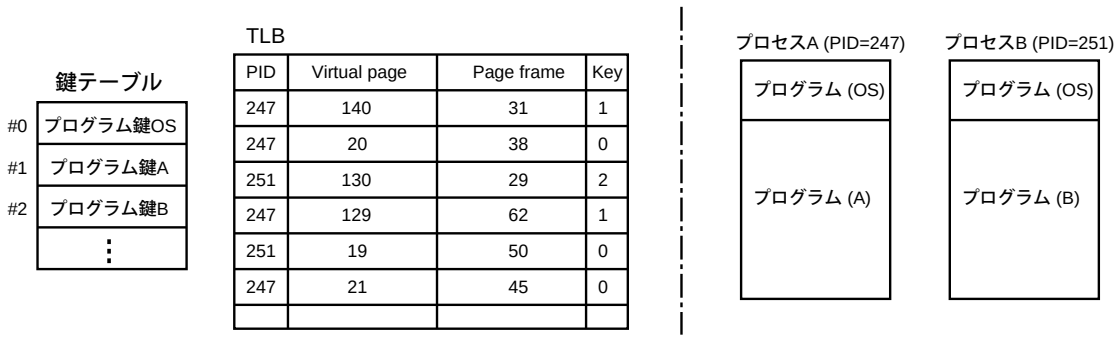

図 3.3: ページ単位のプログラム鍵の対応付け

1. は鍵テーブルはその個数が有限であるので，プログラムの起動や終了に同期して入れ替えを行う必要がある．これは，ハードウェアでは知ることができないので，OS の助けが必要である．また，鍵テーブル実現のためのハードウェア量の削減のため，復号したプログラム鍵の鍵テーブルへの割り付け管理も OS が行うことにした．OS はプログラム鍵の復号時に格納する鍵テーブルのインデックスを KEYDEC 命令によって指定する．ここで，鍵テーブルに格納できるプログラム鍵の個数が問題となるが，同時に実行状態となるプログラムは，高々十数個であろうと想定している．起動中のプログラムが鍵テーブルのサイズを超えた時には，OS は LRU などのアルゴリズムを用いて鍵テーブル内のプログラム鍵の入れ換えを行わなければならない．鍵テーブルからあふれたプログラム鍵のプログラムを再び実行する際には，再度プログラム鍵を復号する必要がでてくる．鍵テーブルのサイズに対しての検討は，今後の課題としている．

2. において，起動中の実行プログラムが主記憶上のどの領域を占めるかは OS のみが知り得る情報である．実行プログラムが占める実アドレス領域とそれを復号するプログラム鍵の対応関係表をプロセッサ内にハードウェアとして持つ場合ハードウェア量が増大するので，本研究では，この対応付けに仮想記憶の枠組みを利用することとした．すなわち，OS がアドレス変換表にページ単位で復号に使うプログラム鍵のインデックスを格納し，プロセッサ内の TLB (Translation Look-aside Buffer) にも同じように変換対とプログラム鍵のインデックスが登録される．プロセッサは TLB を用いたアドレス変換の際に，そのページの命令に対応するプログラム鍵のインデックスを知ることができる．これにより，同一ページ内に複数のプログラム鍵の異なる実行プログラムをローディングすることはできないが，大きな制約とはならないと考えている．また，TLB にプログラム鍵のインデックスを格納する欄が増えることになるが，TLB の各エントリに対して 4, 5 ビット程度の増加になると考えられるため，プログラム鍵のインデックスの追加によるハードウェア量の増加は十分実現の範囲内であると考えられる．仮想記憶を用いたページ単位のプログラム鍵の対応付けのイメージを図 3.3 に示す．

3.3.2 プログラム実行時の動作

実行プログラムがローディングされ実行されるまでの動作を，ハードウェアテーブルウォーク型の仮想記憶システムを例にとって具体的に説明する．なお，ソフトウェアテーブルウォーク型の TLB の場合でもほぼ同様に実現できる．最初に二次記憶上の実行プログラム内から暗号化されたプログラム鍵を実記憶上の特定のアドレスに格納する．実行プログラムが二次記憶から実記憶にローディングされている間に，プロセッサは KEYDEC 命令によって復号回路でプログラム鍵の復号を行う．復号されたプログラム鍵は，KEYDEC 命令内で指定される鍵テーブルのインデックスに格納される．実行プログラムがローディングされ，OS が仮想アドレスと実アドレスの変換対をアドレス変換表に登録する際に，その実行プログラムを復号するプログラム鍵の鍵テーブル上のインデックスも共に登録する．実行プログラムが終了すると OS は，対応するプログラム鍵のインデックスを開放する．

プロセッサが最初に実行プログラムを処理する時には，TLB に変換対がまだ登録されていないので，TLB ミスが起こる．そこで新たな変換対をアドレス変換表を参照し登録する．その際に，TLB に登録する実ページの実行プログラムに対応するプログラム鍵のインデックスも共に登録される．次に TLB によるアドレス変換を行った時には，プロセッサは実行プログラムを復号するために必要なプログラム鍵のインデックスを知ることができる．その後キャッシュミスが起こった場合には，TLB で得られるプログラム鍵のインデックスをもとに鍵テーブルを参照し該当するプログラム鍵を取り出す．そのプログラム鍵を用いて主記憶から読み込まれる暗号化された命令に対して復号を行い，命令キャッシュに格納する．以上のキャッシュミス時にプロセッサ内で行われる処理を図 3.4 に示す．キャッシュにヒットした場合は，既に復号された命令がキャッシュ上に存在しているので，復号処理は必要ない．

3.4 動的な命令書き換えへの対策

本研究のようにプログラムが暗号化されている場合，命令の書き換えが行われると，書き換える命令の暗号化が必要となるため，対応を考えなくてはならない．命令書き換えが必要となるケースは，シェアードライブラリなどの動的リンクの場合である．すなわち，多くのアプリケーションプログラムでは，シェアードライブラリと呼ばれる，多くのプログラムで利用されるような汎用的な機能をまとめたライブラリをプログラム実行時にリンクする場合がある．シェアードライブラリの動的リンクは，プログラム中のシステムコールによって割り込みを発生させ，主記憶上のシェアードライブラリの対応エントリアドレスを得て，そのアドレスへの分岐命令に置き換えることによって実現される．本節では，命令の書き換えを行わずにシェアードライブラリの機能を用いる手法について述べる．

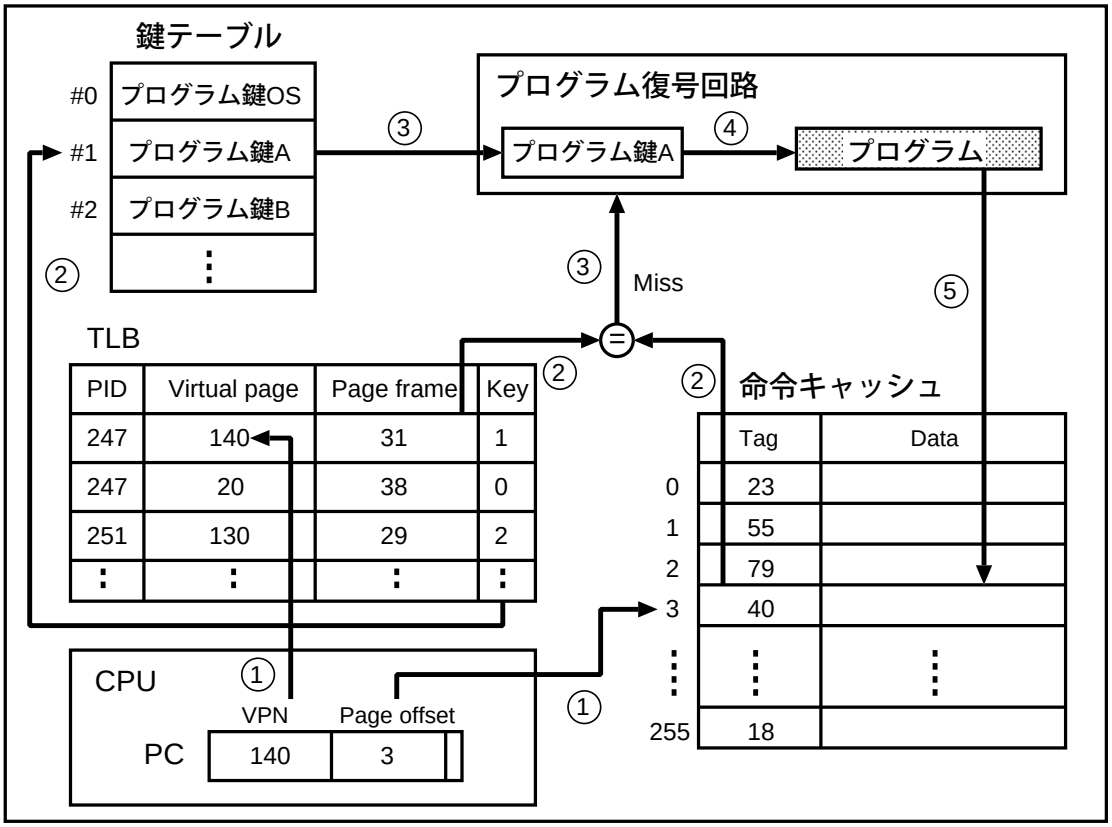

図 3.4: キャッシュミス時のプロセッサ内部処理

まず，単純な手法としては，必要となるシェアードライブラリの機能をリンクしてから，コンパイルして実行プログラムを作り，暗号化する方法が挙げられる．このように静的なリンクによる手法では，動的なリンクのメリットであった実行プログラムのサイズの縮小といった効果は得られなくなるが，簡単に実現できるため，リンクするプログラムサイズが小さい場合には有効である．

次に挙げる手法は，動的リンク時に割り込みを発生させるシステムコール命令を，条件によっては分岐命令の機能を持たせた命令（以下，SWI-B 命令）として実装する方法である．SWI-B 命令は，デコード時にオペランドを参照する．オペランドには，命令の機能を決定するフラグを用意し，その内容によって割り込みと分岐の機能を使い分ける仕様にする．SWI-B 命令を用いた動的なリンクを具体的に説明すると，まず，最初に SWI-B 命令がフェッチされる際にはプログラムのリンクを行う必要がある．SWI-B 命令を割り込みを発生させる命令とするため，該当するオペランドにフラグを設定する．その後 SWI-B 命令がフェッチされ，デコードの際にはオペランドを参照し，割り込み命令として機能することになる．それ以降 SWI-B 命令がフェッチされる際には，必要とするプログラムは既にリンクされているので，オペランドには SWI-B 命令を分岐命令とするフラグを設定する．

したがって，SWI-B 命令は分岐命令として機能する．この手法では，以上の機構を実現するためのハードウェアが必要となるが，従来通りの動的リンクが行うことができる．

3.5 プロセッサ起動時の動作

本研究では，実行プログラムは全て暗号化し，平文のプログラムは一切処理しない．その理由は，平文のプログラムと暗号文のプログラムが混載するシステム上では，プロセッサにそれぞれを処理するモードを設ける必要があり，そこがセキュリティホールになりうると考えたからである．この場合，プロセッサ起動時に処理される boot プログラムも暗号化しておく必要がある．本研究では実行プログラムとプログラム鍵との対応付けに仮想記憶を利用しているので，OS が仮想記憶モードに移行するまでの実行プログラムについては 1 つのプログラム鍵で処理されることとした．まず，プロセッサは起動時に自動的にある値に設定されるプログラム鍵 (boot_ Key) を持ち，プロセッサ起動時に処理される boot プログラムなどは boot_ Key を用いて暗号化する．boot プログラムが処理された後，boot プログラム内の KEYDEC 命令によって，OS の初期化プログラムのプログラム鍵が復号され，仮想記憶モードに移行するまでのプログラムが処理される．ただし，仮想記憶モードに移行する前でも，実アドレスでも TLB をアクセスするようにしておき，TLB にアドレスとプログラム鍵のインデックスを登録すれば，複数の実行プログラムに対応することも可能であると考えられる．

3.6 オブジェクトファイルの形式

提案システムにおいてベンダは作成したプログラムを暗号化し，配布する．つまり，プログラム（ソースファイル）をコンパイルし，出力されたオブジェクトファイルを暗号化する必要がある．以下，オブジェクトファイルの形式と暗号化の方法について述べる．

提案システムでは，オブジェクトファイルの形式にフリーウェアの Linux や UNIX でも採用されている ELF(Executable and Linking Format) 形式を想定している [19, 20]．ELF は伝統的な a.out 形式では対応が難しいクロスコンパイルや動的リンクをサポートしたものである．ELF ファイルには主に次の 3 種類が存在する．

- 再配置可能ファイル：ほかのオブジェクトファイルとリンクして，実行可能ファイル，共有オブジェクトファイル，または別の再配置可能オブジェクトファイルを作成するファイル（拡張子.o）
- 実行可能ファイル：再配置がすべて完了し，実行が可能なファイル

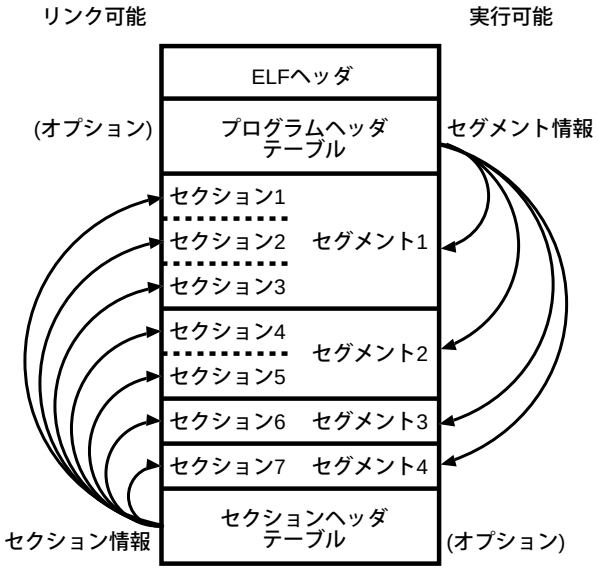

図 3.5: ELF ファイル形式

- 共有オブジェクトファイル：リンカ用のシンボル情報と実行時に使われるコードを格納する，共有ライブラリ等のファイル（拡張子.so）

リンク時と実行時では，**図 3.5** のように，ELF ファイルを異なる構造として解釈する．

ELF ファイルの先頭には必ず ELF ヘッダが存在し，ファイル全体の編成を記述している．プログラムヘッダテーブルは，ELF ファイルが実行可能ファイルの場合に必要なもので，プログラムをメモリにマッピングする際のセグメントを定義している．また，セクションヘッダテーブルは，コンパイラ，リンカが扱う論理単位であるセクションを定義している．１つのセグメントは１つ，または複数のセクションの集合であり，セクションが ELF ファイル内で処理できる最小単位であるのに対し，セグメントはローダが主記憶へのマップを行う際の処理単位である．

コンパイラやアセンブラが作成したバイナリの命令は ELF ファイルの.text，.init，.fini セクションに配置される．.text セクションはプログラム本体の命令コードを保持し，.init，.fini セクションはプログラムの起動時と終了時に実行する命令コードを保持している．この２つのセクションは C では使われないが，C++におけるグローバルコンストラクタやデストラクタの実行に用いられる．

提案システムにおけるプログラムの暗号化の手順を**図 3.6** に示す．以下，図 3.6 に従って説明する．

まず，.text，.init，.fini セクションの位置をセクションヘッダテーブルから読み取り，それぞれプログラム鍵で暗号化する（共通鍵暗号方式）．次に，暗号化に使用したプログラム鍵をセキュアプロセッサの公開鍵で暗号化する（公開鍵暗号方式）．暗号化されたプログラム鍵は ELF ファイル内に新たな.key セクションを設

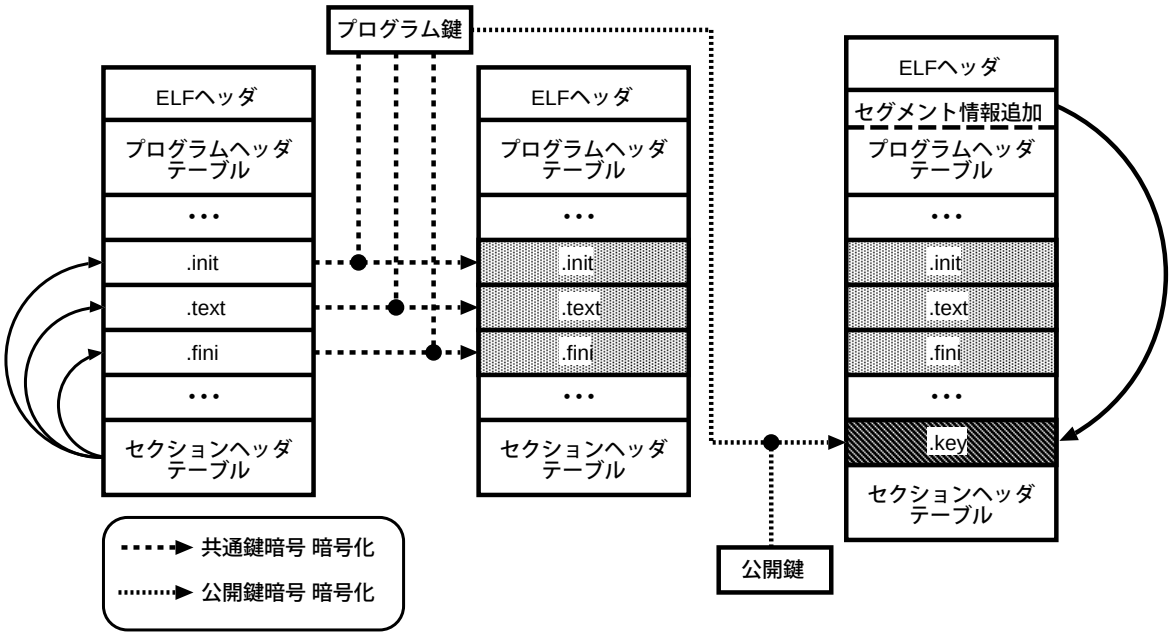

図 3.6: ELF ファイルの暗号化

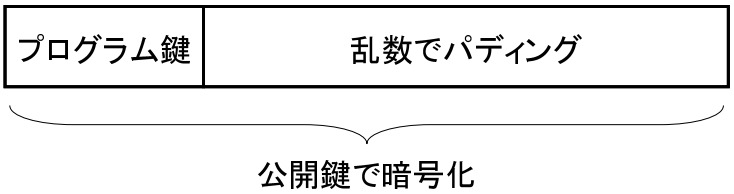

図 3.7: プログラム鍵のパディング

けて，そこに配置する．また，ローダが主記憶に暗号化されたプログラム鍵をマッピングできるように，.key セクションをセグメントとして，プログラムヘッダテーブルに追加することも必要である．以上のようにして，ELF ファイルを暗号化することで，ハイブリッド方式に対応できると考えている．

プログラム鍵の暗号化において，プログラム鍵が同じ場合は，当然同じ暗号文に暗号化される．これはセキュリティ上好ましくない．実際，共通鍵を秘匿化して通信するのに RSA 暗号を使用していた SSL [1]に，約 100 万個の命令を送った反応を解析することで，共通鍵を解読できる可能性があることが判明している．そこで，**図 3.7** に示すように暗号化するプログラム鍵の右側には乱数を連結する．これによって，毎回異なる暗号文が生成されることになり，強度が増す．ただし，パディングした値は公開鍵の法の値より小さくなければならない．そのため，パディングした結果のビット数を法のビット数より 1 つ少ない程度にすればよい．

[1]SSL(Secure Sockets Layer)．Netscape 社によって提唱されている暗号化プロトコル．

第4章　暗号方式の調査

4.1　共通鍵暗号

本節では，広く社会に用いられている共通鍵暗号である DES (Data Encryption Standard)[8] と AES (Advanced Encryption Standard)[9] を取り上げ，それぞれの暗号方式の調査を行い，復号回路の設計を行う．

本研究で共通鍵暗号はプログラムの暗号化に用いられ，その復号はキャッシュミス時に，主記憶からキャッシュラインサイズの暗号化されたプログラム（命令）を命令キャッシュに転送する過程で行われるので，復号回路のレイテンシは命令キャッシュのミスペナルティの増加として性能に影響を与える．一般的に命令キャッシュのヒット率は非常に高くキャッシュミスペナルティの増加が 5 割程度であれば性能に対して与える影響も大きくは無いと考え，今回の設計において復号回路のレイテンシは，一般的な組み込みプロセッサのキャッシュミスペナルティを 25 サイクル前後と予想し，10〜15 サイクルを目標とした．設計した復号回路が性能に与える影響は，5.2 節で詳しく調査する．

本節で取り上げる共通鍵暗号の DES と AES は，共にブロック暗号[1]と呼ばれる暗号方式に分類される．ここで，キャッシュミス時に復号を行う際に，復号回路のスループットに比べ主記憶からのデータの転送速度の方が速い場合，後続のデータは先行するデータの復号処理を待つ必要があり，結果的にデータの転送能力を低下させることになる．この解決として，パイプラインで復号処理を行う回路を設計する手法も考えられるが，大幅な回路面積の増加を招くため，今回の設計では，復号回路のスループットよりも回路面積を優先し，1 ブロックサイズ毎に復号処理を行う回路とした．また，復号回路をプロセッサに実装することによる回路面積の増加は，1 割程度に抑えることを目標としている．

4.1.1　DES

DES は， IBM 社の研究者 Feistel によって設計され，1977 年に NBS (アメリカ商務省標準局＝現 NIST) によって，米国政府標準暗号として採用された暗号方式である．事実上の世界標準暗号として広く使用されていたが，暗号化に用いる鍵の長さが 56 ビットであるため，計算機の処理能力の向上にともない，近年では全

[1]決められたブロックサイズのデータに対して暗号化/復号を行う

数探索法[2]に対する安全強度が十分とは言えない．そのため現在では，DES を 3 回繰り返すことにより全数探索法への安全強度を高めた Triple-DES が用いられている．Triple-DES の暗号化では，2 つの共通鍵 (Key1, Key2) を用いる E-D-E 方式と 3 つの異なる共通鍵 (Key1, Key2, Key3) を用いる E-E-E 方式が存在する．E-D-E 方式では，Key1 での DES 暗号化を行い，次に Key2 で復号し，最後に Key1 で再び暗号化を行う．E-D-E 方式は Key1=Key2 の場合に，従来通り DES 一回で暗号化/復号する Single-DES として利用することができるが，本論文では，プログラムの暗号化に安全性の低い Single-DES を用いることはないと考え，より安全強度の高い E-E-E 方式の Triple-DES について評価を行う．

Single-DES には，1993 年に三菱電機の松井充によって線形解読法と呼ばれる有効な解読法が発見されている．線形解読法は既知平文攻撃[3]の一種であり，2^{43} 個の平文と暗号文の組みが必要となるが，2^{42} 回の DES 暗号化に相当する処理時間で解読が可能である．しかし，Triple-DES では，Single-DES に比べ暗号化処理のラウンド数が多いため，より多くの平文と暗号文の組みが必要となり有効な解読法とはならないと考えている．

4.1.1.1 暗号化/復号のアルゴリズム

Triple-DES は Single-DES を 3 回繰り返すことで実現されるので，Single-DES の暗号化/復号のアルゴリズムについて説明する．DES には，64 ビット長の鍵[4]が用いられ，64 ビットのブロック長のデータに対して暗号化/復号処理が行われる．暗号化処理のアルゴリズムを図 4.1 に示し，以下図 4.1 にしたがって説明を行う．

まず，入力される 64 ビットのブロックに対して，ビット単位の置換処理 (IP) が行われる．IP 後，ブロックは F 関数とよばれる処理と排他的論理和演算 (XOR) を 16 回繰り返すことで暗号化され，ブロックの暗号結果を出力する前に，再びビット単位の置換処理 (IP^{-1}) が行われる．

F 関数は 32 ビットと 48 ビットの 2 つの入力データに対して操作を行い，32 ビットの結果を出力する．32 ビットの入力は，64 ビットのブロックのうちの右 32 ビットであり，48 ビットの入力は，後述する鍵拡大処理によって 64 ビットの鍵から生成されるラウンド鍵 ($K_{1...16}$) である．F 関数の概略図を図 4.2 に示す．

F 関数では，32 ビットのデータをビット選択表 E によって，48 ビットのデータに変換し，入力された 48 ビットのデータと XOR を行う．次に，XOR の出力 48 ビットを 6 ビット ×8 に分割し，それぞれの 6 ビットの値を変換表 ($S_{1...8}$) によって 4 ビットに変換する．最後に，4 ビット ×8 のデータをまとめ，それに対しビット単位の置換処理 (P) を施し，結果が出力される．暗号化処理における $S_{1...8}$ のような変換表は，一般的に S-Box と呼ばれ，非線形性の強い数表となっている．

---

[2]全ての鍵を総当たり的に試す方法．56 ビットの鍵の場合，2^{56} 個の鍵を試すこととなる．

[3]既知の平文と対応する暗号文のペアを得られる条件で行う攻撃

[4]64 ビットの鍵うち 8 ビットはパリティビットで，実質的な鍵の長さは 56 ビットとなっている

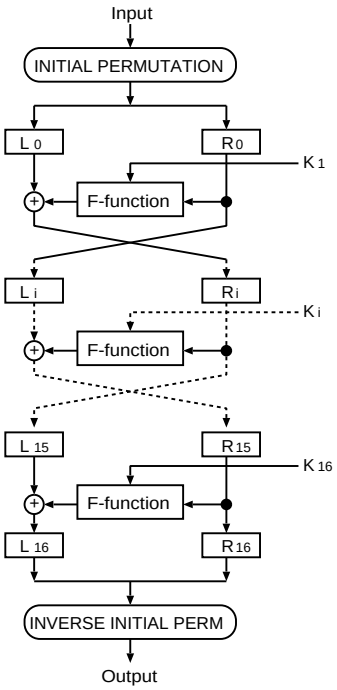

図 4.1: DES 暗号化のアルゴリズム

F 関数の入力の 1 つであるラウンド鍵は，64 ビットの鍵から鍵拡大処理によって生成される．DES 暗号化/復号は F 関数による処理を 16 回行うので，鍵拡大処理によってラウンド鍵を 16 個生成する．DES の鍵拡大処理の概略図を図 4.3 に示す．

鍵拡大処理では，最初に 64 ビットの鍵からパリティビットである 8 ビットの除去と置換処理 ($PC-1$) を行う．置換処理を行った 56 ビットのデータは，2 つに分割され (C_0, D_0)，それぞれシフト操作が行われる．そして，C_0 と D_0 は再び結合させ，56 ビットから 48 ビットへの選択置換処理 ($PC-2$) を行うことによってラウンド鍵 K_1 が生成される．以降シフト操作と選択置換処理を繰り返しラウンド鍵を 16 個生成する．

以上，Single-DES 暗号化処理のアルゴリズムについて説明したが，復号処理のアルゴリズムも基本的に暗号化処理とほぼ同様である．復号処理では，入力ブロック（暗号文）に対し IP^{-1} を行った後，16 回の F 関数と XOR を実行し，最後に IP を行えば出力（平文）が得られる．ただし，F 関数に用いるラウンド鍵は暗号化とは逆の順序となり，1 ラウンド目の F 関数で K_{16} を用い，16 ラウンド目の F 関数で K_1 を用いる．

4.1.1.2 復号回路の設計

Single-DES は，F 関数と XOR を 1 ラウンドの処理とした場合，復号には 16 ラウンドの処理が行われるので，Triple-DES 復号は 16×3=48 ラウンドの処理が行わ

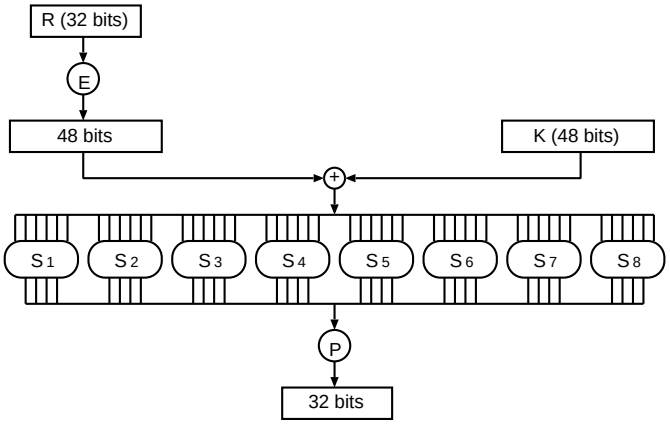

図 4.2: F 関数の概略図

れる．図 4.4 に，1 サイクルで 1 ラウンドの復号処理を行う回路 (Des_dec1R) のブロック図を示す．図 4.4 の回路を用いると，64 ビットの暗号文を 48 サイクルかけて復号処理を行うこととなる．復号回路において，1 サイクルで処理するラウンド数を増やせば復号の処理時間は短縮できるが，遅延時間と回路面積も増加する．4.1.3 項では，1 サイクルで処理するラウンド数を変化させた復号回路について評価を行う．

また，復号処理にはラウンド鍵を用いるためラウンド鍵を生成する鍵拡大処理回路と，復号回路に処理するラウンドに応じたラウンド鍵を提供する鍵スケジュール回路も設計する．鍵拡大処理回路では，3 回の DES それぞれの鍵から，48 ラウンドの Triple-DES 復号処理のラウンド鍵を生成し，生成した鍵は鍵スケジュール回路に格納する．

4.1.2 AES

1990 年代後半，それまで事実上の世界標準暗号であった DES は，計算機の処理能力の向上により安全性に限界がきていた．そこで 1997 年，NIST (アメリカ商務省標準技術協会) は，AES (Advanced Encryption Standard) と呼ぶ新たな米国政府標準暗号となる暗号化アルゴリズムを公募し，ベルギーの数学者 Joan Daemen と，レーベン・カトリック大学の Vincent Rijmen が設計した「Rijndael」が採用された．

Rijndael において暗号化/復号に用いる鍵の長さ N_k は，DES が 64 ビットであったのに対し，128, 192 または 256 ビットから選択できるようになっている．また，暗号化/復号を行うブロックサイズ N_b に関しても 128, 192 または 256 ビットから選択できるが，AES の規格では，ブロックサイズは 128 ビットに固定されている．AES 暗号化/復号も DES と同じように，ある決められた処理を繰り返すことで実現されるが，その繰り返し回数 N_r は N_b と N_k によって表 4.1 に示すように変化する．N_k は長いほど全数探索法に対して安全強度が高くなるが，N_k を長くすると N_r も

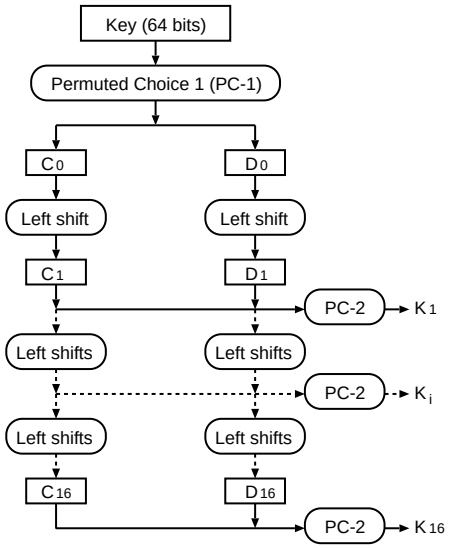

図 4.3: DES の鍵拡大処理の概略図

表 4.1: 各 N_b, N_k においての N_r

N_r	$N_b = 128$	$N_b = 196$	$N_b = 256$
$N_b = 128$	10	12	14
$N_b = 196$	12	12	14
$N_b = 256$	14	14	14

増加し，復号の処理時間が増加するので，本論文では，N_b と N_k 共に 128 ビットの AES (AES-128) について評価を行う．現在 AES において，現実的な時間で解読できるような効果的な解読法は発見されておらず，N_k が 128 ビットの場合でも全数探索法に対して十分な安全強度があると考えられている．

4.1.2.1 暗号化/復号のアルゴリズム

AES-128 の暗号化は，AddRoundKey, ShiftRows, SubBytes, MixColumns と呼ばれる処理を 1 ラウンドとし，それを 10 回繰り返すことで実現される．それに対し復号は，AddRoundKey, InvShiftRows, InvSubBytes, InvMixColumns の暗号化で行った処理の逆関数となる処理を 1 つのラウンドの中で行う．また AES-128 では，AddRoundKey の処理にてラウンド鍵を用いるので，鍵拡大処理によってラウンド鍵を生成する必要がある．生成されるラウンド鍵の長さは 128 ビットであり，AES-128 の場合，$N_r + 1 = 11$ 個のラウンド鍵を生成する．AES-128 の暗号化/復号のアルゴリズムを図 4.5 に示し，以下図 4.5 にしたがって説明を行う．

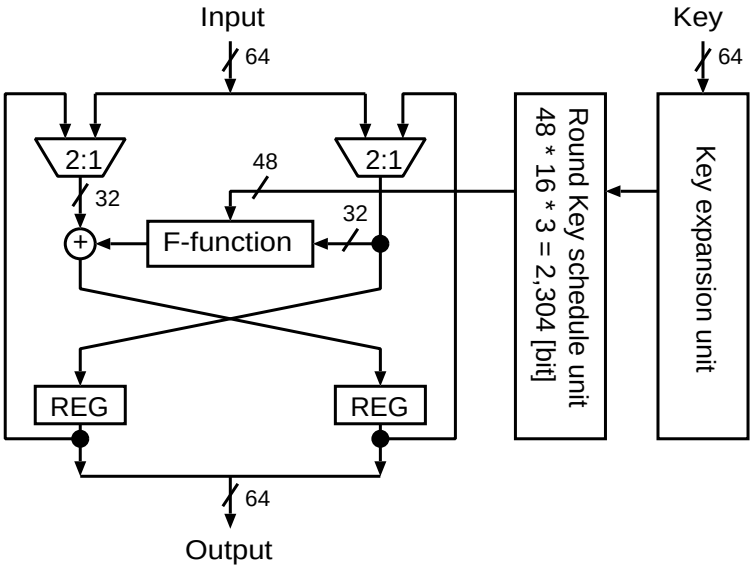

図 4.4: DESdec_1R のブロック図

4.1.2.1.1 State の作成 AES-128 では，128 ビットのブロック長のデータに対して暗号化/復号処理が行われる．また内部では，1 バイト単位で処理が行われるため，入力ブロックを 1 バイト毎に分割する．分割されたブロックは，State と呼ばれる 4 行の配列に格納される．State の模式図を図 4.6 に示す.

4.1.2.1.2 AddRoundKey AddRoundKey では，State とラウンド鍵の XOR を行う．ラウンド鍵も入力ブロックと同様に，1 バイト毎に分割される．AddRoundKey の模式図を図 4.7 に示す.

4.1.2.1.3 SubBytes/InvSubBytes SubBytes/InvSubBytes では，State の各要素に対して変換処理を行う．この変換処理は，数学的にはアファイン変換（$X = a \times x + b$）と 2^8 のガロア体 $GF(2^8)$ 上での逆数計算で定義されている．アファイン変換は式 4.1 で表現される行列変換式となり，逆数計算は，8 次の規約多項式 ($m(x) = x^8 + x^4 + x^3 + x + 1$) での逆数計算となる．また，上記の変換処理は S-Box による変換表で表現することができる．SubBytes/InvSubBytes の模式図を図 4.8 に示す.

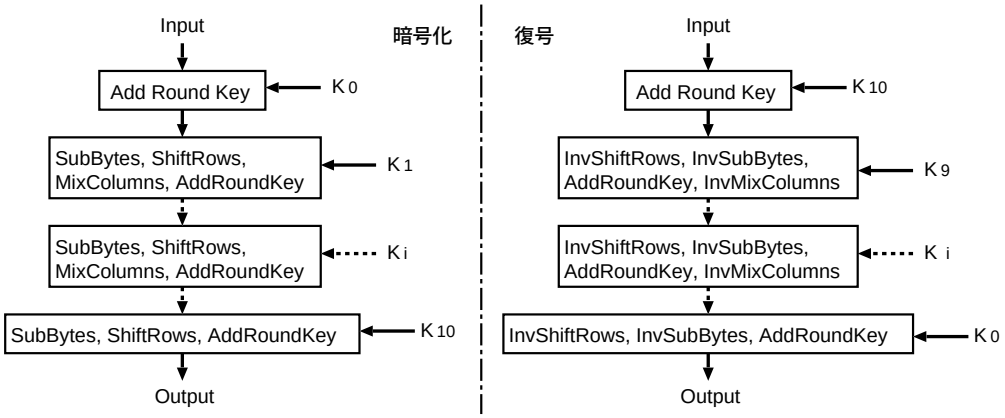

図 4.5: AES-128 の暗号化/復号のアルゴリズム（仮）

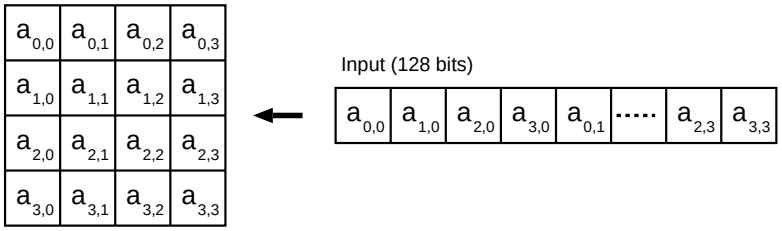

図 4.6: State の模式図

$$\begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{4.1}$$

4.1.2.1.4 ShiftRows/InvShiftRows ShiftRows では，State の第 i 行目の要素を左へ i ローテートシフト，InvShiftRows では，State の第 i 行目の要素を右へ i ローテートシフトを行う．ShiftRows/InvShiftRows の模式図を図に 4.9 示す.

4.1.2.1.5 MixColumns/InvMixColumns MixColumns では，State の各列を 4 つの項からなる多項式とみなし，$x^4 + 1$ を法とした世界で $c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ との乗算を行う．乗算は，式 4.2 で表される行列演算で表現され，具

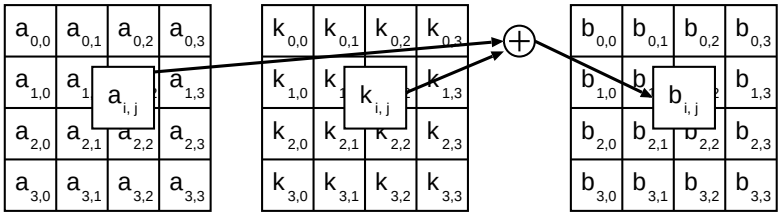

図 4.7: AddRoundKey の模式図

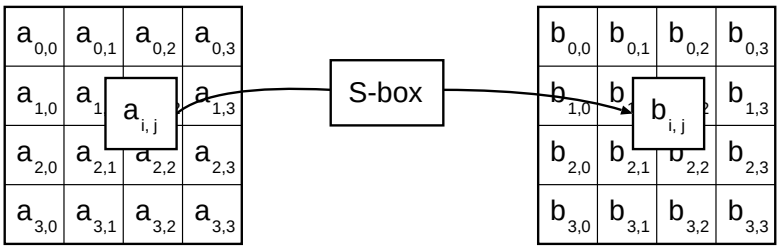

図 4.8: SubBytes と InvSubBytes の模式図

体的な演算は式 4.3 のようになる．ここで，⊕ は XOR，• は $GF(2^8)$ 上での乗算であり 8 次の規約多項式 ($m(x) = x^8 + x^4 + x^3 + x + 1$) を法とした多項式の乗算と一致する．

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} for\ 0 \le i \le 3 \tag{4.2}$$

$$\begin{aligned} b_{0,i} &= (\{02\} \bullet a_{0,i}) \oplus (\{03\} \bullet a_{1,i}) \oplus a_{2,i} \oplus a_{3,i} \\ b_{1,i} &= a_{0,i} \oplus (\{02\} \bullet a_{1,i}) \oplus (\{03\} \bullet a_{2,i}) \oplus a_{3,i} \\ b_{2,i} &= a_{0,i} \oplus a_{1,i} \oplus (\{02\} \bullet a_{2,i}) \oplus (\{03\} \bullet a_{3,i}) \\ b_{3,i} &= (\{03\} \bullet a_{0,i}) \oplus a_{1,i} \oplus a_{2,i} \oplus (\{02\} \bullet a_{3,i}) \end{aligned} \tag{4.3}$$

InvMixColumns では，State と $c^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$ との乗算を行う．MixColumns/InvMixColumns の模式図を図に 4.10 示す．

4.1.2.1.6　鍵拡大処理　AES-128 の鍵拡大処理のアルゴリズムを図 4.11 に示し，以下図 4.11 にしたがって説明を行う．まず最初に，入力された鍵と等しいラウンド鍵 K_0 が生成され，128 ビットの鍵が 1 ワード毎に分割される (Key=$[W_0, W_1, W_2, W_3]$)．次に RotWord で，W_3 をバイト単位で左ローテートシフトを行う．RotWord の結果

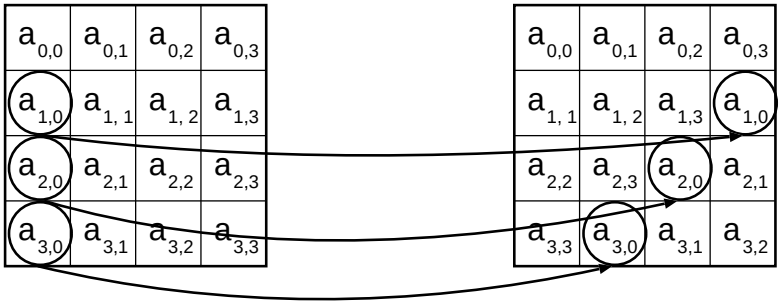

図 4.9: Shift Rows と InvShiftRows の模式図

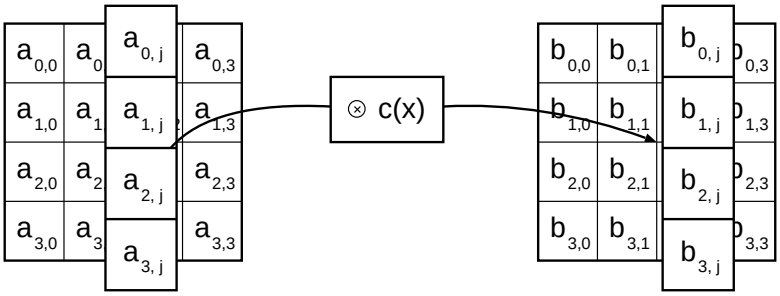

図 4.10: Mix Columns と InvMixColumns の模式図

は，SubWord によって変換処理が行う．SubWord は SubBytes と全く同様の変換処理である．SubBytes の結果は，W_0 と Rcon[1] の XOR の結果と再び XOR を行われ，その結果 W'_0 を得る．Rcon[i] の値は $[x^{i-1}, \{00\}, \{00\}, \{00\}]$ として与えられ，x^{i-1} は $GF(2^8)$ 上での {02} の累乗を意味している．W'_0 と W_1 の XOR から W'_1 を，W'_1 と W_2 の XOR から W'_2 を，W'_2 と W_3 の XOR から W'_3 をそれぞれ得る．$[W'_0, W'_1, W'_2, W'_3]$ は，ラウンド鍵 K_1 として生成される．以上の処理を繰り返すことで，10 個のラウンド鍵 $K_{1\cdots10}$ を生成する．

4.1.2.2　復号回路の設計

AES-128 では DES に比べ 1 ラウンドの処理が複雑なので，1 サイクルで複数回のラウンドを処理する回路を設計すると遅延時間と回路面積が大幅に増大すると考えたため，1 サイクルで 1 ラウンドの復号処理を行う回路を設計する．AES-128 の Nr は 10 回なので，復号回路の処理時間は 10 サイクルとなる．また，鍵拡大処理についても同様に 1 サイクルで 1 個のラウンド鍵が生成される回路を設計する．鍵拡大処理回路は 10 サイクルで 11 個のラウンド鍵を生成し，鍵スケジュール回路に格納する．図 4.12 に，設計する AES-128 復号回路，鍵拡大処理回路，鍵スケジュール回路のブロック図を示す．

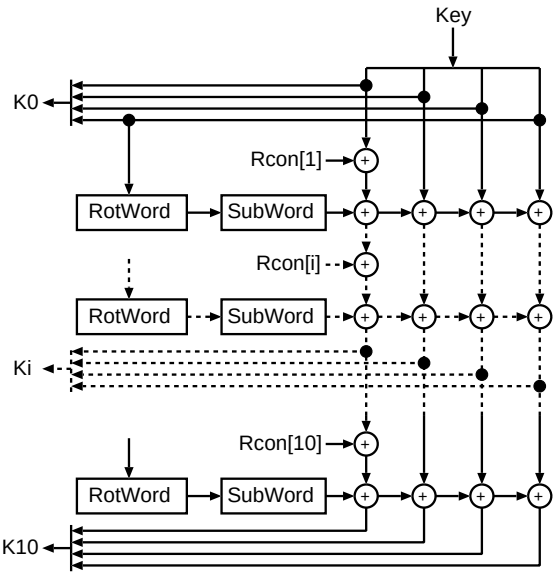

図 4.11: AES-128 の鍵拡大処理のアルゴリズム

4.1.3 復号回路の評価

本項では，Triple-DES と AES-128 の復号回路の評価を行う．図 4.4 と図 4.12 の回路を論理合成し回路面積を求めた．回路面積とその回路を用いた場合の復号処理時間について評価を行い，本研究でプログラムの暗号化に用いる共通鍵暗号として最適な復号回路を調査した．

4.1.3.1 論理合成結果と評価

回路の設計記述には Verilog-HDL を用いた．設計した回路は，ケイデンス社の Verilog-XL によって論理シミュレーションを行い，Synopsys 社の Design Compiler によって論理合成を行った．論理合成には HITACHI 0.18μm プロセス用に京都大学で作成されたスタンダードセルライブラリを使用した．

まず Triple-DES において，1 サイクルに 1 ラウンドを処理する図 4.4 を基本とし，1 サイクルに処理できるラウンド数を 2, 3, 4 とした回路を設計した．表 4.2 に，各回路の論理合成結果を示す．論理合成時の遅延制約には，0.18μm プロセスで設計されたプロセッサである ARM 社の ARM922T (動作周波数 200 MHz) を参考にし，5.0ns の遅延制約を設定した [30]．表 4.2 中の復号処理時間は，その回路を用いた場合に Triple-DES 復号に要する処理時間である．

表 4.2 で評価した全ての復号回路において，S-Box での処理時間が全体の処理時間の約半分を占めている．本研究の評価環境において，S-Box には最低でも 0.85ns 程度の処理時間を要する．DES_dec4R では 4 回の S-Box での処理が存在するため，5.0ns の遅延制約を満たせず，遅延時間は 6.24ns となった．

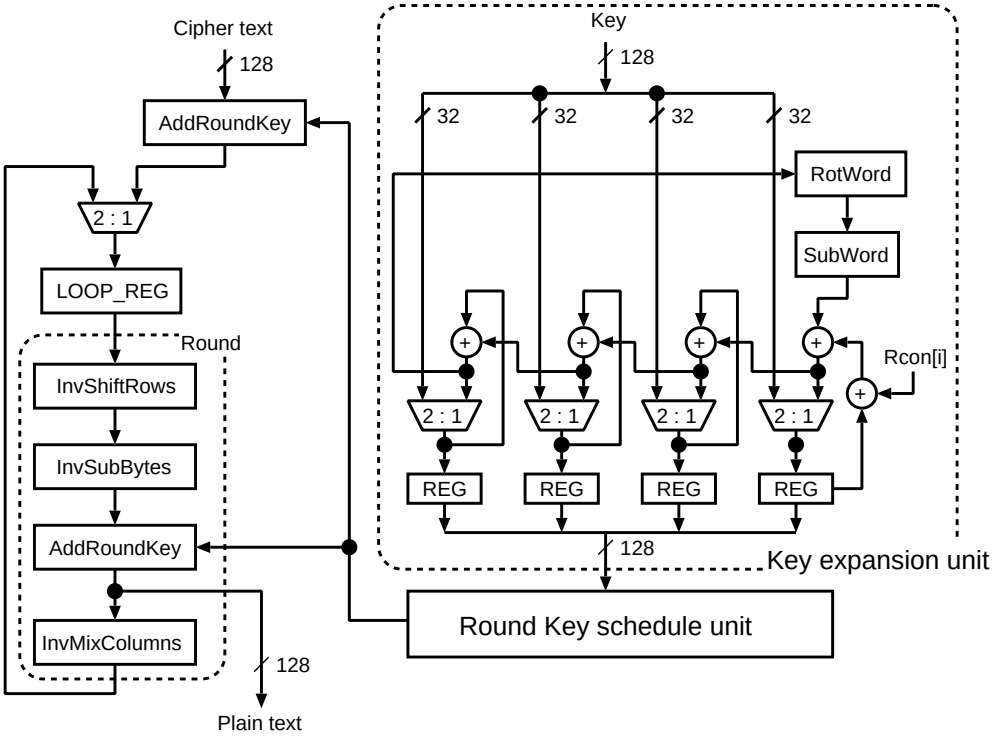

図 4.12: AES 復号回路の構成

表 4.2: Triple-DES 復号回路の評価結果

回路	復号処理時間 [cycle]	回路面積 [mm^2]	遅延時間 [ns]
DES_dec1R	48	0.0465	3.91
DES_dec2R	24	0.0898	5.00
DES_dec3R	16	0.274	5.00
DES_dec4R	12	0.465	6.24

次に，Triple-DES の鍵拡大処理回路と鍵スケジュール回路の論理合成の結果を表 4.3 に示す．DES の鍵拡大処理は，選択置換処理とシフト操作から構成されており，選択置換処理は処理するビット線つなぎ換えで実現できるため，鍵拡大処理回路の面積は，復号回路や鍵スケジュール回路と比べると十分に小さい．鍵スケジュール回路において回路面積の約 53%はラウンド鍵のためのレジスタファイルが占め，約 24%は復号処理時に必要なラウンド鍵を選択する回路が占めている．

最後に AES-128 復号において，図 4.12 をもとに設計した回路の論理合成結果を表 4.4 に示す．Triple-DES 復号回路と同じく，論理合成時の遅延制約は 5.0ns とした．

表 4.4 より，設計した AES-128 復号回路は遅延制約を満たしている．4.1.2 項で説明したよう AES-128 の鍵拡大処理は，DES の鍵拡大処理に比べ複雑であるため，

表 4.3: Triple-DES の鍵拡大処理回路と鍵スケジュール回路の評価結果

回路	回路面積 [mm^2]	遅延時間 [ns]
鍵拡大処理	0.0176	2.26
鍵スケジュール	0.615	4.35

表 4.4: AES-128 の復号回路，鍵拡大処理回路，鍵スケジュール回路の評価結果

回路	復号処理時間 [cycle]	回路面積 [mm^2]	遅延時間 [ns]
復号処理	10	0.458	5.00
鍵拡大処理	–	0.135	4.99
鍵スケジュール	–	0.349	2.40

DES 鍵拡大処理回路よりも面積は大きい．しかし，復号処理に必要なラウンド鍵は Triple-DES が，48×16×3=2,304 ビットであるのに対し，AES-128 は 128×11=1,408 ビットであるため鍵スケジュール回路の面積は Triple-DES よりも大幅に小さくなっている．

4.1.3.2 暗号方式の選定

共通鍵暗号はプログラムの暗号化に用いられ，キャッシュミスが発生する毎に復号処理が必要となるため，復号の処理時間は性能に大きく影響を及ぼす．Triple-DES 復号回路において最も復号処理時間が短いのは，遅延制約を満たしていない DES_dec4R を除くと DES_dec3R であるため，以降 DES_dec3R を Triple-DES の復号回路として評価する．鍵拡大処理回路と鍵スケジュール回路を含む Triple-DES 復号回路の面積は，表 4.2 と表 4.3 より，0.907mm^2 となる．一方，鍵拡大処理回路と鍵スケジュール回路を含む AES-128 復号回路の面積は，表 4.4 より，0.942mm^2 となる．DES_dec3R の場合の Triple-DES と AES-128 の復号回路の諸元を表 4.5 に示す．

表 4.5 より，回路面積は AES-128 復号回路の方が Triple-DES 復号回路に比べてわずかながら大きいが，復号処理時間に関しては AES-128 復号回路の方が 6 サイクル短い．また，一度に復号処理できるデータであるブロックサイズに関しても，Triple-DES の 64 ビットに対し，AES-128 は 2 倍の 128 ビットとなっている．ここで，キャッシュラインサイズを 32 バイトと仮定した場合，暗号化されたラインサイズのデータ全ての復号にかかる処理時間を考えると，Triple-DES は，64 ビット毎に処理するので計 4 回の復号を行い，16×4=64 サイクルの処理時間となる．一方，AES-128 は，128 ビット毎に処理するので計 2 回の復号を行い，10×2=20 サイクルの処理時間となる．通常キャッシュミス時には，キャッシュラインサイズのデータが一度にプロセッサに読み込まれるわけではなく，複数回に分けてメモリバスの転送能力に応じたサイズのデータが読み込まれるため，ブロックサイズが

表 4.5: Triple-DES と AES-128 復号回路の諸元

	回路面積 [mm^2]	復号処理時間 [cycle]	ブロックサイズ [bit]	鍵の長さ [bit]
Triple-DES	0.907	16	64	168 (196)
AES-128	0.942	10	128	128

小さい Triple-DES の方が先に復号処理を開始できるという利点はあるが，その点を踏まえても Triple-DES と AES-128 復号回路のスループットの差は顕著であるといえる．

次に，Triple-DES と AES-128 の安全性について評価する．Triple-DES, AES-128 ともに全数探索法に対しては，現時点での計算機の計算能力に対して十分な鍵の長さをもっていると考えられている．Triple-DES において，選択平文攻撃[5]の一種である中間一致攻撃 (Meet in the middle Attack) と呼ばれる解読攻撃が存在するが，解読攻撃に必要となる選択平文数が 2^{56} 個必要であり，また，平文を用意できたとしても 2^{108} 程度の計算量を有するため現時点では有効な解読法とはならないと考えられる．AES についても，現在有効な解読法は発見されておらず，Triple-DES と AES-128 ともに安全性に関しては十分な強度を持っていると考えられる．

以上から，Triple-DES，AES-128 とも全数探索法に対して十分な安全強度を持っているが，同程度の面積の復号回路において，AES-128 復号回路の処理性能が Triple-DES 復号回路よりも高いため，本論文では，実行プログラムを暗号化する共通鍵暗号として AES-128 を採用する．今回，AES-128 復号回路の設計評価に当たっては，暗号モードは ECB (Electronic Code Book) を前提としている．ECB は，一番単純な暗号モードであり，平文を暗号鍵を用いて暗号化し暗号文とするが，この場合，同一の平文は同一の暗号文に暗号化されるため，暗号文一致攻撃[6]に弱いという欠点がある．先行研究で設計された AES 復号回路では命令長を 32 ビットとした場合，4 個の命令が同じ並びになれば，そのブロックは同じ暗号文として出力される．4 命令が同じ並びになるブロックの出現頻度は高くないと考えられるが，暗号文一致攻撃への対策を講ずるのであれば，CBC (Cipher Block Chaining) モード [18] などの暗号モードを利用する必要がある．

CBC モードでは，最初のブロックの平文に対して，初期値 *IV*（Initial Vector）[7]との排他的論理和をとった値を入力として，暗号化を行う．得られた暗号文を次のブロックの平文との排他的論理和をとり，次の入力とする．以下，同じように暗号化を行う．復号は，通常の復号を行った後に，前のブロックの暗号文との排他的論理和をとることで行う．**図 4.13** に CBC モードでの処理の様子を示す．CBC モードでは復号に，前のブロックの暗号文を用いるため，主記憶から 2 個分のブロックサイズのデータを転送してくる必要がある．また，復号処理ごとに排他的

[5]任意の平文に対応する暗号文を得られる条件で行う攻撃
[6]同一の暗号文の出現頻度から，もとの平文を推定する攻撃.
[7]*IV* は毎回異なる任意の値で，平文と同じサイズのものを用いる．

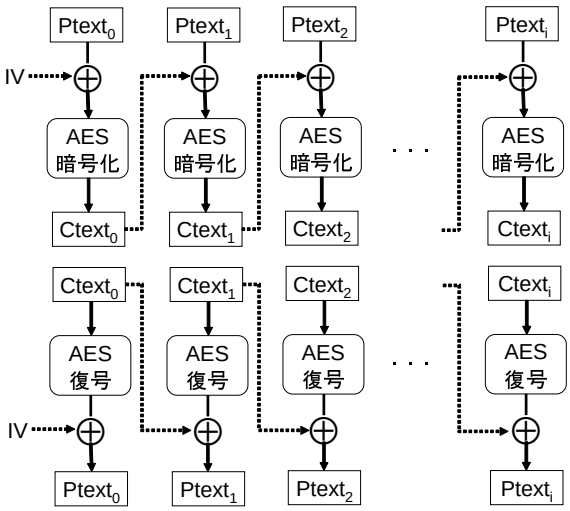

図 4.13: CBC モードでの処理の様子

論理和の処理も追加されるため，その点が復号回路の設計に与える影響について調査し，回路面積や処理性能を評価する必要がある．

4.2 公開鍵暗号

本節では，広く社会に用いられている公開鍵暗号である RSA を取り上げ，暗号方式の調査を行う．

本研究では，プログラム鍵の暗号化/復号に公開鍵暗号を用いる．暗号化はベンダがプログラムをプログラム鍵で暗号化したあとさらにそのプログラム鍵を暗号化する際に行われる．復号はセキュアプロセッサでプログラムが実行され，プロセスが生成される際に，OS の発行する KEYDEC 命令によって行われる．起動中の実行プロセスの数が鍵テーブルのインデックス数を越える場合は，LRU などのアルゴリズムを用いて使われていないプログラム鍵を新しい鍵と入れ換えなければならない．もし入れ換える前のプログラム鍵が必要になったときには再び対応するプログラム鍵の復号を行う必要があるが，鍵テーブルの数が十分であればほとんどのプログラムでは，復号はプログラム実行時の 1 回でよい．また，復号処理時間はプログラムを二次記憶から主記憶にローディングしている間にこの復号処理を並行して行うとすると，処理時間を隠蔽できることが期待できる．このため，公開鍵暗号の復号回路は，回路規模の小面積化に重点を置いた設計にすることで影響を最小限にする．

本節で取り上げる公開鍵暗号のアイデアは 1976 年に Whitfield Diffie と Martin E.Hellman によって考案されたものである．そのアイデアとは，

- 鍵のペア A，B を作成する

- 鍵 A で暗号化した暗号文はペアとなる鍵 B でのみ復号可能である
- 鍵 B で暗号化した暗号文はペアとなる鍵 A でのみ復号可能である
- 暗号化に用いる鍵がわかっても復号に用いる鍵を推測すること，暗号文から平文を推測することは計算量的に不可能である

というものである．それまでの共通鍵暗号を用いた秘密通信では，通信の前に相手と暗号化/復号に使用する鍵を秘密に共有する必要があり，どうやって他者に秘密に鍵を共有するかが問題となっていた．しかし，公開鍵暗号のアイデアを用いればこの問題が解決できる．受信者は作成した鍵のペアのうち片方を公開鍵として他者に広く公開し，もう片方の鍵を秘密鍵として，秘密に保管しておく．情報の送信者は送信の際に情報を受信者の公開鍵で暗号化しておけば，復号できるのは秘密鍵を持つ受信者だけなので，鍵の共有を気にする必要はない．

4.2.1 RSA

1977 年に Ron Rivest，Adi Shamir，Leonard Adleman は公開鍵暗号のアイデアを実現する暗号方式を考案した．この暗号方式は 3 人の名前の頭文字から RSA 暗号 [11, 12] と名付けられた．その後，ElGamal 暗号や楕円曲線暗号など，多くの公開鍵暗号方式が考案されてきたが，RSA 暗号が最も実用化されている暗号方式である．提案システムでは公開鍵暗号としてデファクトスタンダードである RSA を採用した．

まず，RSA について説明する前に，用語の定義を述べ，次に暗号化/復号の手順を説明し，RSA の安全性，演算アルゴリズムについて説明する．

4.2.1.1 用語の定義

- 互いに素
  定義　整数 a, b の最大公約数が 1 であるとき，a と b は互いに素であるという．
- 合同
  定義　自然数 n と整数 a, b に対して，差 $a - b$ が n の倍数であるとき，a と b は n を法として合同であるという．数式で表すと以下のようになる．

$$a \equiv b \ (mod\ n)$$

- オイラー関数
  定義　自然数 n に対して，1〜n の中で n と互いに素となる数の個数を $\varphi(n)$ で表し，この φ をオイラー関数という．

- オイラーの公式
  定理　素数 p に対して

  $$\varphi(p) = p - 1$$

  が成り立つ．また，自然数 a, b に対して，a, b が互いに素であるならば

  $$\varphi(ab) = \varphi(a)\varphi(b)$$

  が成り立つ．したがって，素数 p, q に対して

  $$\varphi(pq) = (p-1)(q-1)$$

  が成り立つ．

- オイラーの定理
  定理　自然数 n と整数 a に対して，n, a が互いに素であるならば

  $$a^{\varphi(n)} \equiv 1 \ (mod\ n) \tag{4.4}$$

  が成り立つ．

4.2.1.2　暗号化/復号のアルゴリズム

RSA の暗号化/復号の手順について説明する．

1. 相異なる素数 p, q を決め，$M = pq$ となる M を求める
2. $\varphi(M) = (p-1)(q-1)$ と互いに素になる e を求める
3. $ed \equiv 1 \ (mod\ \varphi(M))$ となる d を求める
4. M, e を公開鍵，d を秘密鍵とする
5. $Ctext \equiv (Ptext)^e \ (mod\ M)$ で暗号化する
6. $Ptext' \equiv (Ctext)^d \ (mod\ M)$ で復号する

ここで，平文は $Ptext$，暗号文は $Ctext$ である．また，暗号化する平文 $Ptext$ は M より小さい値でなくてはならない（$0 < Ptext < M$）．平文 $Ptext$ が M より大きい場合は，分割して M より小さくなるようにすればよい．

$Ptext'$ と $Ptext$ が同じ値であることは以下のように証明される．

証明　暗号化の式 $Ctext \equiv (Ptext)^e \ (mod\ M)$ の両辺を d 乗[8]すると，

$$(Ctext)^d \equiv (Ptext)^{ed} \ (mod\ M) \tag{4.5}$$

[8]等式の場合と同様に，合同式は両辺の加算，減算，乗算，べき乗が可能である．

となる．$ed \equiv 1\ (mod\ \varphi(M))$ であるので，

$$ed = k\varphi(M) + 1 \tag{4.6}$$

となる．ここで k は整数である．
オイラーの定理の式 4.4 を $Ptext$ と M に当てはめ，両辺を k 乗すると，

$$(Ptext)^{k\varphi(M)} \equiv 1\ (mod\ M)$$

となり，さらに両辺を $Ptext$ で乗じると，

$$(Ptext)^{k\varphi(M)+1} \equiv Ptext\ (mod\ M)$$

となる．よって，式 4.6 から

$$(Ptext)^{ed} \equiv Ptext\ (mod\ M) \tag{4.7}$$

が成立する．したがって，式 4.5，4.7 から

$$Ptext' \equiv (Ctext)^d \equiv (Ptext)^{ed} \equiv Ptext\ (mod\ M)$$

となる．ここで $0 < Ptext < M$ かつ $0 \leq Ptext' < M$ なので $Ptext' = Ptext$ である．

4.2.1.3 RSA の安全性

公開鍵暗号では暗号化に用いる鍵がわかっても復号に用いる鍵を推測することや暗号文から平文を推測することは計算量的に不可能でなくてはならない．つまり，暗号文 $Ctext$ と公開鍵 M, e がわかっても秘密鍵 d を推測すること，暗号文 $Ctext$ から平文 $Ptext$ を推測することは計算量的に不可能でなくてはならない．RSA では，d を計算せずに $Ctext$ から $Ptext$ を得る方法，また，$M = pq$ であることを知らずに d を得る方法は見つかっていない．そのため，$Ctext$ から $Ptext$ を得るには必ず d を計算しなければならないが，d を得るためには $\varphi(M)$ を計算する必要があり，$\varphi(M)$ は $M = pq$ という素因数分解をしなければ求められない．

現在のところ 2 つの素数を掛け合わせた数を素因数分解する効率的な方法は見つかっておらず，小さな素数から順に割り切れるかどうか確かめていく方法がとられるが，M は 10 進数で 300 桁以上の値（1,000 ビット程度）を用いることが多く，その場合，計算機を使っても現実的な時間内で素因数分解を行うことはできない．つまり，この素因数分解の困難さが RSA の安全性の根拠となっているといえる．

ただし，RSA 暗号を解読することは M を素因数分解することと同程度らしい，としかいえない．これは，現在のところ素因数分解が難しいならば，RSA 暗号の解読も難しいという関係が完全に証明されているわけではないからである．つまり，素因数分解をしないで RSA 暗号を解読する方法が存在するかもしれない．ただ，今のところはそのような方法は発見されていない．

4.2.1.4 RSA での演算について

公開鍵の 1 つである法 M のビット数 n はセキュリティパラメータと呼ばれ，この値が大きいほど素因数分解が困難になるため，RSA の強度と密接な関係がある．現在，安全性の観点から 1,024 ビット以上が利用されることが多い．ここで，もう片方の公開鍵 e と秘密鍵 d は n ビットまでの値をとりうるため，RSA における暗号化/復号の演算，

$$Ctext \equiv (Ptext)^e \ (mod\ M)\text{（暗号化）}$$
$$Ptext \equiv (Ctext)^d \ (mod\ M)\text{（復号）}$$

は非常に高い次数でのべき乗演算となる．また，べき乗剰余演算は $A{\cdot}BmodM$ の乗算剰余演算に分解できるが，乗算を行った後に除算を行うと非常に計算コストがかかる．よって，これらの演算を効率よく処理するアルゴリズムが必要である．以降 RSA における演算アルゴリズムについての説明を行う．なお，提案システムでは復号回路のみを有するため以後の説明では復号を例に説明するが，暗号化についても $Ctext$，d をそれぞれ $Ptext$，e におきかえれば同様に処理できる．

4.2.2 べき乗剰余演算のアルゴリズム

RSA の復号処理では，べき乗剰余演算が必要である．M を法とするべき乗剰余演算は $(Ctext)^d modM$ を計算する演算である．本節では効率よくべき乗剰余演算を行うためのアルゴリズムとしてバイナリ法と k-ary 法について説明する．

4.2.2.1 バイナリ法

$(Ctext)^d$ の計算において，指数 d を $d = 2d_1 + d_1'$（d_1' は 0 または 1) と分解する．すると $(Ctext)^d = ((Ctext)^{d_1})^2 \cdot (Ctext)^{d_1'}$ と表すことができ，そのまま C を d 回乗算するよりも乗算回数は少なくなる．さらに，$(Ctext)^{d_1}$ を $d_1 = 2d_2 + d_2'$ と分解していくことで，乗算回数を削減できる．この性質を利用したものがバイナリ法である．**図 4.14** にアルゴリズムを示す．

このアルゴリズムでは d を 2 進表記した場合のビット数 $i(i = \lfloor log_2 d \rfloor + 1)$ 回の 2 乗剰余と 1 が立っているビット数分の乗算剰余が必要である．M のビット数を 1,024 ビットとした場合，最悪の乗算剰余回数は $1,024 + 1,024 = 2,048$ 回，平均 $1,024 + 1,024/2$ =1,536 回となる．

4.2.2.2 k-ary 法

バイナリ法を改良し，より乗算回数を減らすことのできるアルゴリズムとして k-ary 法 [13] が考案されている．図 4.14 に示したバイナリ法のアルゴリズムでは上位か

Input: C,d,M
Output: $P = C^d \bmod M$

STEP1: $P = 1$;
STEP2: $i = \lfloor log_2 d \rfloor + 1$;
STEP3: $P = P{\cdot}PmodM$; $i = i - 1$;
STEP4: if ($d_i = 1$) $P = P{\cdot}CmodM$;
STEP5: if ($i \neq 0$) return STEP3;
else output P;

図 4.14: バイナリ法

ら 1 ビットずつ d を調べ，現在着目している d のビットが 1 であれば $P = P{\cdot}Pmod\ M$ の後に $P = P{\cdot}Cmod\ M$ を計算するが，k-ary 法では d を k ビットごとにグループ化し，d を 1 ビットずつではなく k ビットずつ読み込んで，乗算回数を削減する手法である．**図 4.15** にアルゴリズムを示す．着目している d のビットに関係なく $P = P{\cdot}Pmod\ M$ は計算する必要があるためこの演算は k ビットごとに k 回行うが，$P = P{\cdot}Cmod\ M$ の演算は P と掛け合わせる値を $C,\ C^2, C^3 ... C^{2^k-1}$ と用意しておけば k ビット分の演算をまとめて処理できる．つまり，$P = P{\cdot}Cmod\ M$ の演算を k ビットごとに 1 回に削減できる．$C,\ C^2, C^3 ... C^{2^k-1}$ の内のどれを乗じるかは着目している d の k ビットごとのビットパターンによって選択する．

k-ary 法を用いることにより，バイナリ法に比べ全体の乗算回数は削減できるが，あらかじめ $C,\ C^2, C^3 ... C^{2^k-1}$ を計算しておかなければならないため，前処理の演算として $2^k - 2$ 回の乗算が余分に必要である．注意しなければならないのは k の値を大きくしすぎると前処理の演算回数が指数関数的に増えるため，かえって演算回数が増加してしまうということである．

4.2.2.3　べき乗演算アルゴリズムの評価

RSA の法 M のビット数 $n = 1,024$ の場合に，バイナリ法と k-ary 法での乗算回数の比較し評価を行う．1,024 ビットのうち 1 の立っている数によって乗算回数は変化するため，最悪の場合として，全てのビットが 1 のときと，平均して半分のビットが 1 のときに分けて考え，k-ary 法における k の値を変化させた場合の乗算回数を計算した．それぞれの結果を**図 4.16,4.17** に示す．ただし，図 4.17 における k-ary の乗算回数は着目しているグループ化した値 d_i のビットパターンによって，乗算回数に変化が生じるため[9] $M = (1, 0, 1, 0 \cdots 1, 0)_2$ とした場合と 1,024 ビットの

[9] $k = 2$ の場合を考えると $d_i = (00)_2$ のときは乗算しなくてよい．

Input: C,d,M
$d = (d_{t-1} \cdots d_0)_{2^k}$
Output: $P = C^d mod M$
. . . . . .
STEP1: $for\ \ i = 1\ \ to\ \ 2^k - 1$
$C_i = C^i mod M$;
STEP2: $P = C_{d_{n-1}}$; $i = t - 1$;
STEP3: $for\ \ j = 1\ \ to\ \ k$
$P = P \cdot P mod M$;
STEP4: $i = i - 1$; $P = P \cdot C_{d_i} mod M$;
STEP5: if ($i \neq 0$) return STEP3;
else output P;

図 4.15: k-ary 法

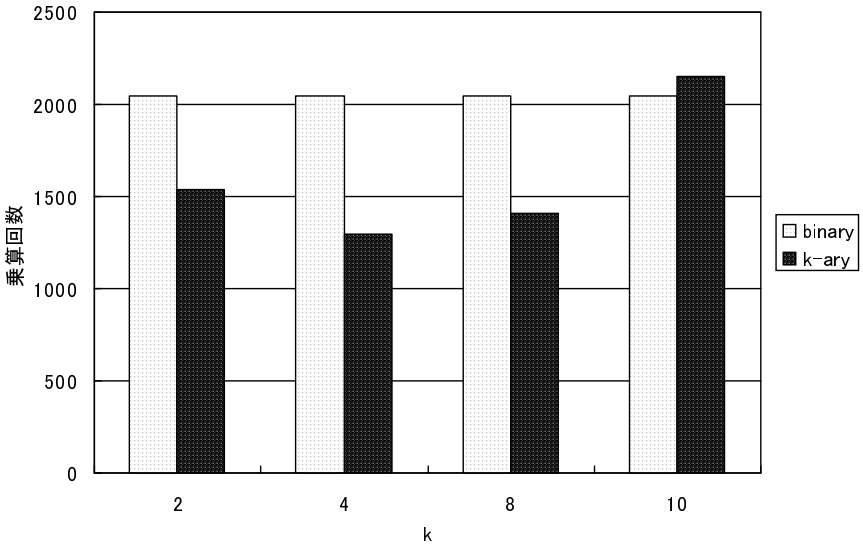

図 4.16: 全てのビットが 1 の場合の乗算回数の比較

内，前半の 512 ビットが連続して 1 であり，後半の 512 ビットが全て 0 である場合をそれぞれ計算しその平均の値を乗算回数としている．

結果より $k = 4$ のときに最も乗算回数が削減できることがわかる．また，$k = 10$ 以上にするとバイナリ法を用いるよりも乗算回数が多くなる．これは前処理の演算回数が指数関数的に増えたためである．以上より，$k = 4$ が最もよいと考えられるが，ハードウェア実装を考えると前処理の計算結果 C_i を保存しておくレジスタがバイナリ法に比べて k-ary 法では $2^k - 2$ 個余分に必要となる．C_i の値は 1,024 ビッ

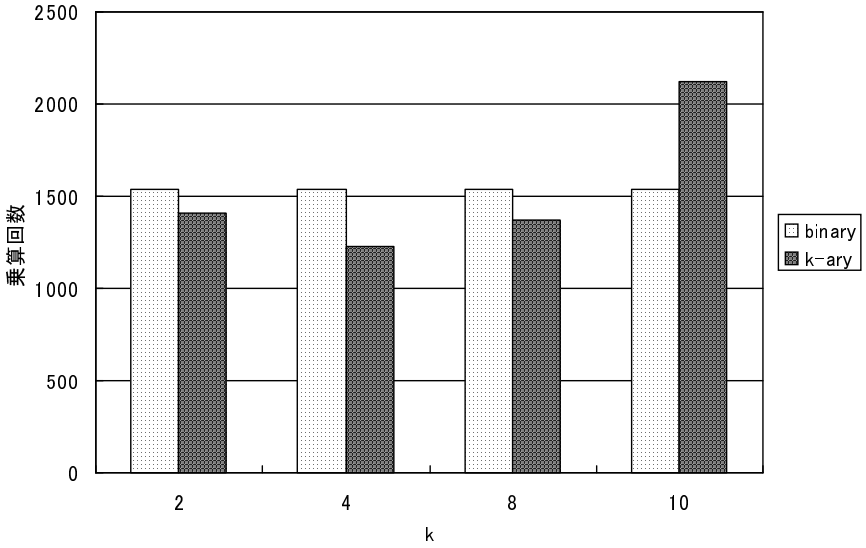

図 4.17: 半分のビットが 1 の場合の乗算回数の比較

トまでの値をとりうるために，$k = 2$ だとしても面積の増加は大きい．RSA の復号回路は回路規模の小面積化に重点を置いた設計にするので，べき乗演算アルゴリズムにはバイナリ法を用いることにする．

4.2.3 乗算剰余演算のアルゴリズム

バイナリ法を用いる場合，べき乗剰余演算は乗算剰余演算に分解される．M を法とする乗算剰余演算は，$A{\cdot}BmodM$ を計算する演算である．効率よく乗算剰余演算を行うためのアルゴリズムとしてモンゴメリ法ついて説明する．

4.2.3.1 モンゴメリ法

$A{\cdot}BmodM$ において M による剰余を求める処理は除算を利用すると非常に処理時間がかかる．そこで，剰余を除算を用いることなく処理する計算手法としてモンゴメリ法が考案されている [14, 15].

モンゴメリ法は

$$AB + MT = WR \tag{4.8}$$

という方程式において W を求める方法である．ここで，$R = 2^n$ であり，A，B は与えられた非負の整数，M は n ビットの法で，$R/2 < M < R$ という範囲にあるものとする．T，W は未知数である．このとき，M は奇数であり，R，M は互いに素となるから，式 4.8 は無限個の解をもつ．式 4.8 の両辺を M で mod をとると，

$$W \equiv ABR^{-1}(modM)$$

となる．したがって，式 4.8 を満たすような n ビットの T を構成すれば，W が得られることになる．T は式 4.8 の両辺を R で mod をとると，

$$N \equiv -ABM^{-1}(mod 2^n)$$

となる．このことから，T を求めるのに必要な剰余計算は，$mod\ 2^n$ の形となる．$mod\ 2^n$ の形の剰余計算は単に下位 n ビットを取り出すだけでよいので簡単である．ただし，上記のモンゴメリ法で求まる剰余の値 W は $ABmodM$ そのものではなく，$ABR^{-1}modM$ である．以下，$W = Mont(A, B, M) = ABR^{-1}modM$ とする．ここで，モンゴメリ形式として $A^* = ARmodM$ とすると

$$\begin{aligned} & Mont(A^*, B^*, M) \\ = \ & (AR)(BR)R^{-1}modM \\ = \ & ABRmodM \\ = \ & (AB)^* \end{aligned}$$

となる．つまり，モンゴメリ形式のデータ同士をモンゴメリ法で演算したとき，結果は，積のモンゴメリ形式になる．したがって，べき乗剰余演算を実行する際は，データをモンゴメリ形式に変換しておけば，その後の計算も，データ形式を修正することなく行える．ただし，最後にモンゴメリ形式 Z^* から Z に変換する必要がある．この変換は $Mont(Z^*, 1, M)$ で行える．また，A から A^* への変換は $Mont(A, R^2 modM, M)$ で行える．これらの変換は計算全体の最初と最後に行うだけでよい．

モンゴメリ法の解 W は乗数 B の下位ビットから 1 ビットずつ逐次計算することで求めることができる．AB の最下位ビットと MT の最下位ビットの和が 0 になるように T の着目しているビット t_j を決定していけばよい．この手順を乗数 B の 1 ビットずつ繰り返すことで**図 4.18** に示すように W を求めることができる．

アルゴリズムを**図 4.19** に示す．このアルゴリズムは，累算結果 Q_j に部分積 $A{\cdot}b_j$ を加え，その和 R_j が偶数ならこれを 2 で割り，奇数ならこれに法 M を加えてから 2 で割るというステップを n 回繰り返すものである．Q_j と $A{\cdot}b_j$ との和が偶数ならば最下位ビットが 0 なので，Mt_j との最下位ビットの和を 0 にするには $t_j = 0$ でなければならない．$Mt_j = 0$ となり，最下位ビットの 0 を削除するために 2 で割る（1 ビット右シフトする）．Q_j と $A{\cdot}b_j$ との和が奇数の場合，Mt_j との最下位ビットの和を 0 にするには $t_j = 1$ でなければならない．$Mt_j = M$ となるので，M を加えてから 2 で割る．ここで，$B = (b_{n-1}, b_{n-2}...b_0)_2$，$T = (T_{n-1}, T_{n-2}...T_0)_2$ としている．

4.2.3.2　2 次ブースのアルゴリズム

上記のモンゴメリ法は乗数 B の 1 ビットずつ部分積 $A{\cdot}b_j$ を生成した．つまり，部分積の数は乗数 B のビット数 n に等しい．ここで，部分積の数を少なくできれば，

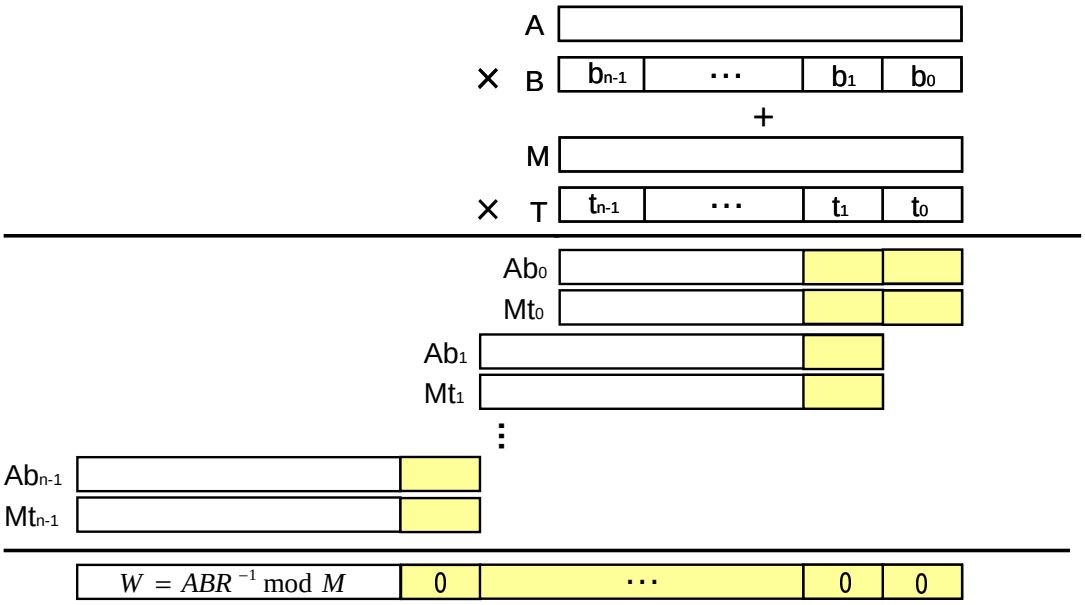

図 4.18: モンゴメリ法の演算

Input: A,B,M
Output: $W = Mont(A,B,M)$
$= ABR^{-1} mod M$

STEP1: $Q_0 = 0$;
STEP2: $for \;\; j = 0 \;\; to \;\; n-1\{$
$R_j = Q_j + A \cdot b_j$;
if $(2|R_j) \quad Q_{j+1} = R_j/2$;
else $Q_{j+1} = (R_j + M)/2;\}$
STEP3: if $(Q_{n-1} \geq M) \quad W = Q_{n-1} - M$;
else $W = Q_{n-1}$;

図 4.19: モンゴメリ法

計算の高速化につながる．2 次ブースのアルゴリズム [16] を用いると, 生成される部分積の個数を半分にすることができる．

被乗数 A, 乗数 B は，ともに 2 の補数表現で与えられるものとする．ここで a_{n-1}, b_{n-1} は符号ビット，a_i，b_i は数値ビットで，$a_{-1} = b_{-1} = 0$ とする．

表 4.6: 2 次ブースのデコード規則

b_{2j+1}	b_{2j}	b_{2j-1}	$\hat{b_j}$	出力 P_j
0	0	0	0	0
0	0	1	1	1A
0	1	0	1	1A
0	1	1	2	2A
1	0	0	-2	-2A
1	0	1	-1	-1A
1	1	0	-1	-1A
1	1	1	0	0

乗数 B は次のように表すことができる.

$$
\begin{aligned}
B &= -1\cdot b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \\
&= -1\cdot b_{n-1}2^{n-1} + 2\sum_{i=0}^{n-2} b_i 2^i \ - \sum_{i=0}^{n-2} b_i 2^i \\
&= -\sum_{i=0}^{n-1} b_i 2^i \ + 2\sum_{i=0}^{n-2} b_i 2^i \\
&= -\sum_{i=0}^{n-1} b_i 2^i \ + \sum_{i=0}^{n-1} b_{i-1} 2^i \\
&= \sum_{i=0}^{n-1} (-b_i + b_{i-1}) 2^i \\
&= \sum_{i=0}^{(n-2)/2} (2(-b_{2i+1} + b_{2i}) + (-b_{2i} + b_{2i-1}))2^{2i} \\
&= \sum_{i=0}^{(n-2)/2} (-2b_{2i+1} + b_{2i} + b_{2i-1})2^{2i} \qquad (4.9)
\end{aligned}
$$

式 (4.9) において, $(-2b_{2i+1} + b_{2i} + b_{2i-1})$ の値は b の値が 0 または 1 であるので $-2, -1, 0, 1, 2$ のいずれかとなる．この値をデコード値という．部分積は乗数 B の最下位に $b_{-1} = 0$ を追加し，2 ビットずつシフトしながら，最下位の 3 ビットを読み取りデコードすることで求める．$(-2b_{2i+1} + b_{2i} + b_{2i-1})$ の値が 2 のときは単純に被乗数 A の値を 1 ビット左シフトすればよい．-2 のときは A の値を 1 ビット左シフトし，2 の補数をとればよい．また，式 (4.9) より i の値は 0 から $(n-2)/2$ まで $n/2$ 個の値をとることがわかる. よって，部分積は $n/2$ 個生成されることになり，部分積の個数が半分に削減できる. デコード規則を**表 4.6** に示す.

Input: A,B,M
Output: $W = Mont(A, B, M)$
$= ABR^{-1} mod M$

STEP1: $Q_0 = 0$
STEP2: $for \;\; j = 0 \;\; to \;\; n/2 - 1\{$
$(t_{j1}, t_{j0}) = (Q_j + P_j)(mod 4)$;
if $(t_{j0} = 0)\{$
if $(t_{j1} = 0)$
$Q_{j+1} = (Q_j + P_j)/4$;
else
$Q_{j+1} = (Q_j + P_j + 2M)/4;\}$
else {
if $(t_{j1} = m_1)$
$Q_{j+1} = (Q_j + P_j - M)/4$;
else
$Q_{j+1} = (Q_j + P_j + M)/4;\}\}$
STEP3: if $(Q_{n/2-1} < 0)$ $W = Q_{n/2-1} + M$;
else $W = Q_{n/2-1}$;

図 4.20: 基数 4 のモンゴメリ法

4.2.3.3 基数 4 のモンゴメリ法

2 次ブースのアルゴリズムを利用する場合，モンゴメリ法における法を加算するかどうかの判定部分は乗数の 2 ビットごとに部分積を生成しているため，従来のように和 R_j が奇数か偶数かを調べるだけでは不十分である．そのため，基数 4，つまり，乗数の 2 ビットずつ計算を行う場合のモンゴメリのアルゴリズムが提案されている [17]．このアルゴリズムを利用すればイタレーション回数は半分になり，サイクル数の削減ができる．

部分積 P_j は先ほど述べた 2 次ブースのデコード規則を用いて生成する．これはシフトと正負反転で行える．デコード結果が負の場合 2 の補数をとるが，正負反転後の 1 を加算する処理は後の加算器で行うことができる．基数 4 のモンゴメリ法では，法 $M = (m_{m-1}, ...m_1, m_0)_2$ の m_1 と累算結果 $Q_j + P_j$ の LSB の 2 ビット (t_{j1}, t_{j0}) により加算する法の値 $(2M, M, -M,$ または $0)$ を決定する．**図 4.20** にアルゴリズムを示す．なお，2 ビットずつの計算なので法を加算した後は 4 で割る必要がある．

第5章　設計・評価

本章ではフィージビリティ・スタディとして，本提案システムを実際にプロセッサチップに実装した場合に，どの程度のチップ面積増加が見込まれるかを調査する．また，プロセッサが本論文で提案するプログラム保護機能を持った場合に，プログラムの実行時間与える影響についても調査する．

5.1　比較対象となるプロセッサ

プロセッサの回路面積と比較評価するに当たって，評価に用いるプロセッサは公平な比較のため，AES-128 復号回路と同じように設計されたプロセッサであることが望ましい．つまり，様々な小面積化技術を用いてカスタム設計で実現されるような市販のプロセッサの回路面積と比較を行うのではなく，AES-128 復号回路と同様に，プロセッサの RTL 記述をスタンダードセルライブラリを用いて論理合成して得られる回路面積と比較を行うことで評価のレベルを合わせる．このため，本論文では比較の対象となるプロセッサとして，本研究室で設計されたシンセサイザブルなプロセッサ (Luehdorfia)[33] を用意した．Luehdorfia は，ARM アーキテクチャを基に設計され，プロセッサとして必要最低限の機能を有した非常にシンプルな構成のプロセッサコアとなっている．Luehdorfia の構成を表 5.1 に示す．

次に，Luehdorfia を AES-128 復号回路と同様のスタンダードセルライブラリを用いて論理合成を行った結果を表 5.2 に示す．ただし，通常レジスタファイルには当該プロセスに最適化された 6T-SRAM(6Tr./bit) が使用されるが，本論文で評価に用いているセルライブラリには 6T-SRAM が無いので，命令キャッシュ，データキャッシュそして CPU コア内のレジスタファイルの面積については，6T-SRAM での実現を仮定した値とする．6T-SRAM の回路面積は，同程度のトランジスタサイズを持つインバータの回路面積を参考にする．詳しくは，6T-SRAM の回路（図 5.1）から，NMOS トランジスタのサイズを 1，PMOS トランジスタのサイズを 2 とすると 6T-SRAM のトランジスタサイズは 8 となり，基本の 3 倍の強さを持ったインバータ (INV_3) と同程度のトランジスタサイズとなるため，本論文で論理合成に用いているセルライブラリのうち INV_3 に相当する回路の面積を参考にし，6T-SRAM の回路面積は $48\mu m^2$ とした．6T-SRAM による回路の実現では，別途アドレスデコーダやセンスアンプが必要となるので，仮定した 6T-SRAM の回路面積は，少し大きめに設定している．

表 5.1: Luehdorfia の構成

CPU コア	ARM version 5 互換命令セット 6 段パイプライン バレルシフタ 1 個 ALU1 個 32×32 乗算器 1 個 汎用レジスタ（32 ビット ×16 本） 特権レジスタ（32 ビット ×20 本） 動作周波数 200 MHz
命令キャッシュ	4KB ダイレクトマップ方式 ラインサイズ：16B
データキャッシュ	4KB ダイレクトマップ方式 ラインサイズ：16B

表 5.2: Luehdorfia の回路面積

	面積 [mm^2]
CPU コア	1.12
命令キャッシュ (4KB)	2.16
データキャッシュ (4KB)	1.87
合計	5.15

5.2 AES-128 復号回路の設計・評価

5.2.1 AES-128 復号回路との面積比較

図 4.4 で示した AES-128 復号回路のうち，鍵スケジュール回路では，ラウンド鍵を格納するレジスタファイルが回路面積の大部分を占めている．鍵スケジュール回路のレジスタファイルを 5.1.1 項で述べた 6T-SRAM を用いて実現したとすると，鍵スケジュール回路の面積は 0.072mm^2 となるので，AES-128 復号回路全体の回路面積は 0.665mm^2 となる．Luehdorfia に，AES-128 復号回路を搭載したとすると，表 5.2 より，回路面積は約 13%増加し，5.82mm^2 となる．また，AES-128 復号回路は Luehdorfia のデータキャッシュの約 35%であり，およそ 1.4KB のデータキャッシュの性能と引き換えに設計した復号回路を実装することができる．現時点ではプロセッサに対する AES-128 復号回路の面積はやや大きいと考えられるが，5.1.3 項で述べる回路面積の削減を行うことにより，十分現実的な範囲内になると考えられる．

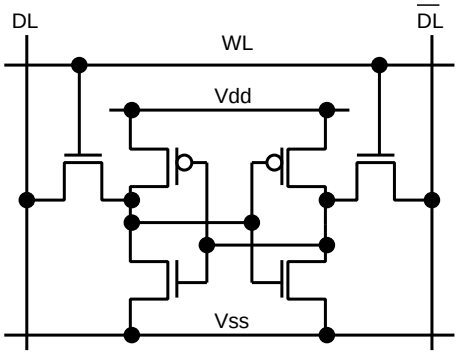

図 5.1: 6T-SRAM の回路

表 5.3: AddRoundKey, InvSubBytes, InvMixColumns の回路面積

	面積 [mm^2]	遅延時間 [ns]
AddRoundKey	0.0108	0.49
InvSubBytes	0.320	3.24
InvMixColumns	0.0824	2.45

5.2.2 復号回路の小面積化

5.2.2.1 BDD による InvSubBytes の実現

AES-128 復号回路の小面積化について考えた場合に，AES-128 復号処理を行う各モジュールの回路面積を把握するため AddRoundKey, InvSubBytes, InvMixColumns の回路を個別に論理合成し，回路面積を求めた．結果を表 5.3 に示す．InvShiftRows は処理するビット線のつなぎ換えで実現されるため，表には含めていない．

表 5.3 から，InvSubBytes が他の 2 つの回路に比べて非常に大きく，AES-128 復号回路中でも回路面積の大部分を占めていると考えられるため，InvSubBytes 回路が小面積で実現することができれば AES-128 復号回路の面積削減に大きく効果があると考えられる．4.1.2 項で述べたように InvSubBytes は，S-Box による変換表で実現されている．InvSubBytes の S-Box は 8 ビット入出力の変換表であるため，16 個の S-box から構成される回路で 128 ビットのデータを処理している．本論文では，S-Box を Verilog-HDL の case 文の記述し，論理合成ツールによるゲートレベルで最適化された回路で実現しているが，森岡らの提案している S-Box を BDD[1] を用いて実現する手法では，消費電力は増えるが，自動合成で得られる回路と同程度の遅延時間で，回路面積を削減できることが示されている [10]．鍵拡大処理内の SubWord でも S-Box を使用しているため，BDD による S-Box の実現手法は，AES-128 復号回路の面積削減への効果が高いと考えられる．

[1] Binary Decision Diagram: 二分決定グラフ．論理関数の表現手法の一つ．

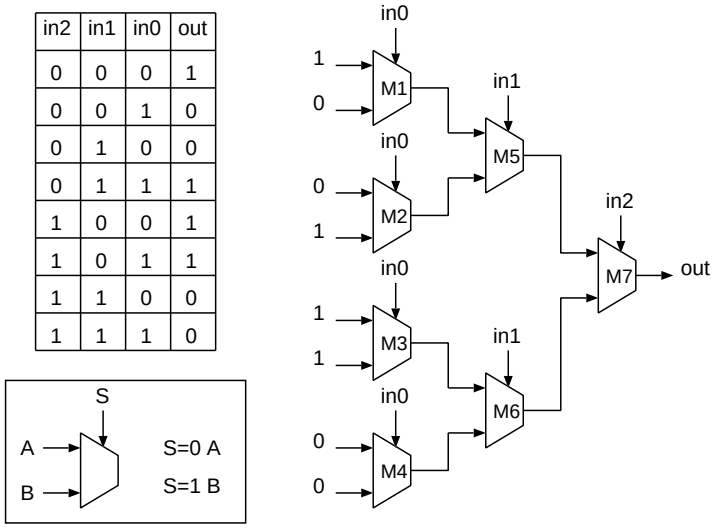

図 5.2: BDD による真理値表の実現

そこで，S-Box を実現する BDD を作成するため，任意の真理値表から BDD を生成するプログラム作成した．C 言語によって記述し，600 行程度の規模となっている．BDD による真理値表の実現の例を図 5.2 に示す．図 5.2 の真理値表において，{*in*2, *in*1, *in*0}={0, 0, 0} であった場合，図 5.2 の BDD では，全ての 2to1 Multiplexer (MUX) で入力 A が通過するので，M1 の入力 A である 1 が，M5, M7 を経て out まで伝搬する．別の論理も同様にして，最下段の MUX の入力が {*in*2, *in*1, *in*0} の値によって選択され，out まで伝搬することで，真理値表が実現される．ここで，M3 は入力が A, B ともに 1 であるので，M3 を消去し論理値 1 の信号線に置き換えることができる．同様にして，M4 は論理値 0 の信号線に置き換えることができる．次に，M2 では入力 (A, B) が (0, 1) であるため，*in*0=0 では 0，*in*0=1 では 1 が出力されるので，M2 は信号線 *in*0 と置き換えることができる．そして，M1 は M2 と入力 A, B が逆になっているので，信号線 $\overline{in0}$ と置き換えることができる．また，M3 と M4 は 0 と 1 の信号線に置き換えられているので，M6 は *in*1 と置き換えることができるため，図 5.2 の真理値表は BDD によって，2 個の MUX と 1 個のインバータで実現することができる．図 5.2 では，出力から順に {*in*2, *in*1, *in*0} の順序で BDD を構成しており，この順序を変更することで同じ真理値表でも別の BDD を生成することができる．変数の順序を変更することで得られる各 BDD は，それぞれ必要とする MUX の個数も変わってくるため，真理値表から BDD を生成する際には，変数の順序付けを十分考慮する必要がある．

図 5.3 に，InvSubBytes の Sbox を示す．図 5.3 は 8 ビット入力の真理値表と考えられるため 8!=40,320 通りの変数の順序が存在し，それぞれの順序から生成される BDD において必要となる MUX の個数を作成したプログラムを用いて求めた．プログラムの結果によると，必要となる MUX の個数が最小の順序付けは {*in*2, *in*5,

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

図 5.3: InvSubBytes の Sbox 表現（入力：XY）

表 5.4: BDD による S-Box と自動合成による S-Box

	面積 [mm^2]	遅延時間 [ns]
論理合成	0.0196	3.09
BDD	0.0261	2.97

*in*0, *in*6, *in*1, *in*3, *in*4, *in*7} となり，386 個の MUX で BDD を実現できる（最大は 422 個）．得られた BDD を用いて設計した S-Box と Verilog-HDL の case 文で設計し論理合成することによって最適化された S-Box の面積と遅延時間を表 5.4 に示す．

表 5.4 より，設計した BDD を用いて実現した S-Box は，論理合成で最適化された S-Box 対して遅延時間はわずかながら下回っているが，回路面積は逆に大幅に増加している．BDD を用いた S-Box の面積は，回路内で大量に用いられている MUX セルに小面積なものを採用することで削減できると考えられる．

5.2.2.2 パストランジスタ論理

AES-128 復号回路において XOR は，復号回路だけでなく鍵拡大処理回路の中まで，様々な処理で使用されている．ここで，小面積な XOR ゲートを用意できれば，AES-128 復号回路内のあらゆる場所に適用でき，回路面積の削減に寄与できると考えられる．小面積な XOR ゲートの構成法として，パストランジスタ論理による構成法が挙げられる．CMOS 論理で構成された XOR を図 5.4 にパストランジスタ論理で構成された XOR を図 5.5 に示す．

図 5.4 から，CMOS 論理で構成した XOR では，8 トランジスタ必要となるが，

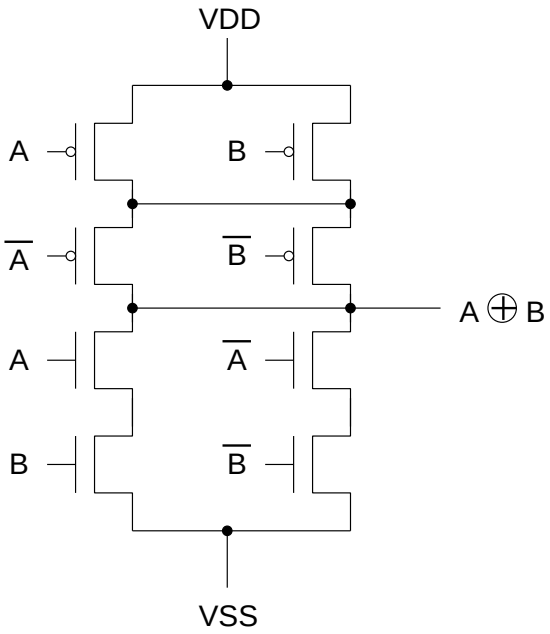

図 5.4: CMOS 論理で構成した XOR

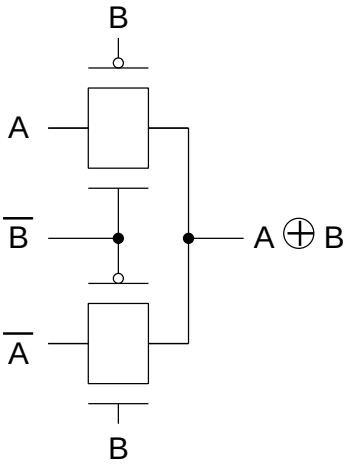

図 5.5: パストランジスタ論理で構成した XOR

パストランジスタ論理で構成した XOR では，図 5.5 より 4 トランジスタで実現することができる．また，図 5.5 において，同じ回路構成のまま入力を変更することで MUX セルを実現することができる．本論文で論理合成に用いているセルライブラリには，パストランジスタ論理で構成されたセルは用意されていなかったが，今後，復号回路の実現にこれらのセルを使用できれば，回路面積の削減が期待できる．

5.2.3 関連研究との比較

Suh らは，文献 [5] で暗号化/復号回路などの評価を行っている．本節では，これらとの比較を行う．AEGIS においても共通鍵暗号方式には AES-128 が用いられているが，AEGIS では，プログラムの暗号化だけでなく，プログラム実行中に用いるデータの暗号化も行っているため，暗号化/復号の両方を行う回路（以下，AES unit）として実装されている．文献 [5] において，AES unit の回路面積は，本論文と同様に，Verilog-HDL による設計記述を Synopsys 社の Design Compiler による論理合成することによって求められている．論理合成に用いられた設計プロセスは，TSMC 0.18μm メタル 6 層プロセスである．

表 5.5 に，文献 [5] で示された AES unit の回路面積と，表 4.4 で示された本提案の AES-128 復号回路の回路面積を示す．表 5.5 において，AES unit は本提案では復号処理回路と鍵拡大処理回路の合計にあたり，Key SPRs (Special Purpose Registers) はラウンド鍵を保存するためのレジスタファイルであり，本論文では鍵スケジュー

表 5.5: 回路面積の比較（文献 [5] の Table 5 を参考）

AEGIS			本提案		
回路	面積 $[mm^2]$	ゲート数	回路	面積 $[mm^2]$	ゲート数
AES unit (3 AES units)	0.238 (0.713)	23,807 (71,426)	復号処理 ＋鍵拡大処理	0.593	19,253
Key SPRs	0.108	10,843	鍵スケジュール	0.349	11,331

ル回路にあたる．ただし，文献 [5] において AEGIS では 3 個の AES unit を実装しており，合計の面積として示されていたので，表 5.5 では，1 個分の回路面積を示している．また，設計環境による回路面積の差を無くした評価を行うため，論理合成で用いるライブラリの NAND2 セルの面積を参考にし，回路面積を NAND2 ゲート換算で回路規模を示している．論理合成で用いるライブラリの NAND2 セルの面積は，AEGIS が $9.97\mu m^2$ なのに対し，本提案では $30.8\mu m^2$ となっている．

表 5.5 に示されたゲート数での比較において，本論文で設計した鍵スケジュール回路と，AEGIS の Key SPRs の差は約 5%であり，2 つの回路は同程度の面積であるといえる．一方，本論文で設計した復号処理回路＋鍵拡大処理回路の面積は，AES unit の回路面積の約 81%である．AES unit は，暗号化機能を持っているので，復号回路だけの本論文の回路との差が小さいように見えるが，復号回路と多くの部分を共有化するように暗号化回路を設計すれば，本論文で設計した回路に暗号化機能を追加した場合も，ほぼ同程度の面積になると考えられる．評価環境が異なるため完全な比較評価ではないが，ほぼ同等の評価といえる．

5.3 RSA 復号回路の設計・評価

先に述べた，バイナリ法，基数 4 のモンゴメリ法に基づき，RSA 復号回路を設計する．べき乗剰余演算をバイナリ法を用いて，乗算剰余演算に分解し，乗算剰余演算をモンゴメリ法を用いて計算する．**図 5.6** に設計する RSA 復号回路のブロック図を示す．

5.3.1 べき乗剰余回路

図 5.6 に示したバイナリ法を用いたべき乗剰余回路部ではまず，初期値として REG_P に 1，REG_C に暗号文 C，REG_D に秘密鍵 d を入力しておく．1 サイクルごとに REG_D の上位ビットから 1 ビットずつ値を読み込み，その値が 0 であれば，MUX1 で REG_P の値が選択され，REG_P 値が乗算剰余回路に 2 つの入力として送られる．そしてその結果は REG_P に格納される．この操作はバイナリ法における $P = P \cdot Pmod\ M$ に相当する．REG_D から読み込んだ値が 1 であれば MUX1 で

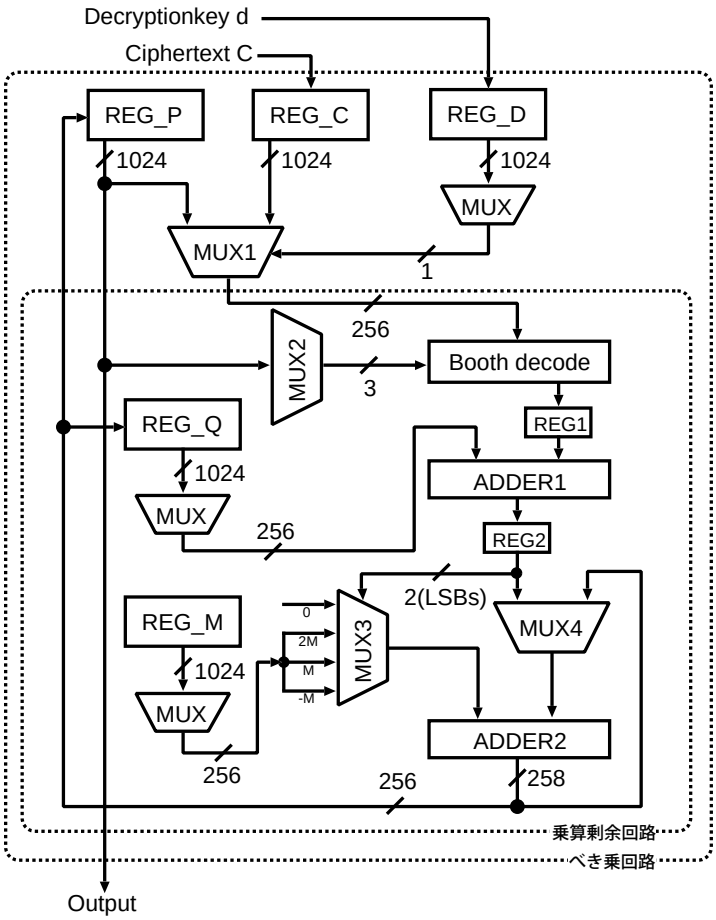

図 5.6: RSA 復号回路の構成

REG_P の値を選択し，$P = P{\cdot}Pmod\ M$ を行った後に，次は REG_D の値を読み込まずに，MUX1 で REG_C の値を選択し，$P = P{\cdot}Cmod\ M$ を行う．

注意すべき点はモンゴメリ法を用いる場合，最初にデータ形式をモンゴメリ形式へ変換しておかなければならないことである．REG_P の初期値 1 と REG_C の暗号文 C が変換対象となる．暗号文 C を変換するには $Mont(C, R^2 modM, M)$ を計算すればよい．$R^2 modM$ はあらかじめ計算しておく．また，P の初期値 1 は固定値なので変換した値 $RmodM$ もあらかじめ計算しておく．変換した結果はそれぞれ REG_P，REG_C に初期値として格納する．$Mont(C, R^2 modM, M)$ の計算は乗算剰余回路を用いて前処理として計算する．なお，実際はレジスタの初期化や MUX1 による入力の選択は，制御回路としてのステートマシンを利用して行っている．

5.3.2 乗算剰余回路

乗算剰余回路部では，入力が決定すると，表4.6のデコード規則にしたがって部分積生成回路Booth decoderで部分積を生成し，REG1に保持する．REG_Dの値を読み込んでからここまでの処理を1サイクルで行う．部分積の生成はシフトとビット反転という簡単な処理で実現できる．2の補数をとる場合，ビット反転の後に1を加算しなければならないが，ADDER1における下位からの桁上げビットを1にすることで対応できるため，新たな加算器などは必要ない．ここで，REG1の値は図4.20における部分積P_jであり，REG_Qの値は累積結果Q_jである．法MはREG_Mに保存されている．

次サイクルで，1,024ビットの加算（$Q_j + P_j$）をADDER1で行う．ここで1,024ビットの加算器を構成するのは面積的にも速度的にも現実的ではないため，より小規模の加算器を複数回使い1,024ビットの加算を実現することにした．どの程度のビット幅の演算をすればよいのかは面積と速度とのトレードオフを考慮して選択する必要があるが，本システムでは256ビットの加算器を4回使う構成とした．つまり，1,024ビットの加算を4サイクルかけて計算する．また，部分積を保存するREG1は256ビット演算の場合，256ビットでよいので，この部分で面積の削減ができる．なお，この構成にした場合，ステートマシンを利用してBooth decorderに1サイクルで256ビットずつ部分積が生成されるようにしなければならない．

ADDER1で1回目の256ビットの加算が終わると，その結果はREG2に保存される．次のサイクルでREG2のLSB2ビットとMの値から加算する法の倍数を決定し，ADDER2で加算する．このとき，ADDER1では2回目の256ビット加算を行うことができる．この方式で計算すると，最初にREG_Qに1,024ビットの結果が保存されるのに7サイクルかかる．次から結果が保存されるのは，ADDER2で4回目の加算をしているときに，ADDER1では次の新しい部分積の1回目の計算を行うことができるため，4サイクルで行うことができる．加算のようすを**図5.7**に示す．

ADDER2での結果を4で割ったものがQ_{j+1}である．割る値が定数なので除算器等は必要なく，単純に右に2ビットシフトするだけでよい．Booth decorderにより乗数の2ビットずつ計算しているため，1,024ビット加算は512回処理される．このため，1回の乗算剰余計算は2,051サイクル（511×4+7）となる．結果が負の場合は，最後に法Mを加算する必要がある(図4.20のSTEP3)．この計算はADDER2を利用し，4サイクルかけて処理する．バイナリ法の説明で述べたように，この乗算剰余計算は，最悪の場合2,048回，平均して1,536回行われる．

REG_Dを最後まで読み込み，乗算剰余回路での計算が終わるとREG_Pに復号した結果が格納されるが，この値はモンゴメリ形式なので通常の形式に変換する必要がある．これは$Mont(P, 1, M)$の計算を行えばよい．

ここで，加算器は自動合成で得られる最適化されたCLA（Carry Look ahead Adder）を用いている．

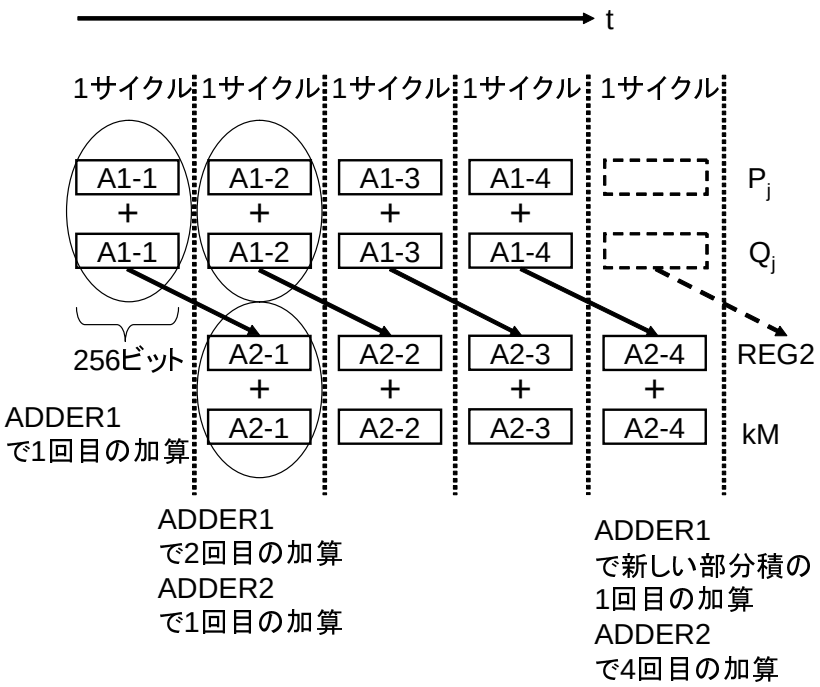

図 5.7: 4 分割の加算

5.3.3 RSA 復号回路の評価

本項では，RSA 復号回路の評価を行う．図 5.6 の回路を論理合成し回路面積を求め，その回路を用いた場合の復号処理時間について評価を行う．

RSA 復号回路は Verilog-HDL で設計し，ケイデンス社の Verilog-XL によって論理シミュレーションを行った．論理合成はシノプシス社 Design Compiler で行い，HITACHI 0.18 μm プロセス用に京都大学で作成されたスタンダードセルライブラリを用いた．なお，論理合成時の遅延制約は AES 復号回路に合わせて 5.0 [ns](動作周波数 200 [MHz]) としている．

表 5.6 に合成結果を示す．ここで，秘密鍵/公開鍵 (法) レジスタはチップ製造時にヒューズ回路を設けて，外部からデータを書き込み，検査後に外部からのアクセス回路を切断する方法で実装できると考えた．実際，現在のプロセッサには，個体ごとの識別 ID が付加されているので，同様の方式が可能と考える．ただし，書き込み後に書き込み回路自体を外部から遮断するようなヒューズ回路を設けて，復号鍵を盗まれないような方策が必要である．一般のコンピュータシステムのソフトウェアライセンス保護に利用する場合は，各個体ごとに公開鍵を異なったものにする必要があるが，組み込みシステムの組み込みプログラムのアルゴリズム保護を目的とする場合，組み込みシステムベンダごとで，あるいは，製品品種ごとで，同じ公開鍵を用いることで充分であり，部品の故障時の交換が容易に行えるなどの利点も考えられる．また，内蔵ヒューズを切る方法の場合，面積は無視で

表 5.6: RSA 復号回路の合成結果

面積 (mm^2)	クロック数 (cycle)	遅延時間 (ns)	復号処理時間 (ms)
1.75	4,212,750	5.0	21

きるほど小さいと考えられるので評価結果の面積には含めていない．

1 回の乗算剰余計算に必要なクロックサイクル数は最悪の場合，2,055 サイクル[2]であり，処理時間は 5.0ns × 2,055cycle = 10μs となる．また，法 M のビット数が 1,024 ビットであれば，すべてのビットが 1 である最悪の場合，形式変換も含め，2,050 回の乗算剰余計算が必要となる．したがって，総クロックサイクル数は 2,050× 2,055cycle = 4,212,750cycle となり，復号処理時間は 5.0ns × 4,212,750cycle = 21ms である．平均的な場合，1 回の乗算剰余計算は 2,051 サイクル，それが 1,538 回行われるので，総クロックサイクル数は 1,538× 2.051cycle = 3,154,438cycle，復号処理時間は 5.0ns × 3,154,438cycle = 16ms となる．

5.4 鍵テーブルの設計

提案システムで用いる鍵テーブルについて述べる．鍵テーブルは RSA 復号回路で復号されたプログラム鍵を格納しておくレジスタファイルである．格納するプログラム鍵の数はプログラムが起動するごとに増えていくが，本稿では，同時に実行されるプログラムはそれほど多くはないと考え，鍵テーブルのエントリ数を 16 として設計を行った．**図 5.8** にブロック図を示す．プログラム鍵のサイズは 128 ビットであるが，1 エントリサイズは 1 ビット拡張し，129 ビットとしている．

プログラムの主記憶へのローディングがプログラム鍵を RSA 復号回路で復号するよりも早く終了した場合，命令キャッシュミス時に AES 復号回路で命令の復号が行われるが，対応するプログラム鍵はまだ用意できていない．そのため，鍵テーブルには RSA 復号回路で復号しているプログラム鍵がどのエントリに格納されるかを示す flag を 1 ビット設けている．flag が立っている場合，そのエントリは，RSA で復号しているプログラム鍵が入る予定として，そのエントリのプログラム鍵を用いる AES 復号回路での復号は待たされるようになっている．RSA での復号が終わると，flag は落ちる．

鍵テーブルの面積は 5.1 節で述べた 6T-SRAM での実現を仮定しており，0.1mm^2 である．

[2]乗算剰余計算 2,051 サイクル+法の加算 4 サイクル．

図 5.8: 鍵テーブルの構成

5.5 命令用 TLB の設計

提案システムで用いる命令用 TLB について述べる．TLB は仮想アドレスと物理アドレスの対を格納しておくページテーブルの CPU 内バッファである．提案システムの命令用 TLB は，通常のものに比べて，各エントリにプログラム鍵のインデックスを格納するためフィールドが 4 ビット[3]追加されることになるが，ハードウェア量の増加はわずかである．**図 5.9** にブロック図を示す．図の短形部分は記憶素子の配列を表している．エントリは VPN(Virtual page number) フィールド，PPN(Physical page number) フィールド，ページ書き込みを記録する Dirty フィールド，エントリが有効かどうかを示す Valid フィールド，プロセスの ID を記録する PID フィールド，鍵のインデックスを記録する KEY_INDEX フィールドから成る．ページサイズは 4KB を想定しており，VPN フィールド，PPN フィールド共に 20 ビットである．また，エントリ数は 16，フルアソシアティブ方式で設計を行っている．

命令用 TLB の評価環境は AES，RSA 復号回路と同じであり，面積は $0.28mm^2$ となっている．

5.6 KEYDEC 命令の実装

提案システムでは暗号化されたプログラム鍵を復号する命令として新たに KEYDEC 命令を設けているが，本節ではその詳細について述べる．

KEYDEC 命令は**図 5.10** に示すように，OS が発行する命令であり，主記憶からの暗号化されたプログラム鍵のロード，RSA 復号回路での復号処理の開始，復号結果を鍵テーブル上のどのエントリに格納するか，を指示する命令である．

本稿では，KEYDEC 命令を実装するにあたって，対象とする命令セットとして ARM 命令セット [26] を想定している．ARM 命令セットでは，オプションとして

[3]鍵テーブルのエントリ数が 16 であるため．なお，ページテーブルも 4 ビットのフィールドが追加される．

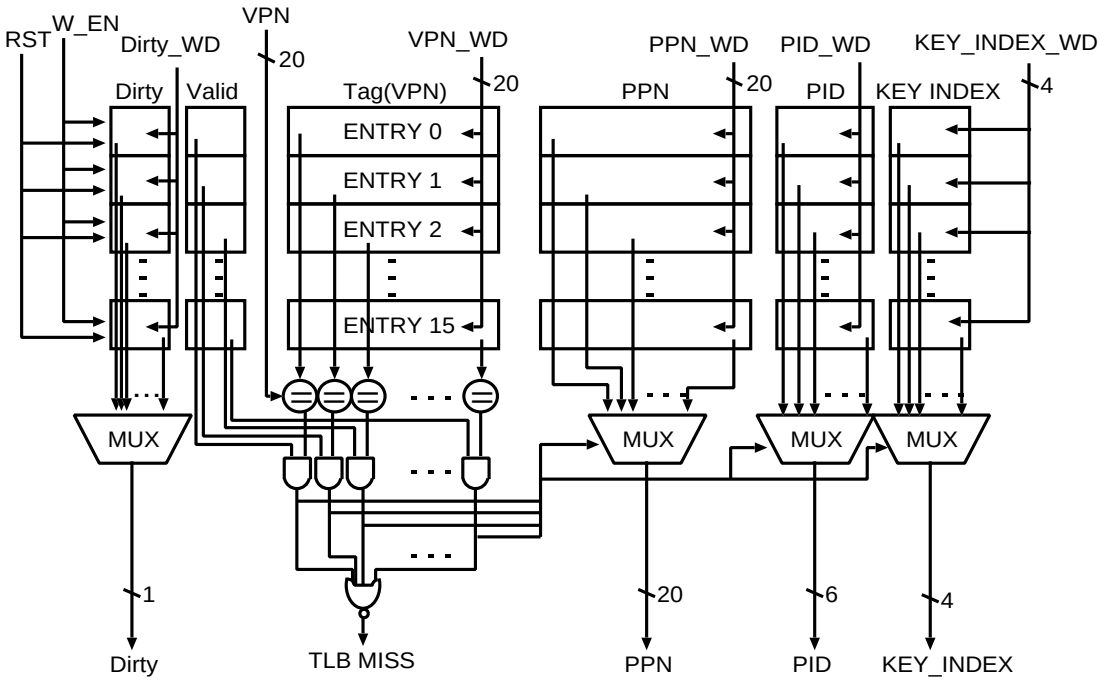

図 5.9: 命令用 TLB の構成

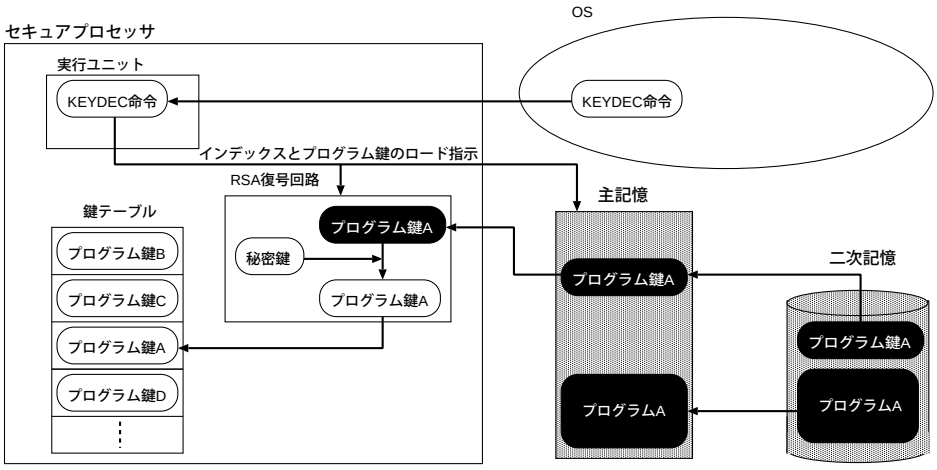

図 5.10: KEYDEC 命令の詳細

コプロセッサ用の命令が用意されており，RSA 復号回路をコプロセッサとして扱うことにより，コプロセッサ用の命令を利用して KEYDEC 命令の実装ができる．ARM 以外の命令セットでも，コプロセッサ用の命令が用意されていれば，同様の方法で実装が可能である．

まず，主記憶上の暗号化されたプログラム鍵を RSA 復号回路に転送する命令として，LDC 命令を用いる．LDC 命令は，連続したメモリアドレスのシーケンスか

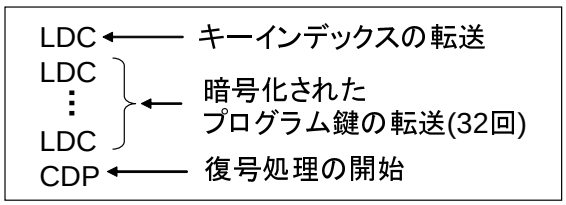

図 5.11: KEYDEC 命令

らデータをコプロセッサにロードする命令である．本稿では，LDC 命令を 32 ビットデータロードとして実装した．そのため，1,024 ビットの暗号化されたプログラム鍵を転送し終わるまで，32 回の LDC 命令を実行する仕様としている．転送データは 32 ビットごとに RSA 復号回路のレジスタ（図 5.6 の REG_C）に保存される．

RSA 復号回路は復号したプログラム鍵を保存する鍵テーブルのインデックスを知る必要があるが，これは，インデックスをデータとした LDC 命令を実行することで対応できる．LDC 命令のディスティネーションを指定することで，転送されてきたデータがインデックスなのかプログラム鍵のデータなのかを判断する．RSA 復号回路はインデックスが指定されると，鍵テーブルの対応エントリの flag を立ち上げる．

次に，復号処理の開始を指示する命令として CDP 命令を用いる．CDP 命令はコプロセッサに，ARM レジスタと主記憶から独立した演算を実行するように通知する命令である．この命令により，RSA の復号処理が開始される．

以上より，KEYDEC 命令は**図 5.11** に示す命令列で実装を行うことができる．今回の実装では KEYDEC 命令を複数の命令の集まりとして実装を行ったが，新しく KEYDEC という命令を設けて，CPU におけるデコード時に命令列に分解するという実装方法も考えられる．

5.7 追加機能のチップ面積への影響

Luehdorfia に本システムを組み込む場合，新たに追加が必要な回路は**図 3.1** の RSA 復号回路，AES 復号回路，鍵テーブルである．

表 5.7 にそれぞれの回路面積を示す．設計した TLB は Luehdorfia の面積に含めている．なお，CPU コア内のレジスタファイル，命令キャッシュ，データキャッシュの面積は 5.1 節で述べた 6T-SRAM での実現を仮定した値である．

プロセッサチップに組み込んだ場合の全体の面積は 11.32mm^2 であり，もとの約 29%の面積増加で抑えられる．この回路面積は LSI に十分実装可能な大きさである．

表 5.7: 全体の回路面積

	面積 (mm^2)
CPU コア	1.04
命令キャッシュ (8KB)	3.74
データキャッシュ (8KB)	3.74
TLB	0.28
(Luehdorfia 合計)	8.80
AES 復号回路	0.67
RSA 復号回路	1.75
鍵テーブル	0.10
(追加回路合計)	2.52
合計	11.32

5.8 プロセッサの性能への影響

5.8.1 プログラムの復号処理時間

本論文では，キャッシュミス時に，主記憶からプログラム（命令）を命令キャッシュに転送する過程で復号処理を行うので，復号処理にかかる時間は，プロセッサの通常のキャッシュミスペナルティに加算されることとなる．ここで，キャッシュミス時に主記憶から転送されるラインサイズのプログラムの復号を考えた場合に，復号処理時間はラインサイズやメモリバスの転送能力などによって変化する．例えば，ラインサイズが 16B（128 ビット）の場合は，AES-128 復号回路を用いて 1 回の復号処理でラインサイズの全てのデータを復号できるが，ラインサイズが 32B の場合は，全てのデータを復号するためには 2 回の復号処理が必要となる．本論文では，Luehdorfia を AES-128 復号回路を搭載するプロセッサと仮定し，キャッシュミス時に要する復号処理時間について調査した．また，キャッシュラインサイズやメモリバスの転送能力が復号処理時間に与える影響について述べる．

Luehdorfia のキャッシュと主記憶に関係する仕様を以下に述べる．

- Luehdorfia は，主記憶として SDRAM PC-100 CL3 を想定している．プロセッサの動作周波数が 200 MHz であり，メモリバスの動作周波数は 100 MHz である．
- キャッシュラインサイズは 16B，メモリバス幅は 64 ビットである．キャッシュミス時には，主記憶から 64 ビット毎に 2 回のバースト転送が行われる．プロセッサに到着したデータは，メモリコントローラによって 32 ビット毎にキャッシュに転送される．

- キャッシュミスが発生してから 22 サイクル後に 1 回目のデータがメモリコントローラに到着する．その 2 サイクル後（メモリサイクルでは 1 サイクル後）に 2 回目のバースト転送分のデータが到着し，キャッシュラインサイズの命令が全て到着する．

- プロセッサは，キャッシュミスを発生させたデータがメモリコントローラからキャッシュに転送される際に，バイパスしてプロセッサに取り込む．そのため，プロセッサに見えるキャッシュミスペナルティは，キャッシュミスを発生させたデータがキャッシュラインの最初である場合は 23 サイクルとなり，キャッシュラインの最後である場合は 26 サイクルとなる．

以上の条件を基に，AES-128 復号回路を用いた復号処理について考える．まず，キャッシュラインサイズは 16B となっており，AES-128 のブロックサイズ 1 個分であるので，AES-128 復号処理 1 回で全てのデータを復号できる．キャッシュミスから 24 サイクル後にブロックサイズのデータがプロセッサに揃い，その後，10 サイクルかけて復号処理を行うので，キャッシュミスから 34 サイクル後に復号処理が完了する．復号結果である平文のデータは，通常通り 32 ビット毎にキャッシュに転送され，プロセッサはそこからキャッシュミスを発生させたデータをバイパスするため，プロセッサに見えるキャッシュミスペナルティは 35〜38 サイクルとなる．以上から，AES-128 復号回路を Luehdorfia に搭載し，キャッシュミス時に復号処理を加えた場合，復号処理時間 10 サイクルと復号処理を行うブロックサイズが揃うまでの待ち時間 2 サイクルを合計した 12 サイクルがキャッシュミスペナルティに加算される．5.2.2 項では，このキャッシュミスペナルティの増加が，プログラムの実行時間にどの程度の影響を与えるかを調査する．

次に，キャッシュラインサイズとメモリバスの転送能力が復号処理時間に与える影響について述べる．前述したようにキャッシュラインサイズが 32B の場合は，AES-128 復号回路で 2 個分のブロックサイズのデータを復号する必要がある．32B のデータのうち前半のブロックサイズのデータは，プロセッサに全てのデータが到着次第復号処理が行われるが，後半のブロックサイズのデータは，全てのデータがプロセッサに到着しても前半のデータの復号処理が終わるまで待たなければならない．ここで，Luehdorfia と同様のメモリバスの転送能力を想定すると，後半のブロックサイズのデータは復号処理が開始して 4 サイクル後に全てのデータがプロセッサに到着し，復号処理が終了するまでの 6 サイクルの待ち時間が発生する．そのため，キャッシュミスを発生させたデータが後半のブロックサイズである場合は，先程の 12 サイクルに加えて復号処理待ち時間の 6 サイクルが加わった 18 サイクルがキャッシュミスペナルティの増加分となる．復号処理待ち時間は，キャッシュラインサイズが大きくなり，キャッシュミスを発生させたデータが後半にあるほど長くなる．

また，メモリバスの転送能力も復号処理待ち時間に影響する．プロセッサとメモリバスの動作周波数を固定して，メモリバス幅を 32 ビットとすると，1 個分の

ブロックサイズのデータを転送するのに 8 サイクルかかるので後半のブロックサイズのデータの復号処理待ち時間は 2 サイクルとなり，反対にメモリバス幅を 128 ビットとすると，復号処理待ち時間は 8 サイクルとなる．したがって，メモリバスの転送能力が上がるほど復号処理待ち時間は長くなる．

以上のような復号処理待ち時間によるキャッシュミスペナルティの増加を防ぐには，キャッシュミス時に主記憶からプロセッサへデータを転送する際に，キャッシュミスを発生させたデータを最初に転送させるような機構 (Critical word first) の適用が考えられる．上記手法を適用すれば，キャッシュラインサイズが増加した場合でも，復号回路で最初に処理するブロックサイズの中にキャッシュミスを発生させたデータが含まれることになるので，復号処理待ち時間自体を消去することができる．

5.8.2 ソフトウェアシミュレータによる評価

5.2.1 項から Luehdorfia に本論文で提案するプログラム保護機能を持たせた場合，キャッシュミスペナルティが 12 サイクル増加することを述べた．そこで本項では，本研究室で作成されたプロセッサの性能評価シミュレータ (asim)[32] を用いて Stanford Benchmark と MiBench[21] の一部のベンチマークプログラムを実行し，復号処理が加わった際の実行時間の増加を評価した．

5.8.2.1 asim の概要

asim は，ARM アーキテクチャを対象とした性能評価シミュレータである．asim の基本構造は，Intel 社の XScale[31] 参考にしている．XScale では，データ処理命令，ロード/ストア命令，乗算命令をそれぞれ独立したパイプラインで処理しており，asim はこれをモデル化しデータ依存によるパイプラインのストールを算出している．asim の特徴を以下に述べる．

- ARMv5 命令セットアーキテクチャ（Thumb 命令及び DSP 命令を除く）
- トータルクロック数，キャッシュヒット/ミス数，命令出現頻度などを算出
- プログラム実行中の 1 命令毎に，汎用レジスタ，ステータスレジスタ，命令/データキャッシュの値を観測可能

asim で実行するオブジェクトプログラムは，Linux システム上の ARM クロスコンパイル環境において生成され asim は，その ELF 形式の実行プログラムを読み込み実行する．プログラム中で発行されるシステムコールは，シミュレータを実行しているマシン上の OS に肩代りさせて実行している．プログラム中の入出力も，ダミーのライブラリルーチンを用意し，fprintf や fscanf のレベルでホスト OS の機能として肩代りさせて実行している．また，XScale では除算命令や浮動小数点演

算命令をサポートしていないので，プログラム中の該当部分に対してはコンパイラが組み込み関数に置換して実現される．

5.8.2.2 ベンチマークの概要

本評価では，ベンチマークプログラムとして Stanford Benchmark と MiBench を用いた．Stanford Benchmark，MiBench ともに C 言語で記述されたベンチマーク・スートである．

5.8.2.2.1 Stanford Benchmark Stanford Benchmark は，John Hennessy らによってスタンフォード大で開発され “Perm”, “Towers”, “Queens”, “Intmm”, “Puzzle”, “Quick”, “Bubble”, “Tree” の整数型のベンチマークプログラムが 8 個と，“Mm”,“FFT” の単精度浮動小数点数のベンチマークプログラム 2 個の計 10 個から構成される小規模なベンチマーク・スートである．Stanford Benchmark の全てのベンチマークプログラムは，入力ファイルを必要としない．

5.8.2.2.2 MiBench MiBench は，Matthew R. Guthaus らによってミシガン大で開発された組み込みプロセッサ向けのベンチマーク・スートである．MiBench では，様々な場面で用いられる組み込みプロセッサの多様性を反映し，“Automotive and Industrial Control”, “Network”, “Security”, “Consumer Devices”, “Office Automation”, “Telecommunications” の 6 つのカテゴリから成る計 35 個のベンチマークプログラムが無料で提供されている．MiBench のベンチマークプログラムの一覧を表 5.8 に示す．本論文では，表 5.8 のベンチマークプログラムの中で太字で示されたベンチマークプログラムを評価に用いた．評価に用いたベンチマークプログラムの概要を以下に示す．

- **bitcount** 整数型配列のビット数を数えることによって，プロセッサのビット操作能力をテストするプログラムである．
- **qsort** クイックソートアルゴリズムを使用して，文字列が格納された巨大な配列を昇順に並び替えるプログラムである．
- **susan** 画像認識を行うプログラムである．脳の磁気共鳴映像のコーナーやエッジを認識するために開発され，画像のスムージングなどの処理も行う．
- **dijkstra** 入力グラフを隣接行列で表現し，1 節点から全節点への最短経路をダイクストラ法を用いて計算するプログラムである．
- **sha** 与えられた入力データから，160 ビットのメッセージダイジェストを生成するセキュアハッシュアルゴリズムを計算するプログラムである．MD4 や MD5 のハッシュ関数として用いられている．

表 5.8: Mibench Benchmarks

Auto./Industrial	Consumer	Office
basicmath	jpeg	ghostscript
bitcount	lane	ispell
qsort	mad	rsynth
susan (edges)	tiff2bw	sphinx
susan (corners)	tiff2rgba	stringsearch
susan (smoothing)	tiffdither	
	tiffmedian	
	typeset	
Network	Security	Telecomm.
dijkstra	blowfish enc.	CRC32
patricia	blowfish dec.	FFT
(CRC32)	pgp sign	IFFT
(sha)	pgp verify	ADPCM enc.
(blawfish)	rijndael end.	ADPCM dec.
	rijndael dec.	GSM enc.
	sha	GSM dec.

5.8.2.3 asim 実行結果

Stanford Benchmark と MiBench の bitcount, qsort, susan, dijkstra, sha の各ベンチマークプログラムを asim で実行した．Linux 上に構築したクロスコンパイル環境で，各ベンチマークプログラムの実行プログラムを作成した．gcc のバージョンは 3.4.1，コンパイルオプションは armv3-softfloat-linux-gcc -static -nostartfiles -msoft-float -march=armv3 -O2 である．MiBench の各ベンチマークプログラムの入力データを表 5.9 に示す.

asim の命令キャッシュの構成は，XScale を参考にしているため 32KB，ラインサイズ 32B，32 ウェイセットアソシアティブ方式となっており，組み込みプロセッサの中では，豊富な命令キャッシュを有しているといえる．ここで，他の組み込みプロセッサではこれよりも小規模な命令キャッシュを搭載していることも考えられるため，各キャッシュサイズにおいてベンチマークプログラムを asim で実行し，そのヒット率を調査した．キャッシュサイズの変更はウェイ数の設定により行った．結果を表 5.10 に示す.

表 5.10 より，キャッシュサイズが 1KB の場合の susan (edges) の命令キャッシュヒット率が 95.2%と最も低くなっている．実験より，キャッシュサイズを 8KB 以上にすると全てのベンチマークプログラムにおいて，命令キャッシュヒット率は 99.9%を上回った.

表 5.9: 各ベンチマークプログラムの入力データ

	Format	Size
bitcount	array	75000
qsort	string	10000
susan	pgm image	76×95 pixel
dijkstra	matrix	100×100
sha	text	311,824 Byte

次に，復号処理が加わった際の実行時間の増加を評価する．命令キャッシュのヒット率は，キャッシュサイズ 1KB の場合を用いることで最も厳しい制約条件下での評価となっている．5.2.1 項から，復号処理なしの通常の命令キャッシュのミスペナルティを 24 サイクル，復号処理ありの命令キャッシュのミスペナルティを 36 サイクルとし，それぞれのキャッシュミスペナルティの場合の総実行クロック数を表 5.11 に示す．

5.8.2.4 考察

表 5.10 より，キャッシュサイズが 1KB の場合でも全てのベンチマークプログラムにおいて命令キャッシュヒット率が 95%を上回った．ここで，キャッシュを搭載している一般的な組み込みプロセッサであれば，命令キャッシュは 1KB 以上搭載していると予想されるため，今回実行したベンチマークにおいて，命令キャッシュのヒット率は，95%以上になる考えられる．

また，表 5.11 より，実行時間増加の割合は最大で 1.20 倍となり，かなり小規模な命令キャッシュのサイズであっても 2 割程度の実行時間の増加で抑えることができている．また，命令キャッシュのサイズが増加すれば，本提案の実現によるプロセッサの性能低下は更に減少する．

今回，提案するプロセッサの適用フィールドとして組み込みプロセッサを想定していたため，Stanford Benchmark や MiBench など比較的小規模なベンチマーク・スートで調査を行った．今後は汎用プロセッサへの適用も考え，SPEC などのベンチマークでの命令キャッシュのヒット率の調査も行いたい．

5.9 RSA 復号回路の性能への影響

RSA 復号回路による処理時間はプログラムを二次記憶から主記憶にローディングしている間に並行して行うとすると，処理時間を隠蔽できることが期待できると述べたが，実際にどの程度隠蔽できるのかを調査した．

主記憶と二次記憶を結ぶバスの構成や，用いる二次記憶装置の種類によって転送処理速度等が異なってくるため，正確な測定はできないが，ある程度の評価の

表 5.10: 各キャッシュサイズにおける命令キャッシュのヒット率

Stanford	1KB	2KB	4KB	32KB
Perm	99.9%	99.9%	99.9%	99.9%
Towers	99.9%	99.9%	99.9%	99.9%
Queens	99.8%	99.8%	99.8%	99.9%
Intmm	98.6%	99.9%	99.9%	99.9%
Puzzle	99.9%	99.9%	99.9%	99.9%
Quick	97.4%	99.9%	99.9%	99.9%
Bubble	99.9%	99.9%	99.9%	99.9%
Trees	97.0%	99.9%	99.9%	99.9%
Mm	99.5%	99.9%	99.9%	99.9%
FFT	96.5%	99.9%	99.9%	99.9%
MiBench				
bitcount	99.9%	99.9%	99.9%	99.9%
qsort	99.8%	99.9%	99.9%	99.9%
susan (edges)	95.2%	96.7%	99.1%	99.9%
susan (corners)	95.8%	97.8%	99.5%	99.9%
susan (smoothing)	99.3%	99.6%	99.9%	99.9%
dijkstra	99.5%	99.9%	99.9%	99.9%
sha	99.5%	99.9%	99.9%	99.9%

目安になると考え，以下のような条件のもとで性能への影響がどれくらいであるのかを考える．

- 二次記憶の平均シーク時間は 15 ms であり，ディスクの回転速度は 4,200 rpm, 平均内部転送速度は 26 MB/s である[4].
- 外部インターフェースは DMA100 で接続されており，転送速度は 100 MB/s である．
- プログラムのサイズは Stanford Benchmark suite を参考に 554KB とする．この内，主記憶にマッピングされるデータサイズはおよそ 439KB である．
- プログラムは ELF 形式を想定し，プログラムが起動されるとまず，.key セグメントが二次記憶から主記憶にローディングされる．key セグメント内の暗号化されたプログラム鍵が CPU を介し RSA 復号回路に送られ，復号処理を

[4] これらの性能値は組み込みシステムの一例として Apple 社製の携帯音楽プレーヤー iPod に用いられている東芝製 HDD，MK3006GAL の値を参考にしている．

表 5.11: asim 実行結果

Stanford	通常時	復号処理有り	I$ hit ratio
Perm	2,760,596	2,760,878 (1.00)	99.9%
Towers	2,802,314	2,802,638 (1.00)	99.9%
Queens	33,966	34,229 (1.01)	99.8%
Intmm	1,827,057	1,980,585 (1.08)	98.6%
Puzzle	11,082,916	11,083,684 (1.00)	99.9%
Quick	1,827,791	2,068,079 (1.13)	97.4%
Bubble	1,700,228	1,712,468 (1.01)	99.9%
Trees	7,216,612	8,176,336 (1.13)	97.0%
Mm	14,098,480	14,498,976 (1.03)	99.5%
FFT	813,998	942,866 (1.16)	96.5%
MiBench			
bitcount	86,952,070	86,958,609 (1.00)	99.9%
qsort	48,250,708	48,953,410 (1.01)	99.8%
susan (edges)	8,513,606	10,233,964 (1.20)	95.2%
susan (corners)	5,219,322	6,125,693 (1.17)	95.8%
susan (smoothing)	35,001,513	36,673,096 (1.05)	99.3%
dijkstra	93,971,207	97,001,026 (1.03)	99.5%
sha	20,096,791	20,804,839 (1.04)	99.5%

行っているあいだに，プログラムの残りのセグメントが主記憶にローディングされる．

以上の条件のもとで，プログラムの.key セグメント以外のセグメントが主記憶にローディングされるのにどれほど時間がかかるのかを調べる．かかった時間が RSA 復号処理を隠蔽できる時間となる．

平均ローディング時間として，平均ディスクアクセス時間を求めることでローディング時間を見積もる．これは二次記憶からデータの読み出しを行うのに必要な時間である．平均ディスクアクセス時間は平均シーク時間，平均回転待ち時間，転送時間の合計で算出する．ただ，.key セグメントとその他のセグメントはディスク上ですぐ近くに位置するはずなので，.key セグメントの位置をシークした後は，残りのセグメントの位置をシークする時間はそれほどかからないはずである．よってシーク時間をほぼ無視できるものとした．平均ディスクアクセス時間は以下のようになる．

$$\frac{0.5\text{回転}}{4,200rpm} + \frac{439KB}{26MB/s} = 7.1 + 16.9 = 24ms$$

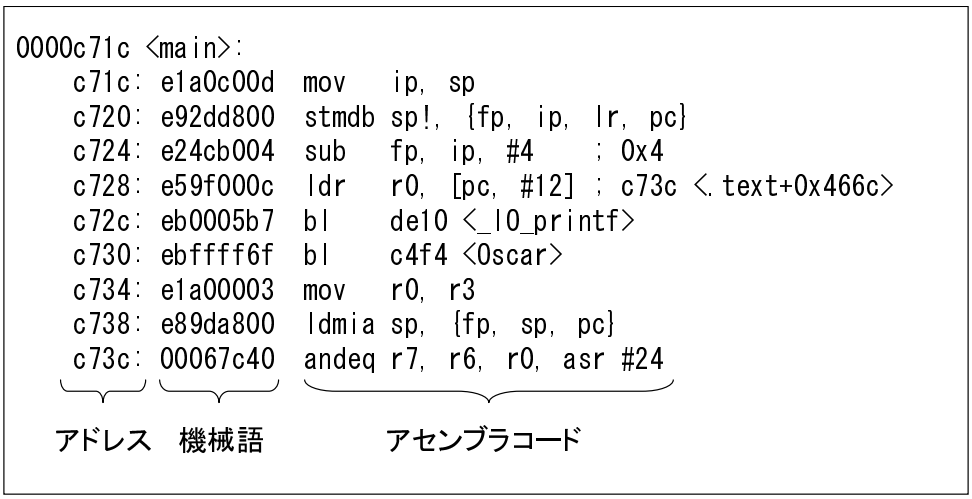

図 5.12: FFT のコードの一部（機械語とアセンブラコード）

RSA の復号処理時間は最悪の場合 21*m*s なので一般的にほとんどの復号時間を隠蔽できることが期待できる．

5.10 コンパイラのコード生成に関する留意点

提案システムでプログラムを動作させる上で，現状提供されているコンパイラの生成するコードでは問題があることが分かった．本節ではその問題についてと，どのようなコードを生成すればよいのかについて説明する．

提案システムでは暗号化されたプログラムのみを扱うため．シミュレーション時に実行するプログラムは暗号化したものを用意する必要がある．そのため，3.3 節で説明した方法でオブジェクトファイルを暗号化するプログラムを作成した．一例として Stanford Benchmark のオブジェクトファイルを暗号化したものを用いて論理シミュレーションを行った際の問題点について説明する．このオブジェクトファイルには AES で暗号化された命令列と RSA で暗号化されたプログラム鍵を含んでいる．このオブジェクトファイルを Verilog-HDL で記述されたメモリにマッピングし，論理シミュレーションを行った．

論理シミュレーションの結果，ロード命令時に予定された値とは異なる値をロードしようとする問題点があることが分かった．**図 5.12** は Stanford Benchmark の 1 つである FFT における暗号化されていないオブジェクトファイルの.text セクションのコードの一部分を示したものである．また，**図 5.13** は該当部分の C 言語記述である．

5 行目のロード命令 ldr はレジスタ r0 に 0xc73c 番地の内容をロードするというものであるが，問題点はデータを保存しておく 0xc73c というアドレスが.text セクション内に配置されていることである．0xc73c 番地の内容の 0x67c40 は**図 5.14**

```
main()
{
printf("FFT¥n"); Oscar();
}
```

図 5.13: FFT のコードの一部（C 言語）

```
67c1c 72332069 6e204275 62626c65 2e0a0000  r3 in Bubble....
67c2c 0a000000 20202531 352e3365 2531352e  ....   %15.3e%15.
67c3c 33650000 4646540a 00000000 2f70726f  3e..FFT...../pro
67c4c 632f7379 732f6b65 726e656c 2f6f7372  c/sys/kernel/osr
67c5c 656c6561 73650000 46415441 4c3a2063  elease..FATAL: c
```

図 5.14: FFT のコードの一部（.rodata セクション）

に示しているが，これは文字列リテラルや定数を保持する.rodata セクションに位置しており，文字列 ”FFT ”を保持している．実際，シミュレーション時には ldr 命令によって，0xc73c に格納されているアドレスのロードが行われるが，.text セクションは暗号化されているため，0xc73c 番地の内容も暗号化されてしまっている．提案システムでは，実行時に命令は復号されるが，データは暗号化しないので，平文のままで扱う．データとして読み込んだ 0xc73c 番地の内容は当然復号されないので，予定と異なる値がロードされてしまうのである．つまり，.rodata セクションのアドレスを.text セクションのデータとして扱っていることが問題である．これは，オブジェクトコードを出力したコンパイラの仕様である．なお，コンパイラは gcc version 3.4.1 を ARM 用にビルドしたクロスコンパイラを用いている．ARM の命令セットではプログラムカウンタが汎用レジスタ 15 番として定義されており，アドレッシングの際のベースレジスタとしてこの汎用レジスタ 15 番を指定し，プログラムカウンタからのオフセットを与えてやることで比較的簡単にディスティネーションアドレスを計算できる仕様になっている[5]．データを.data セクションや.rodata セクションに分離しないのは，この仕様によるものと考えられる．解決策として，データを完全に.text セクションから分離するようにコンパイラを改良することが考えれるが，これは今後の課題である．

[5]ARM 命令セットでは PC は通常 2 つ先の命令を指している．図 5.12 では PC の指す 0xc734 にオフセット 12 を加えた 0xc73c がディスティネーションアドレスとなる．

第6章　保護能力評価

我々の提案プロセッサではプログラム内に含まれる重要なアルゴリズムを保護することに重点を置き，実行プログラム本体のみを暗号化の対象として，プログラムの処理対象となるオペランドデータの暗号化は行なっていない．2.3 節で示す他の先行研究では，プログラムのオペランドデータと割込み発生時のコンテキストデータの暗号化も検討されている．しかし，データの暗号化を行うということは，主記憶からデータを読み込んで復号を行い，主記憶へ書き戻す際に暗号化を行うことになる．これを悪用することで，暗号化されたプログラムをデータとして読み込み，主記憶へ書き戻すことでプログラムを解読される危険性が有り得ると考えられた．よって，我々の提案プロセッサではあえてデータの暗号化を行わないとしている．

オペランドデータを観測されることで保護すべきアルゴリズムを推定される危険性が存在しているが，保護すべきアルゴリズムが充分に複雑であればオペランドデータによる命令列の同定も充分に困難だと考えていた．ただ，この予測に基づいた根拠はなく，本当に困難であるのか調査されていなかった．そこで本研究ではオペランドデータによる命令列の同定が，どの程度の困難かを調査しその度合を評価することで，提案システムにおけるプログラムの保護能力の評価を行うこととした．

6.1　ARM アーキテクチャ

本研究で評価を行うプロセッサは命令セットとして，ARM アーキテクチャ[27, 28, 29] を採用し検討を進めている．ARM アーキテクチャの基本構成や特徴について簡単に説明を行う．

ARM アーキテクチャは，組込み機器に広く使用されている 32 ビットの RISC プロセッサのアーキテクチャである．ユーザレベルのプログラムでは，15 本の 32 ビット長の汎用レジスタ (r0〜r14)，32 ビット長のプログラムカウンタ (r15)，条件コードフラグを含むカレント・プログラム・ステータス・レジスタ (CPSR) が使用される．PC はユーザから可視の状態にあり，ユーザは PC を任意に変更でき演算にも使用できる．演算に使用した場合は使用した命令が格納されているアドレス+8 の値として用いられる．条件コードフラグは CPSR レジスタの最上位 4 ビットで示され次の意味を持っている．

表 6.1: ARM アーキテクチャの条件コード

OP コード	ニモニック	意味	実行のためのフラグ
0000	EQ	等しい	Z == 1
0001	NE	等しくない	Z == 0
0010	CS/HS	符号無しで大きいかまたは等しい	C == 1
0010	CC/LO	符号無しで小さい	C == 0
0100	MI	負	N == 1
0101	PL	正またはゼロ	N == 0
0110	VS	オーバーフローあり	V == 1
0111	VC	オーバーフローなし	V == 0
1000	HI	符号無しで大きい	C == 1 and Z == 0
1001	LS	符号無しで小さいかまたは等しい	C == 0 or Z == 1
1010	GE	符号付きで大きいかまたは等しい	N == V
1011	LT	符号付きで小さい	N != V
1100	GT	符号付きで大きい	Z == 0 and (N == V)
1101	LE	符号付きで小さいかまたは等しい	Z == 1 or (N != V)
1110	AL	常に	条件なし

- N(負:Negative)，フラグ更新を行う最後の ALU 処理が，負の結果を生成.
- Z(ゼロ:Zero)，フラグ更新を行う最後の ALU 処理が，ゼロの結果を生成.
- C(桁上げ:Carry)，フラグ更新を行う最後の ALU 処理が，ALU 算術計算もしくはシフタによる桁上げを生成.
- V(あふれ:oVerflow)，フラグ更新を行う最後の ALU 処理が，符号ビットにあふれを生成.

ARM アーキテクチャの特徴に，全命令に対する条件付き実行が可能という点が挙げられる．条件付き分岐は他の多くのアーキテクチャで実装されているが，ARM アーキテクチャではすべての命令に拡張されている．条件フィールドは 32 ビット命令フィールドの最上位 4 ビットで表現され 15 種類の条件実行が可能である．条件付き実行の状態や意味を表 6.1 に記す．OP コードは命令フィールドの最上位 4 ビットを示し，表中のフラグ状態を満たす場合に限り命令が実行される．条件を満たさなければ命令は実行されずにスキップされ，NOP と同等となる．

6.2 同定アルゴリズム

オペランドデータの暗号化を行っていない我々の提案システムでは，コンテキスト退避の際にレジスタデータなどが暗号化されずに主記憶へ格納されてしまう．これらの情報から保護したいプログラムの命令列が，現実的な時間で同定されてしまう危険性があるのかを，実際に情報を基に解析を試みることで調査を行った．その解析方法を本節及び 6.3 節に記す．

6.2.1 前提条件

本評価の前提環境として，セキュアプロセッサにとって非常に悪い環境を想定して解析を行う．そのための前提条件を以下に記す．このような攻撃者に有利な環境においてさえも保護強度を証明できれば，強度が充分であることを示せると考えたからである．

- **周辺回路の構成**
  セキュアプロセッサ以外の周辺ハードウェア回路は自由に構成できるとする．メモリバスといったプロセッサ外部に存在するバスも観測できるとする．

- **オペレーティング・システム**
  セキュアプロセッサに付属するはずの本来の正規 OS は，暗号化されているためにハッキングは不可能である．そこで，自ら作成するなどを行って，攻撃者の思い通りに動作する OS を用意して解析を行うと考える．これにより，攻撃者は OS 特権機能を利用できるとする．

- **主記憶装置**
  ハードウェア構成の自由化に伴い，主記憶上の値は任意の時点で自由に閲覧及び改竄が可能と考える．また，解析を行う下準備として，主記憶上の値を一定の法則に基づいた値で初期化しておく．ロード命令を行った際にロードされた値を得ることで，ロードアドレスを得ることが可能となるようにするためである．本研究では，主記憶をワードサイズ単位で，その格納アドレスに任意のオフセット値を加算する法則とした．具体的には，オフセット値を 16 進数で 0x80 と設定し，例えば 0x00100000 のアドレスには 0x00100080 のデータが格納されているとした．ただし，プログラム領域などの書き換えることができない領域が存在するが，本研究ではその例外は考慮していない．

- **解析対象プログラム処理の逐次実行**
  外部割込みなどの方法を用いて，解析対象プログラムを 1 命令処理するごとに中断できるとする．割込みを発生させることにより OS に制御が戻され，レジスタなどのコンテキストデータを主記憶に退避することで，それらのデータの取得が可能となる．

- **データキャッシュの無効化**
  プロセッサには通常キャッシュが搭載されていて，ストア命令の反映はまず先にキャッシュに対して行われ，主記憶への反映には時間差が存在する．またロード命令ではキャッシュ上に要求データが存在すると，主記憶にはアクセスを行わない．この状態ではロードストア命令の解析に都合が悪い．そこでデータキャッシュの無効化を目的として，解析を行うプログラムに制御がある間に，解析の対象となるプログラムとは無関係なアドレスのデータをロードすることで，データキャッシュに存在するデータを全て入れ替える．これによりロード命令は，実行時にキャッシュミスが発生して主記憶にアクセスを行い，アドレスバスをプローブすることでロードアドレスがキャッシュライン単位で判明する．ストア命令は，解析プログラムに制御を戻した後，キャッシュのデータを入れ替えて変更されたデータの書き戻しを発生させることで，主記憶にストア結果を反映させる．以上の動作により事実上のキャッシュ無効化が可能であるとみなして解析を行うとする．加えて，実行された命令が主記憶にアクセスを行うものであるかどうかも判明する．

上記の環境条件により，1 命令実行前後のコンテキスト及び主記憶値から実行された命令の同定を試みる．

6.2.2 解析対象命令

6.2.2.1 候補命令

解析対象とするプログラム内で使用されている命令に，ユーザモードで使用される主要な命令を想定した．それを以下に記す．

- 分岐命令
- カウント・リーディング・ゼロ (CLZ) 命令
- 乗算命令 (32 ビット，64 ビット)
- データ処理 (数値演算，数値比較)
- データ転送 (シングル/マルチ，ロード/ストア)

データ処理の区分として，数値演算とは，算術や論理演算の命令であることを示し，数値比較は結果をレジスタに反映させずにフラグのみを更新する命令のことを指す．ARM ではデータ転送命令に，単一レジスタに対して転送を実行するシングル系転送命令に加えて，一括し複数のレジスタに対して転送を実行するマルチ系転送命令が存在する．

6.2.2.2 除外命令

本研究では想定しない命令を以下に記す.

- ソフトウェア割込み命令
- メモリ-レジスタ間のスワップ命令
- Thumb 関連命令
- コプロセッサ関連命令
- 他，プロセッサ設計者独自の命令など

ソフトウェア割込み命令 (スーパーバイザ・コール) は，ユーザモードで使用される頻度が高いと思われるが，この命令は OS の機能提供を OS に依頼する命令である．OS を制御できる状態であれば全て検知・同定可能として本研究では除外した.

スワップ命令は主にセマフォの構築に使用され，マルチプロセッサ環境では広く使用される命令であるが，シングルプロセッサ環境ではほとんど使用されない命令である．提案システムでは，現段階においてマルチプロセッサ環境を想定していないため，本研究では除外することとした.

また，ARM プロセッサには，Thumb(サム) と呼ばれる 16 ビット長のインストラクションモードが存在するが，機能は 32 ビット長命令のサブセットであるため本研究では除外した.

コプロセッサ関連の命令も存在するがそれも除外した.

その他，ユーザプログラムが使用しない命令 (主に OS 特権命令) や，ソフトウェアデバッグを行うためのブレークポイント命令，未使用の命令空間に当てはめられる ARM 基本仕様にないプロセッサ開発者独自の命令も除外した.

6.2.3 解析フロー

この節では，解析の対象となるプログラムに対し 1 命令ずつ同定を行っていく方法を記す.

同定アルゴリズム内でレジスタやフラグを参照・変更する動作が含まれているが，これは主記憶に退避されたコンテキストに含まれるデータに対して行うものであり，プロセッサ内部に存在するレジスタに直接操作を行う意味ではない．以降，単に「レジスタを変更する」といった表現を使用する.

6.2.3.1 解析処理全体の流れ

同定アルゴリズムにおける処理の流れを以下に示す.

1. 解析対象となるプログラム (以降，対象プログラム) を実行する.
2. 外部割込みにより，解析を行うプログラム (以降，解析プログラム) に制御を移す.
3. 対象プログラムのコンテキスト退避を行う.
4. データキャッシュの無効化を行う.
5. 対象プログラムの退避コンテキスト及び主記憶の使用領域の値を「実行前情報」として記録する.
6. 対象プログラムのコンテキスト復元を行い，対象プログラムに制御を移す. その際リターンアドレスを，解析したい部分のコードを指すように改竄して制御を移す.
7. 対象プログラムを実行する.
8. 外部割込みにより，解析プログラムに制御を移す.
9. 対象プログラムのコンテキスト退避を行う.
10. データキャッシュの無効化を行う.
11. 対象プログラムの退避コンテキスト及び主記憶の使用領域の値を「実行後情報」として記録する.
12. 記録された実行前後の情報を基に解析を行う.
13. 同定が完了していれば解析終了もしくは次に解析したい部分を定めて継続する．同定が完了していなければ，同定アルゴリズムに基づき退避コンテキストを改竄する.
14. 5 へ戻り解析処理を継続する.

上記の 12 にあたる解析部分では毎回，実行前後の情報を比較して，どの番号のレジスタに変化が発生したのか，どのアドレスの主記憶値に変化が発生したのかを検出する．本研究で比較対象にしているのは，汎用レジスタ (r0〜r14)，PC(r15)，条件コードフラグ (CPSR 内の NZCV)，主記憶の領域である．ある 1 命令の解析中に一度でもレジスタやフラグを変更すると，その履歴を記録しておく．PC に関しては，通常カウントアップ値 (+4) 以外の値になると，変化したとみなす.

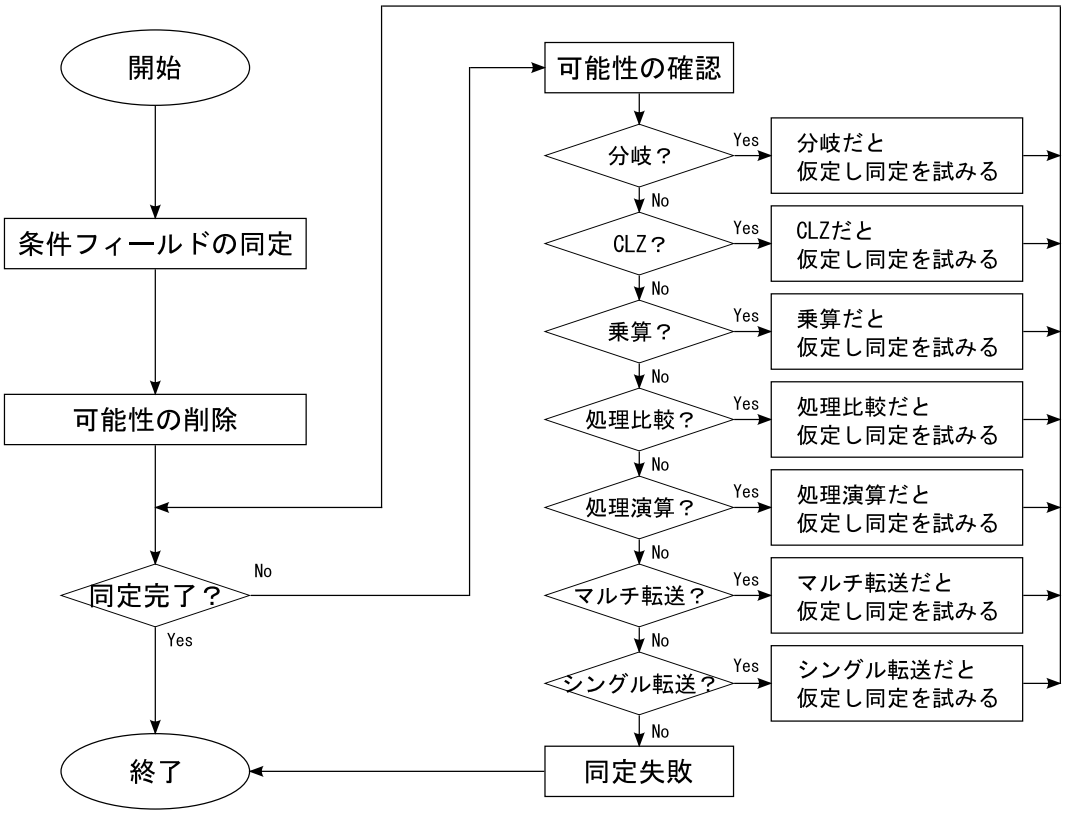

図 6.1: ある 1 命令を解析する場合の論理フロー

何かしらの変化が発生したのであれば何かしらの命令が実行されたという事実を得ることができる．そうでなければ，命令が NOP 命令であったか，条件付き実行でスキップされたのか，実行されたにもかかわらず偶然変化を観測できなかった可能性が存在する．変化を観測できない場合というのは，書き込み先のレジスタ値が書き込みデータと一致した場合などである．例えば，ゼロ値だったレジスタにゼロ値を代入しても，変化は観測できないが実は命令は実行されている．また，実行されているにもかかわらずどんな状況においても変化が観測できない命令というのは，NOP 命令かそれに同等な命令に他ならない．

6.2.3.2 節に，解析部分で行う論理について述べる．

6.2.3.2　解析論理の進め方

図 6.1 に，ある 1 命令の同定を行う場合の解析論理の進め方を記す．6.2.3.1 節の 12 の解析部分で，図 6.1 に記した論理を徐々に進めていき，所々の論理に応じてコンテキストデータを変更して 6.2.3.1 節の 13 に戻り，命令を実行し新たな情報を得て論理を進めていくことで最終的な同定完了を目指す．

開始後，最初にARMアーキテクチャの特徴である条件フィールドの同定を試みる．最初に行う理由としては，条件付き命令であった場合に以降の解析で命令がスキップされることがないように，条件コードフラグを操作して命令が確実に実行されていることを保証するためである．条件フィールドの同定方法を6.2.3.3節に記す．

条件フィールドの同定が終わると，次はそれまでに発生した変化の履歴を基に，その変化を起こし得ない候補命令の可能性を削除する．例えば，解析中フラグが変化した履歴があれば，実行された命令はフラグを変更できない命令では決して有り得ない．可能性の削除を6.2.3.4節に記す．

可能性の削除を終えると，あらかじめ命令の種類ごとにグルーピングを行っておき，可能性が残っている候補命令グループをそれぞれ順に検査する．図6.1の右側に該当し，仮に，分岐命令である可能性が残っている場合は，命令が分岐命令であると仮定して同定を試みる．途中，仮定が誤りだったと判明すれば，分岐命令である可能性を削除し他の可能性を探る．この詳細は6.3章に分けて記す．

6.2.3.3　条件フィールドの同定

本節では条件フィールドの同定方法について記す．

解析したい命令に，表6.1に記されたどの条件が使用されているかを同定する．命令が実行された事実が得られた場合，実行前のフラグの状態で満たされる条件であることが判明する．実行前のフラグを変更して実行する動作を繰り返すことで，満たされる条件が1つになるまで絞る．変化の観測を行いやすくするために，条件フィールドの同定開始前に演算に使用されるレジスタを全て乱数で書き換えておく．演算ソースが乱数であれば，特殊な計算式でない限り何かしらの変化が発生するはずだからである．当然だが，PCを書き換えてしまうと実行される命令が変わってしまうので，PCだけは書き換えることができない．以降のアルゴリズムで使用する乱数書き換え操作の際も同様にPCを除く．

本研究で使用した条件フィールドの同定方法をツリーに見立てた図を，図6.2に記す．スペースの都合上，条件の表記を数字で行っており，表6.1の2進数OPコードを10進数に変換したものに対応する．ボックス内に記述されている条件において，ボックス下部のNZCVフラグの状態で命令を実行した場合に，実行される条件を実線，スキップされる条件を点線で左右に分けて示している．このツリーでは，通常，最も出現率の高いAlways条件を最も早く検出できるように構成を行ったものである．それ以外の条件については，出現率などの考慮は行わず全体的な枝の長さが平均的になるように構成にしている．残り候補が1つになっても，最後に実行されることを確認して同定を終える構成にしている．

条件フィールド同定で候補を絞りこむ際に，実行変化が得られず候補がゼロになるケースが存在する．図6.2中の右端に位置する点線ボックスに辿り着くことである．これはNOP命令を処理したのか，もしくは変化を観測できなかった命令で

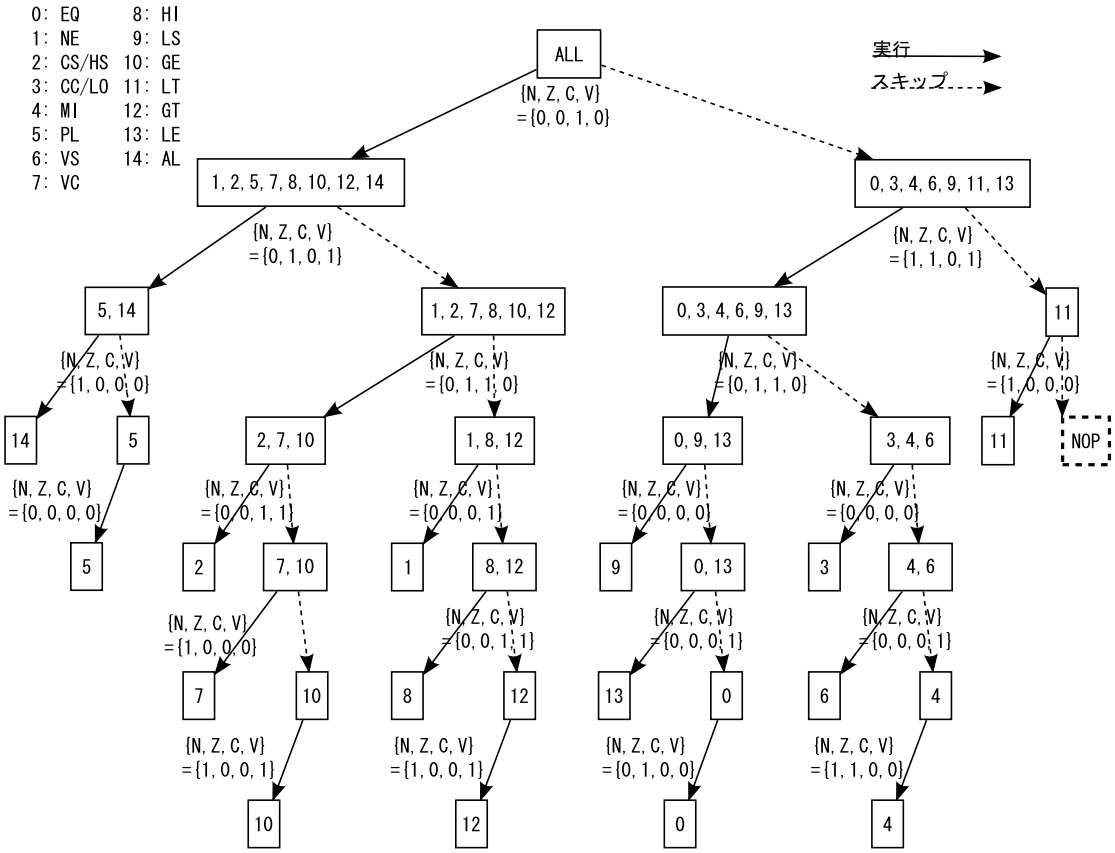

図 6.2: 条件フィールド同定の論理木

あったのかである．変化の観測が行えなかった命令に対処すべく，レジスタを乱数で書き換えて再度図 6.2 の頂上からやり直す．一定回数繰り返しても観測できなければ NOP 命令であると同定して解析を終了する．この一定回数を本研究では 10 回としたが，この値が妥当であるかの調査は行っておらず今後の課題である．

また，変化の観測ができにくい命令とは，4 ビットの条件コードフラグにのみ変化を起こすデータ処理の比較命令である．その他の命令については，レジスタの乱数書き換えにより，変化を観測できない状況になることは稀である．図 6.2 の論理木を吟味し，同じ論理木を繰り返し走査するだけでなく複数用意して使い分けることを行えば，比較命令に対処できると考えられるが，この実装も今後の課題となる．

6.2.3.4 命令種類別の可能性削除

命令がデータリソースに発生させる変化は様々なパターンが存在するが，当然それはアーキテクチャで決定されているものであって，実行中に変動したりするものではない．解析中に発生した変化の履歴を基に，その変化を起こし得ない命

表 6.2: 命令の特徴

命令種類	メモリアクセス	メモリ変化数	フラグ変化	レジスタ変化数	変化可能レジスタ	レジスタ値の範囲
分岐	なし	–	なし	1〜2	r14,r15	全領域
CLZ	なし	–	なし	1	r0〜r14	0〜32
乗算 32 ビット	なし	–	あり	1	r0〜r14	全領域
乗算 64 ビット	なし	–	あり	2	r0〜r14	全領域
データ処理演算	なし	–	あり	1	全て	全領域
データ処理比較	なし	–	あり	0	–	–
ロードマルチ	あり	–	なし	1〜16	全て	全領域
ストアマルチ	あり	1〜16	なし	0〜1	全て	全領域
ロードシングル	あり	–	なし	1〜2	全て	全領域
ストアシングル	あり	1	なし	0〜1	全て	全領域

令を以降の解析から除外することで効率良く解析が行えるようになる．本研究で対象とした命令が発生させる変化を表 6.2 にまとめた．項目の説明を以下に記す．各命令種類の動作については 6.3 章で記すこととする．

- **メモリアクセス**
  命令が主記憶に対して操作を行うかどうか
- **メモリ変化数**
  命令が主記憶上で起こす変化をワード単位で数えた値 (＝ストアされるレジスタ数)
- **フラグ変化**
  命令がフラグを変化する可能性が存在するかどうか
- **レジスタ変化数**
  命令が変化させることができるレジスタ本数の値 (全 16 本中)
- **変化可能レジスタ**
  命令が変化可能なレジスタの番号 (r0〜r15 中)
- **レジスタ値の範囲**
  変化可能なレジスタが取り得る値の範囲

表 6.2 を参考に，メモリアクセスがあればロードストア系の命令であることが判明し，フラグが変化すればフラグ更新可能な命令であることが判明する．レジス

タが 1 本でも変化すればデータ比較命令の可能性は削除できる．このような論理で有り得ない命令の可能性を削除する．ただし，レジスタが 1 本も変化しないからといって比較命令以外を削除することはなく,「変化しない」という情報は使用しない．いくつかの例を以下に示す.

仮に，メモリアクセスがなくレジスタが 2 本変化した場合，レジスタを 2 本変化できる命令は分岐と 64 ビット乗算になる．また，もしその 2 本のうち一つが r15(PC) であった場合，乗算命令の可能性が否定され分岐命令の可能性しか残らない．分岐命令であれば，もう 1 つのレジスタは r14 のはずである.

仮に，メモリアクセスがありレジスタが 3 本以上変化した場合には，ロードマルチ命令の可能性のみ残る.

仮に，メモリアクセスがなくレジスタ 1 本が変化し，その値が 0〜32 以外ならば CLZ 命令ではなくデータ比較命令でもなく，分岐・乗算 (両方)・データ演算の可能性が残る．もし変化したレジスタが r14 もしくは r15 でなければ分岐命令でもない.

このような解析を行い，残った命令の可能性を順に探り解析を行うことで同定を試みる．探査する順番は図 6.1 で示した順番で行う．命令種類ごとの解析方法は次の 6.3 章に記す.

6.3 命令種類別の解析

本章では，解析したい命令がどの命令種類であるかを，順に検査するための方法を記す.

6.3.1 分岐命令

同定を行いたい命令が分岐命令である可能性が残っている場合に，以下の解析手順により同定を試みる.

図 6.3 に分岐命令のバイナリエンコーディングを記す．Cond フィールドは条件コードを示しており，表 6.1 の OP コードに対応する．条件を設定することで条件分岐を実現する．他の命令のバイナリエンコーディングの Cond フィールドについても同様である．分岐命令には，通常の分岐 (B 命令) とリンク付き分岐 (BL 命令) の 2 種類が存在し，L ビットがセットされている場合にリンク付き分岐となる．リンク付き分岐の場合には，分岐の次の命令が格納されているアドレス値が r14 へ保存される．リンク付き分岐は通常，サブルーチンコールに使用され，r14 を r15(PC) にコピーすることでコール元に復帰する．PC にコピーを実現する命令は分岐の 1 形式であるが，実体はデータ処理演算命令もしくはデータ転送命令であり，その解析等はここでは行わない.

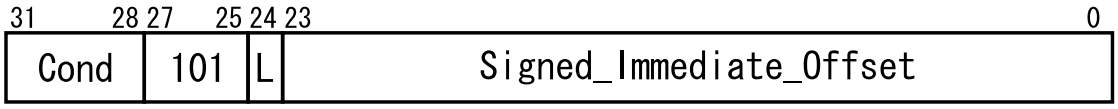

図 6.3: ARM 分岐命令のバイナリエンコーディング

分岐先アドレスの生成は次のように行われる．24 ビットで記された符号付きオフセットを符号拡張して，さらに 2 ビット分論理左シフトしてワードオフセットを生成する．そのワードオフセットと分岐命令が格納されているアドレスと 8 を加算した値が分岐先アドレスとなる．つまりワードオフセット+8 を相対値とした PC 相対分岐と考えられる．

分岐命令の可能性検証及び同定を試みる．条件分岐命令であった場合でも，事前の条件フィールドの同定により命令が必ず実行されるようにフラグを操作しているため，必ず分岐するジャンプ命令と考えてよい．分岐命令の可能性が残っている状況とは，命令の実行前後で PC が大きく異なっている状況である．大きく異なるとは通常カウントアップ値 (+4) でないことであり，次の命令のアドレスを指していない場合である．分岐先のアドレス生成に r0〜r14 の汎用レジスタは関係しないため，これらのレジスタを変更しても実行後の PC 値は変わることがない．よって，汎用レジスタを数回乱数で書き換えて実行しても，実行後の PC 値が左右されなければ分岐命令であるとみなす．この回数を本研究では繰り返し回数を 2 回と設定し，計 3 個の実行後 PC を取得する．最初の状態の実行後 PC と，乱数書き換え実行後 PC と，もう一度書き換えた実行後 PC が一致すれば分岐命令とみなしている．もし異なればそれはレジスタ値に左右される分岐を発生させる命令であり，ここで解析している分岐命令ではない．この 2 回という回数が妥当な数値であるかの調査は行っていない．ただ，レジスタ値に左右される分岐相当な命令の場合，乱数で書き換えられていれば分岐先が変わることが大半であるので，2 回という少ない回数でも充分だと考えている．

6.3.2 カウント・リーディング・ゼロ命令

同定を行いたい命令がカウント・リーディング・ゼロ (CLZ) 命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

図 6.4 に CLZ 命令のバイナリエンコーディングを記す．CLZ 命令は Rm 単体をソースとし，結果を Rd に出力する命令である．出力される結果は，Rm のバイナリ値を上位から下位の方向に見て，0 ビットが連続する数である．例えば，Rm が 0x10000000 ならば，結果は 3 となり，Rm が 0x80000000 ならば 0 となる．Rm がゼロであった場合は，32 を出力することが定義付けられている．つまり，出力される結果は 0〜32 の範囲内で収まることとなる．この命令は主に値の正規化に使用され，出力値分の論理左シフトを行うことで正規化を実現する．この命令を使

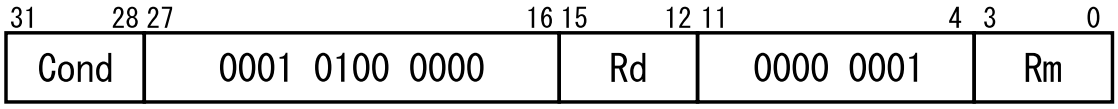

図 6.4: ARM CLZ 命令のバイナリエンコーディング

用するにあたりレジスタの使用制限が存在し，Rd と Rm ともに r15(PC) を指定してはならないという点である．

CLZ 命令の可能性が残る状況とは，変化したレジスタの本数が一つ以下で，それが r15 以外で尚且つその値が 0〜32 に収まる場合である．逆に考えれば，実行前の r0〜r14 のすべてのレジスタが 32 以上であれば，CLZ 命令である場合は変化が必ず発生するはずである．条件コードフラグを更新することもないので，フラグが変化しないことも条件となる．

CLZ 命令の可能性検証及び同定を行う．全てのソースレジスタ (r0〜r14) が 32 以上であれば必ず変化が発生することを利用する．そして全てレジスタで結果が異なるような値を代入しておくことで，Rd と Rm の同定が一度に行える．例えば初期値を 0x3F(63) とし，それぞれのレジスタ番号分の論理左シフトした値をそのレジスタの値とする．r13 ならば 0x3F を 13 ビット左シフトした 0x01F80000 とする．変化したレジスタとその値から Rd と Rm を一度に同定できる．期待する結果が得られなければ CLZ 命令ではないと断定でき，他の命令の可能性を探る．この場合は想定範囲外の値が結果となった場合である．

この時点で命令フィールドの全てを同定できたが，データ処理演算命令の一部で範囲内の結果を出力する命令が存在してしまう．この誤同定を防ぐために，初期値を変更して同じ操作を繰り返す．本研究では繰り返し回数を 1 回として計 2 回行っている．この繰り返し回数が妥当であるかの調査は行っておらず，調査は今後の課題である．

6.3.3 乗算命令

同定を行いたい命令が乗算命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

ARM 命令セットの乗算には，結果を 32 ビットで出力する短乗算と 64 ビットで出力する長乗算が全部で 6 種存在する．32 ビット短乗算には，通常の被乗数と乗数を掛け結果の下位 32 ビットを出力する積算 MUL 命令と，乗算結果にレジスタ値も加算する積和算 MLA 命令がある．64 ビット長乗算には，通常の積算命令と乗算結果と書き込み先レジスタ値を加算する積和算命令が存在し，それぞれ符号付きと符号無しの演算が行える．結果を 32 ビットで出力する場合は，符号付き符号無しのどちらで計算しても結果は同一となるので，符号による命令の区分は存在

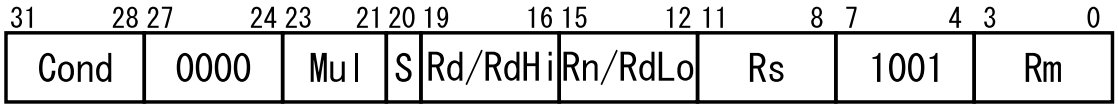

図 6.5: ARM 乗算命令のバイナリエンコーディング

表 6.3: ARM の乗算命令

Mul[23:21]	ニモニック	意味	式
000	MUL	32 ビット 積算	Rd = (Rm * Rs) [31:0]
001	MLA	32 ビット 積和算	Rd = (Rm * Rs + Rn) [31:0]
100	UMULL	符号無し 64 ビット 積算	RdHi:RdLo = (Rm * Rs)
101	UMUAL	符号無し 64 ビット 積和算	RdHi:RdLo += (Rm * Rs)
110	SMULL	符号付き 64 ビット 積算	RdHi:RdLo = (Rm * Rs)
111	SMLAL	符号付き 64 ビット 積和算	RdHi:RdLo += (Rm * Rs)

しない．乗算命令のバイナリエンコーディングを図 6.5 に，各命令の機能を表 6.3 に記した．

乗算命令は結果に応じて条件コードフラグを更新することが可能であり，図 6.5 において，S ビット (ビット 20) がセットされている場合に更新する．表 6.3 において，[31:0] の表記は乗算結果である 64 ビットの下位 32 ビットのみを選択することを指す．RdHi:RdLo の表記は RdHi レジスタ 32 ビットを上位側に，RdLo レジスタ 32 ビットを下位側として連結させた 64 ビットと考える．「+=」は C 言語プログラミングなどで見られる複合代入演算子と同等な意味を持ち，「累積」を表している．

乗算命令を使用する際にはオペランドレジスタの使用制限が存在する．

1. どのレジスタにも r15(PC) を指定してはならない．
2. Rd, RdHi, RdLo は Rm と異なる必要がある．また，RdHi と RdLo も異なる必要がある．

以上の 2 点である．ここで注意すべきは，被乗数にあたる Rm には制限が存在するが，乗数にあたる Rs には制限が存在していないことである．以上の点を踏まえて乗算の可能性検証及び同定を行う．

乗算命令の同定手順の最初として，演算ソースに使用される可能性のある全てのレジスタ値を，-2(0xFFFFFFFE) に変更して命令の実行を行う．この場合 r0〜r14 の計 15 本にあたる．演算ソースがすべて-2 だった場合，表 6.3 の 6 種の命令全てで出力される結果が異なる．その結果を表 6.4 に記す．変化したレジスタの本数と番号と値が判明するので，それを基にどの命令が実行されたのかを突き止める．表

表 6.4: 乗算ソース値が全て-2(0xFFFFFFFE) の時の乗算結果

ニモニック	結果
MUL	Rd = 0x00000004
MLA	Rd = 0x00000002
UMULL	RdHi:RdLo = 0xFFFFFFFC_00000004
UMUAL	RdHi:RdLo = 0xFFFFFFFB_00000002
SMULL	RdHi:RdLo = 0x00000000_00000004
SMLAL	RdHi:RdLo = 0xFFFFFFFF_00000002

6.4 の値と一致しない結果が得られた場合は，その命令は乗算命令ではないと断定でき，乗算の可能性を削除して残りの可能性を探る．

この時点で，乗算の種類と Rd もしくは RdHi:RdLo のレジスタ番号が判明している．

次に，乗算に使用される演算ソースの同定を試みる．乗算結果から被乗数と乗数を求めるには，結果を因数分解することにより求めることが可能である．しかし，2 値への因数分解では複数の組合せが存在することがあり，一意な同定を行うことができず，その調査をしていてはより多くの手間が必要となる．そこで演算ソースのすべてにそれぞれ異なる素数を代入して命令実行することで，素数の特性により一意な因数分解を実現し同定に至るまでの手間を削減する．積算命令にはこの方法で問題ないが，積和算命令には加算が含まれており純粋な乗算結果が得られない問題がある．この問題には，事前に判明している RdHi:RdLo の値をゼロとしておくことで，加算を除去可能である．だが乗数の Rs が RdHi と RdLo のどちらかである可能性も残っており，乗算結果がゼロであればそうであると判明する．MLA 命令の加算を除去することはできないが，事前に命令の種類が判明しているので MLA 命令だった場合には，別の手順により解析を行う．

素数の選出には，結果が 32 ビットでおさまるように 16 ビットで表現できるものを使用する．数の小さい素数を使用するとデータ処理演算命令を乗算命令であると誤同定してしまうことがあるため，選択可能な範囲で大きめの素数を採用する．結果を 32 ビットにおさめたために，RdHi:RdLo のゼロ初期化は下位 32 ビットの RdLo だけで済む．よって結果がゼロになった場合には Rs と RdLo が一致することが判明する．

命令実行により，Rd または RdLo の値を得て因数分解を行う．ここでは演算に使用されている素数の数が限られているために，結果と各素数との除算の余り値を見ることで簡潔に行える．因数分解により得られた値を格納していたレジスタが，Rm と Rs に使用されたレジスタであると同定できる．もし，結果がゼロであれば，RdLo と Rs が同じレジスタに指定されていることになる．この場合は，RdLo(=Rs) に 1 を代入して再度実行する．乗算結果は 32 ビットでおさまるので下位 32 ビット

に注目し，結果は $Rm \times Rs + RdLo = Rm \times 1 + 1$ となる．Rm は素数であるために奇数でもあり，この計算は偶数にしか成り得ない．結果が偶数であることを確認して結果から 1 を差分する．この作業により結果は Rm 単体の値になるので Rm に使用されたレジスタを同定できる．以上，MLA 命令以外の乗算の同定が完了した．

MLA 命令の解析方法は，ソースレジスタ値と出力結果 (Rd) の関係から Rm と Rs と Rn の全ての組合せを考える全数探査を行う．組合せ数は $15^3 = 3,375$ 程度である．すべての演算ソースの組合せにおいて結果が異なる値を使用することで，全数探査も一回の命令実行で同定を行うことができる．本研究では，他の乗算の同定に使用した素数でこの条件を満たしていたため，それを使用した．

よって，MLA 命令の乗算も同定が完了し，乗算全ての同定が行えたことになり，解析を終了する．条件コードフラグを更新している履歴があれば，S ビットがセットされているとして同定を完了する．

6.3.4 データ処理命令

まずデータ処理命令の説明を行う．図 6.6 にデータ処理命令のバイナリエンコーディングを記す．データ処理命令には全 16 種が存在し，内 12 種が計算結果を出力する演算命令で，のこり 4 種が計算結果を出力しない比較命令である．命令の種類を表 6.5 に記す．表中の式の「C」は命令実行前状態の条件コードフラグのキャリーを示しており，「PSR to」とは演算結果をレジスタに反映させずに，プログラム・ステータス・レジスタ (PSR) 内の条件コードフラグを更新することを意味する．

ARM データ処理命令は 3 アドレス形式を採用し，2 つのソースオペランドを指定できる．第 1 ソースオペランドはレジスタ Rn に固定され，第 2 ソースオペランド (表 6.5 中の OP2) は，ビット 11〜ビット 0 のフィールドで表現され 3 種類の方法が使用可能となっている．その表現方法は，定数，レジスタ値の定数値シフト，レジスタ値のレジスタ値シフトである．それぞれ，図 6.6 の (a) が定数，(b) がレジスタ値の定数値シフト，(c) がレジスタ値のレジスタ値シフトのエンコーディング方式となる．(b) と (c) を使用することでシフト処理と演算を 1 命令で行うことが可能となっている．シフトの種類は図中の「SFT」フィールドで表現され，論理左 (LSL)，論理右 (LSR)，算術右 (ASR)，右ローテート (ROR) の基本 4 種類が使用できる．(b) のエンコーディング方式でシフト数がゼロの場合に限り更に 1 種類，拡張右ローテート (RRX) が使用できる．右ローテートとは，右シフト処理の際に最下位ビットから溢れるビットを，最上位の空きビットに埋める方法である．例えば，0x00000001 を 1 ビット右ローテートすると，0x80000000 となる．拡張右ローテートとは，基本的な処理は右ローテートと変わらないが，命令実行前のキャリーフラグをビット 33 として，33 ビット長の右ローテートを 1 ビット分行うシフト方法である．

図 6.6 の各表現についての説明を行う．(a) の方式では OP2 は定数となり，8 ビットの「Immediate」を「ROT_Imm」の値の 2 倍分右ローテートして得られた値となる．

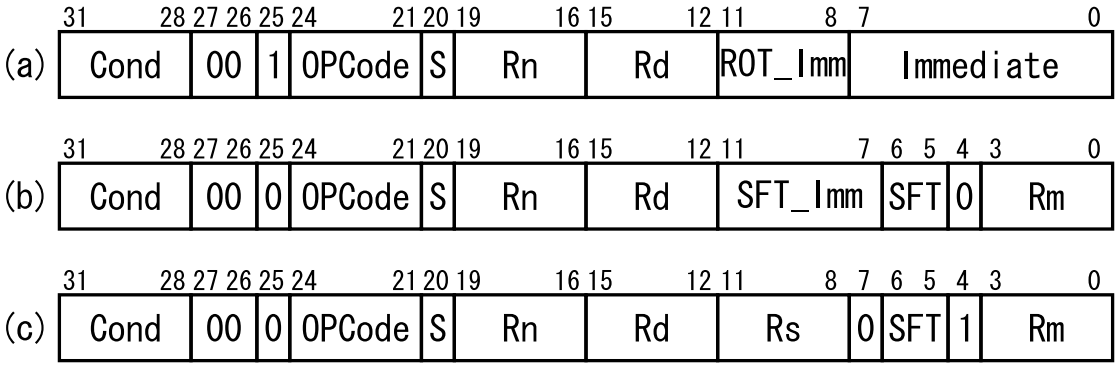

図 6.6: ARM データ処理命令のバイナリエンコーディング

例えば，値が順に 0x01，1 とすると，0x01 を 2 ビット右ローテートした 0x40000000 となる．

(b) の方式では OP2 はレジスタ値の定数値シフトとなり,「SFT_Imm」を定数シフト値として Rm レジスタのシフトを行った値となる．シフトの種類は「SFT」で指定され，定数値がゼロでシフト種類が ROR である場合に RRX が行われる．

(c) の方式では OP2 はレジスタ値のレジスタ値シフトとなり，Rs レジスタ値をシフト値として Rm レジスタのシフトを行った値となる．こちらは通常の 4 種類のシフトしか行わない．

データ処理命令では，レジスタオペランドに r15(PC) を指定することが可能である．(a)，(b) の方式の場合は，制限はなくどのオペランドに使用しても良い．ソースレジスタとして指定された場合は，命令の格納アドレス+8 の値として使用される．ただし (c) の方式の場合では，すべてのレジスタオペランドに PC を使用することができなくなっている．

どのエンコーディング方式でも S ビット (ビット 20) がセットされていると，演算結果に応じて条件コードフラグを更新する．比較命令である場合は必ず S ビットがセットされている．

以上のように多くの演算パターンを含むデータ処理命令を，演算系と比較系に分類して同定を試みる．

6.3.4.1 データ処理命令演算系

同定を行いたい命令がデータ処理演算命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

本研究では，データ処理演算を実現する演算パターンが多く，この種類の命令の効率的な同定方法を導出することができなかったために，全ての可能性について検証を行う全数探査の方法を選択し同定を試みる．PC が演算に使用される可能性もあり，静的な同定アルゴリズムでは動的な PC の値に対応できないことも，全

表 6.5: ARM のデータ処理命令

OPCode[24:21]	ニモニック	意味	式
0000	AND	論理積	Rd = Rn AND OP2
0001	EOR	排他的論理和	Rd = Rn EOR OP2
0010	SUB	減算	Rd = Rn - OP2
0011	RSB	逆減算	Rd = OP2 - Rn
0100	ADD	加算	Rd = Rn + OP2
0101	ADC	キャリー付き加算	Rd = Rn + OP2 + C
0110	SBC	キャリー付き減算	Rd = Rn - OP2 + C - 1
0111	RSC	キャリー付き減算	Rd = OP2 - Rn + C - 1
1000	TST	比較論理積	PSR to Rn AND OP2
1001	TEQ	比較排他的論理和	PSR to Rn EOR OP2
1010	CMP	比較減算	PSR to Rn - OP2
1011	CMN	比較加算	PSR to Rn + OP2
1100	ORR	論理和	Rd = Rn OR OP2
1101	MOV	コピー	Rd = OP2
1110	BIC	ビットクリア	Rd = Rn AND NOT(OP2)
1111	MVN	ビット反転コピー	Rd = NOT(OP2)

数探査を選択した要因である．全ての演算系命令，エンコーディング方式，オペランドレジスタなどから考えられる全ての組合せについてその予測 Rd 値を算出し，解析により得られる実際の Rd 結果と比較を行うことで，組合せの候補を絞っていく．その際，条件コードフラグも比較材料として用いるべきである．解析の効率向上のため，同等な結果を出力する命令の組合せについて，事前にその組合せを全数探査の候補から除外しておく．例えば，加算 $Rd = OP1 + OP2$ の場合，$(OP1, OP2) = (r0, r1)$ と $(OP1, OP2) = (r1, r0)$ は同じ Rd 結果を発生させ，数学的に同一であることが周知であるため，この 2 つを判別することは不可能である．よって片方を候補から除外する．同等な結果を出力する命令は多く存在し，事前に除去するという静的な方法では同等だとみなせない命令の組合せも存在する．この問題については動的に残りの候補を解析し同等であるのかどうかを判断することが好ましいが，本研究では一定回数の解析を行っても同定に至らなければ，残りの組合せ候補は同等だとみなして，残り候補から 1 つ選出して同定結果としている．この一定回数を 100 回と設定したが，この繰り返し回数が妥当であるかの調査は行っておらず，調査は今後の課題である．

6.3.4.2　データ処理命令比較系

同定を行いたい命令がデータ処理比較命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

比較命令では，命令実行により得られる情報が条件コードフラグの 4 ビット分しかなく，実行パターンも豊富に存在するため同定が困難な部分となる．そこで，演算系と同じく全数探査により，全ての組合せの検証を行うことで同定を試みる．全ての比較系命令，エンコーディング方式，オペランドレジスタなどから考えられる全ての組合せについてその予測フラグを算出し，コンテキストの解析により得られる実際のフラグと比較を行うことで，組合せの候補を絞っていく．

ただし，本研究内容では，この方法で同定を完了できるかどうかの検証が完結していないために，この命令種に対するアルゴリズムは未完成である．

4 ビットのフラグの内，N フラグ，C フラグ，V フラグはセット／クリアそれぞれの場合に，多くの候補が存在する．例えば，N ビットがクリアされていれば，内部の演算結果が正になる組合せの演算ということであり，32 ビットの結果で「正」の範囲は広い．だが，Z フラグについては，結果がゼロである場合に限りセットされるので，ゼロを結果とする演算であるという重要な情報が得られる．よって，Z ビットを効率良く利用することで，早期に同定を行うことができるのではないかと考えている．

6.3.5　データ転送命令マルチ系

まず，データ転送命令のマルチ系について説明を行う．ARM 命令では，1 命令で一度に複数のレジスタに対し転送を実行することが可能である．図 6.7 にデータ転送命令マルチ系のバイナリエンコーディングを記す．図 6.7 の L ビット (ビット 20) がセットされているとロードマルチ (LDM 命令)，クリアされているとストアマルチ (STM 命令) になる．ビット 15〜ビット 0 で示されたフィールドは転送制御を示すレジスタリストである．ビット 0 がセットされていると r0 を転送，ビット 1 がセットされていると r1 を転送といった法則で r15 まで続く．Rn は，主記憶にアクセスする際のアドレスを示すベースレジスタである．ベースレジスタは，各ワード転送の度に U ビット (ビット 23) に応じて増減する．U ビットがセットされている場合にインクリメント，クリアされている場合にデクリメントされる．また，P ビット (ビット 24) に応じて，増減をいつ行うかを決定する．P ビットがセットされている場合に転送前，クリアされている場合に転送後に行われる．P ビットと U ビットでアドレシングモードを表現しており，詳細については後に説明を行う．また，W ビット (ビット 21) がセットされていると，ベースレジスタを増減された値で更新する．S ビット (ビット 22) についてはユーザモードで使用されないため，説明は省略する．

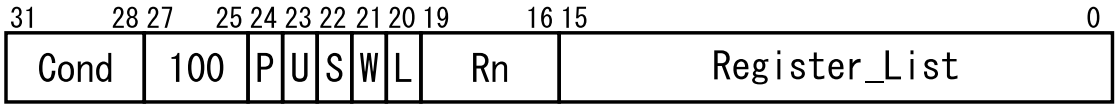

図 6.7: ARM データ転送命令マルチ系のバイナリエンコーディング

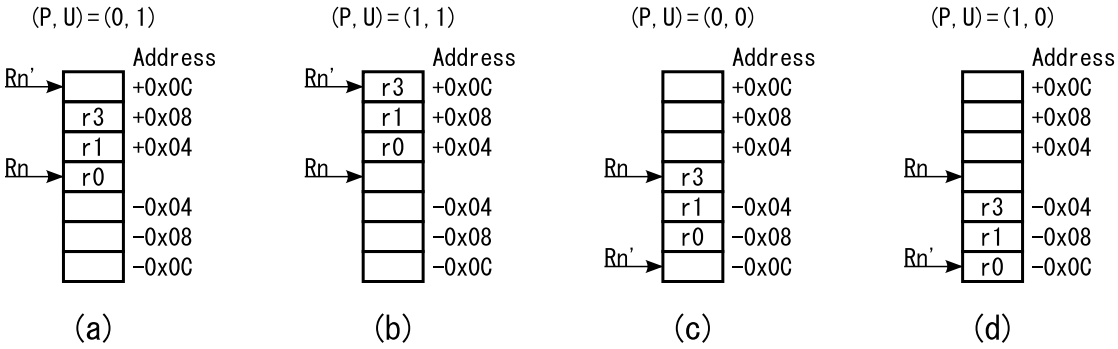

図 6.8: データ転送命令マルチ系におけるレジスタとアドレスの対応

P ビットと U ビットの状況によるアドレシングモードの説明を行う．r0, r1, r3 の転送を行うとした場合に，P と U の値によるアドレシングの違いを図 6.8 に記した．図 6.8 の Rn が指しているボックスが初期のベースアドレスを示しており，上方を正方向としてアドレス値を記した．図中の上部に記されている P と U がそのアドレシング時の状態である．各レジスタにはそれが記されているボックスのアドレスに対し，ロードストア動作が行われる．また，Rn' は W ビットがセットされていた場合の更新後の Rn 値を示している．

マルチ転送を行う際に，レジスタの指定制限が存在する．ベースレジスタの更新を行う場合に，同じレジスタをレジスタリストの中で指定してはならないことである．更新を行わない場合は問題なく指定できる．また，ベースレジスタに r15 を指定してはならない点も存在する．

6.3.5.1 ロードマルチ命令

同定を行いたい命令がロードマルチ命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

6.2.1 節で記した主記憶の状態を前提に同定を試みる．主記憶にはアドレス+オフセット (0x80) の値が格納されている前提である．マルチ系転送命令ではベースレジスタはそのまま使用される．後述の 6.3.6 節に記すシングル系転送のようなオフセット増減は行われない．このことを利用して同定を行う．転送されるレジスタの最大数は 16 であり，ベースレジスタで示されるアドレスの上下 0x40 付近のアドレスにしかアクセスできない．ベースレジスタとして指定され得る r0〜r14 をそれ

ぞれ，一定以上の間隔を空けた値に変更して命令を実行する．その間隔は 0x40 ＋前提オフセット以上とする．例えば，r0 は 0x00100000，r1 は 0x00101000 といった値にする．もしベースレジスタが r2(0x00102000) だった場合，アドレシングが行える範囲は 0x00101FC0〜0x00102040 となり，格納されている値は前提によりアドレスにオフセットを加えた 0x00102040〜0x001020C0 となる．つまりロードされ得る値もこの範囲となる．ロードされた値から前提オフセットを除去すれば，アクセスしたアドレスが得られる．よってロード値に直近な値を格納するレジスタがベースレジスタであると同定できる．実行前のレジスタにロードされ得る値と以前の値とで一致することはないため，レジスタリストの同定も全て行える．また，ベースレジスタの更新が行われていた場合，最大転送数が 15 になるので，アドレシングの範囲は両端から 4 ずつ狭まり 0x00101FC4〜0x0010203C となって，値は 0x00102044〜0x001020BC となる．ベースレジスタの更新値はアドレシングの範囲内であり，ロードされ得る値の範囲とは異なるため，同様に同定が行える．

また，図 6.8 に示される (a), (b), (c), (d) のように，レジスタ番号の若いレジスタがアドレスの下位になるようにアドレシングされるために，ロードされた値の上限値と下限値を調べることで，アドレシングモードの同定も行える．例えば，図 6.8 において最大値がロードされる r3 が，ベースレジスタに対し，上方に位置していれば U ビットがセット，下方に位置していればクリアとなる．そして，最小値がロードされる r0 がベースレジスタと一致すれば P ビットがセット，一致しなければクリアとなる．

以上のことにより，ロードマルチ命令の同定が完了する．

6.3.5.2 ストアマルチ命令

同定を行いたい命令がストアマルチ命令である可能性が残っている場合に，以下の解析手順により同定を試みる．

同定手順は 6.3.5.1 節に記したロードマルチ命令とほぼ同じである．レジスタを一定間隔の値で書き換えて実行し，書き換えられた主記憶のアドレスと値を取得する．もしレジスタが 1 つだけ変更されていた場合は，変更されたレジスタがベースレジスタであり，尚且つ更新が行われたことになるため W ビットはセットされていると同定できる．変更されていなければ更新なしとなり W ビットはクリアとなる．また，更新なしでも書き換えられた領域からベースアドレスを同定でき，書き換えられた値からレジスタリストも同定できる．そして，その領域の両端のアドレスを得ることで，アドレシングモードの同定も可能となる．

以上のことにより，ストアマルチ命令の同定が完了する．

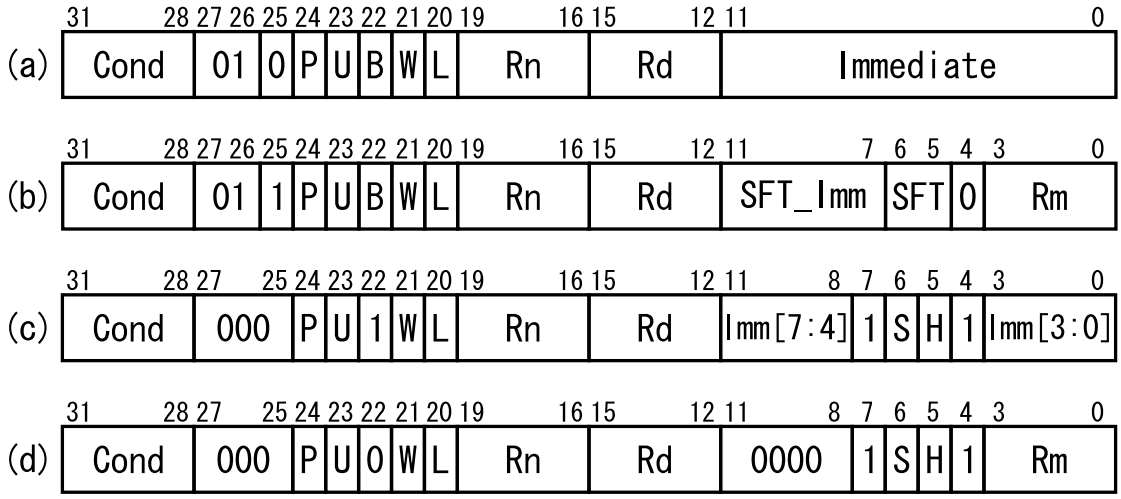

図 6.9: ARM データ転送命令シングル系のバイナリエンコーディング

6.3.6 データ転送命令シングル系

まず，データ転送命令のシングル系について説明を行う．図 6.9 にデータ転送命令シングル系のバイナリエンコーディングを記す．エンコーディングの種類は 4 種類あり，図 6.9 の (a) と (b) が符号なし 1 バイトもしくは 1 ワードの転送に使用され，B ビット (ビット 22) がセットされていると符号なしバイト転送を行い，クリアされているとワード転送を行う．(c) と (d) が，符号付き 1 バイトもしくは 1 ハーフワードの転送に使用され，H ビット (ビット 5) がセットされているとハーフワード転送を行い，クリアされているとバイト転送を行う．また，S ビットがセットされていると符号付き転送となり，クリアされていると符号なし転送となる．ただし，H クリアかつ S クリアの組合せは符号なしバイト転送を意味するが，この転送は (a) と (b) のエンコーディングで実現されるため使用されない．アドレシングモードは，(a) と (c) の場合にイミディエートオフセット，(b) と (d) の場合にレジスタオフセットが使用できる．さらに (b) の場合はレジスタ値を定数値シフトすることも可能である．

図 6.9 の各エンコーディングに共通する部分の説明を行う．L ビット (ビット 20) がセットされているとロード動作になり，クリアされているとストア動作になる．Rd はソース (ストア元) またはデスティネーション (ロード先) のレジスタを指し，Rn は主記憶にアクセスする際のアドレスを示すベースレジスタである．ベースレジスタは U ビット (ビット 23) に応じてオフセット値と加減算を行う．U ビットがセットされている場合に加算，クリアされている場合に減算される．また，P ビット (ビット 24) に応じて，増減をいつ行うかを決定する．P ビットがセットされている場合に転送前，クリアされている場合に転送後に行われる．P ビットがクリアされている場合に，W ビット (ビット 21) がセットされていると，ベースレジス

タを増減された値で更新する．P ビットがセットされている場合は W ビットにかかわらずベースレジスタを更新する．通常，P ビットがセットされている場合では W ビットはクリアされており，両方がセットされている場合は非ユーザモードでのみ使用される命令形式となる．

アドレシングモードの説明を行う．図 6.9 の (a) では，ビット 0〜ビット 11 で示される 12 ビットを符号なしイミディエートとしてオフセットに使用する．同様に (c) ではビット 0〜ビット 3 を下位 4 ビット，ビット 8〜ビット 11 を上位 4 ビットとして連結した 8 ビットを符号なしイミディエートとしてオフセットに使用する．(b) では，Rm のレジスタ値を「SFT」フィールドで示されるシフトタイプでシフトを行い，シフト値はビット 7〜ビット 11 で示される 5 ビットが使用される．シフトタイプは，6.3.4 節で説明したデータ処理命令のエンコーディングの図 6.6(b) の場合と同様である．

また，転送命令を使用するにあたりいくつか注意点があり次に示す．

1. r15(PC) にロードすることは分岐の 1 形式であるが，それはワードサイズのロードに限られる．バイトサイズやハーフワードサイズのデータを PC にロードしてはならない．

2. 通常，Rd と Rn と Rm は互いに異なるレジスタを指定する．ただし，ベースレジスタ Rn の更新を行わない場合に限り，Rd と Rn は同一のレジスタを指定しても構わない．

3. PC のストアはプロセッサの実装方法で異なる結果を生むため，使用するべきではない．

同定を行いたい命令がデータ転送命令シングル系である可能性が残っている場合に，以下の解析手順により同定を試みる．

データ処理命令と同様で，考えられるオフセットのパターンが多く，レジスタオペランドに PC が使用されるパターンもあるため，全数探査による同定を行う．全てのアドレシングパターンの可能性を考え，ロード命令と考えられる場合，ロード値とアクセスアドレスが指し示す値が一致するパターンを探す．

ただし，本研究内容では，この方法で同定を完了できるかどうかの検証が完結していないために，この命令種に対するアルゴリズムは未完成である．

全数探査の効率を向上させる要因として，命令実行前にレジスタを乱数で書き換えておくことが挙げられる．しかし，アドレシングパターンや乱数の組合せによっては，主記憶の領域外アクセスを発生させることが有り得る．領域外アクセスを起こした場合は通常，例外を発生させるため，例外情報を有効に使用できれば効率が向上すると考えられる．

6.4 評価結果

本章では，6.2 章，6.3 章で記したアルゴリズムで，命令列が同定可能であるかを検証し，可能であればそれが現実的な時間内で完了する処理であるのか評価を行う．

6.4.1 評価方法

同定アルゴリズムの検証及び評価を行う．本研究室で作成された，Intel 社の XScale マイクロアーキテクチャを採用した Intel PXA255 プロセッサ [31] のシミュレータ (asim)[32] を，セキュアプロセッサとみなして解析に必要なデータ (コンテキストなど) を取得し，そのデータに対して同定アルゴリズムをプログラム化したもので検証を行う．アルゴリズムの検証を主目的としており，ARM プロセッサ上で稼働するクラックプログラムではなく，単なるアルゴリズムの検証システムである．検証システムは C++言語で作成している．検証システムにシミュレータを内蔵し，必要な情報をシミュレータから得て解析を行う形となる．シミュレータから得る情報は次に示す情報となる．

- 1 命令実行前後のコンテキスト (レジスタ，フラグ)
- 1 命令実行前後の主記憶データ (ただし変化分のみ)
- 命令処理における主記憶アクセスの有無

実行前後の主記憶データの比較を事前に済ましておくとして，ここでは変化分のみを取得することとした．

現段階ではアルゴリズムの検証システムは一部未完成であり，データ処理命令比較系とデータ転送シングル系の命令は対応できていない．解析の対象となった命令がこれらの種類の命令であった場合の評価は行えていない．また，データ処理命令の演算系において，Rd のみを全数探査の材料に使用しフラグ情報を使用していない．このため，フラグを更新する一部の命令で同定に失敗するケースが存在している．

評価を行うための基準を定める．6.2.3.1 節で記した全体の流れの 5〜14 を 1 ループの基準として，1 命令の同定に必要とするループ数を算出する．1 ループに要する解析労力と，ある 1 命令の同定に要するループ数との積値が，1 命令の同定に要する総合労力とする．

1 ループで行う処理を簡潔に記すと次の通りとなる．

1. 対象プログラムの退避コンテキスト及び主記憶の使用領域の値を「実行前情報」として記録する．

表 6.6: ARM プロセッサモデル

動作周波数	200MHz
データキャッシュサイズ	32KB
キャシュラインサイズ	256 ビット (32B)
キャシュミスレイテンシ	25 サイクル

2. 退避コンテキストを復元し，対象プログラムに処理を移して実行する．
3. 外部割込みにより，解析プログラムに制御を戻してコンテキストの退避を行う．
4. データキャッシュの無効化を行う．
5. 対象プログラムの退避コンテキスト及び主記憶の使用領域の値を「実行後情報」として記録する．
6. 実行前後の主記憶値を比較し変化した部分を検出する．
7. 実行前後の情報を基に解析を行う．
8. 退避コンテキストを同定アルゴリズムに応じて初期化し，主記憶が変更されていれば前提条件に従い初期化する．
9. 1 へ戻る．

主記憶に関する処理において，キャッシュを無効化してアドレスバスをプローブできていれば，アクセスしたアドレスがキャッシュライン単位で判明するため，主記憶の使用領域すべてを得る必要はなくなる．ストアマルチ命令による主記憶への書き戻しが最大 64 バイトで，表 6.6 よりキャッシュラインサイズが 32 バイトであるので，書き戻し領域は最大で 3 つのラインにまたがることが考えられる．したがって最大 3 ライン分の 96 バイトを取得し，保存及び比較をすることとなる．

クラックシステムのプログラムは ARM 上で動作することになるが，本システムでは PCLinux 上にシミュレータを用いて作成しているため，7 の部分を PC での実行時間で代用する．その他の部分については ARM プロセッサ上で行うとした場合に実行される命令の処理時間を概算し求める．使用する ARM プロセッサモデルは表 6.6 のものとし，必要な情報のみをここでは掲載する．各命令の実行レイテンシは Intel PXA255 プロセッサのものを使用した．

1 と 5 の部分の見積りを行う．それぞれコンテキスト内の汎用レジスタ 16 本と CPSR の計 17 ワードと，3 キャッシュライン分の 24 ワードを記録する動作である．1 ワードの記録処理にロード命令とストア命令をそれぞれ 1 回要するとし，ロード

命令による読み込みデータを直後で使用する場合のレイテンシが 3 サイクルであるので，以降，ロード命令のレイテンシは 3 サイクルとする．また，ストア命令のレイテンシは 1 サイクルである．合計で $(3+1)\times 41 = 164cycle$ となる．

2 の部分の見積りを行う．退避コンテキストの復元に 16 レジスタ転送のロードマルチ命令を使用すると考える．16 レジスタ転送のロードマルチ命令のレイテンシは $23cycle$ であるので，合計で $23cycle$ となる．

3 の部分の見積りを行う．コンテキスト退避に 15 レジスタ転送のストアマルチ命令を使用すると考え，割り込みからのリターンアドレスをスタックに書き込むためにストア命令も使用すると考える．15 レジスタ転送のストアマルチ命令のレイテンシは $17cycle$ であるので，合計で $(17+1) = 18cycle$ となる．

4 の部分の見積りを行う．この部分に要する時間が他に比べて突出している．これはキャッシュを全て入れ換えるためにキャッシュライン数と同じ回数のロードを実行し，各ロードがキャッシュミスを起こすことで実現されるからである．キャッシュライン数はキャッシュサイズからラインサイズを割った値であるので，表 6.6 より 1024 本となる．各ロード命令に 3 サイクル，表 6.6 よりキャッシュミスレイテンシは 25 サイクルであるので，合計は $(3+25)\times 1024 = 28,672cycle$ となる．

6 の部分の見積りを行う．ここで比較するサイズは 3 キャッシュライン分 96 バイトの 24 ワードである．比較命令を 24 回行い，最大で 16 ワードの不一致が起こる．これはストアマルチ命令でストアされるワード数が最大 16 ワードだからである．1 ワードの比較にロード命令 2 回と比較命令 1 回，記録にストア命令 1 回とする．比較命令は $1cycle$ で行われるので，合計は $(3+3+1)\times 24 + 1\times 16 = 184cycle$ となる．

8 の部分の見積りを行う．レジスタ 15 本に加え CPSR，及び最大 16 ワード分の主記憶の初期化を行うことになる．初期化の動作のみに注目すると，ストア命令を 32 回実行することになるので，合計は $1\times 32 = 32cycle$

以上を全て合計すると $29,275cycle$ となる．200MHz の周波数だと 1 サイクルに $0.5\mu s$ 要するので，$29,275cycle \times 0.5\mu s = 14,638\mu s \fallingdotseq 1.46\times 10^1 ms$ が ARM プロセッサ上での処理時間となる．

7 の部分について，検証システムを動作させた PCLinux のマシン環境は表 6.7 であった．1 ループ分の解析で行う計算量は，解析対象となった命令の種類が何であるかで大きく変動する．命令の種類ごとにある程度の数のサンプルを選出し，1 ループに要する平均時間を表 6.8 にまとめた.「解析部分」の値の意味は，6.2.3.1 節の 12，もしくは本節の 7 にあたる部分に要する 1 ループあたりの平均時間である．データ処理演算系が突出して多い原因は，演算系の解析に全数探査を行うことだと考えられる．対応できていない命令種類は記していない．

計測方法は C 言語標準ライブラリの clock() 関数を用いて行い，計測精度は $10ms$ であった．データ処理演算命令以外については，$10ms$ の計測精度に対し，母数が数千万や 1 億といった数値で平均を算出しているので，精度は $0.1ns$〜$1ns$ となるのでこの精度でも問題ないと考えている．また，データ処理演算命令の母数は少

表 6.7: 検証システムを動作させたマシン環境

OS	Fedora Core 5
CPU	Intel® Pentium® 4 2.80GHz
メモリ	1GB
HT 機能	オン

表 6.8: 命令種類別の解析部分 1 ループの時間

命令種類	解析部分 [μs]
分岐	9.70×10^{-1}
CLZ	8.61×10^{-1}
乗算	4.19
データ処理演算	5.70×10^{3}
データ転送マルチ	1.17

なく数万という数値での平均であるため，0.1μs の精度を示すことになるが，データ処理演算命令の 1 ループ時間の有効単位が 10μs となっているため，この精度でも問題ないと考えている．

また，シングルユーザモードとマルチユーザモードにおいて計測を行ってみたが，計測結果に影響を与えるような違いは見られなかった．Pentium4 の Hyper-Threading 機能をオンにした環境で計測しており，このアルゴリズム検証システム自体がシングルスレッドプログラムであるため，CPU の性能をすべて使用することはできてはいない．そのため，マルチユーザモードにおいても，その他のプロセス処理を空いた時間で処理していたと思われ，結果に差異は出なかったと考えられる．

よって先ほど概算した ARM プロセッサ上での処理時間と，表 6.8 の「解析部分」の各時間との合算値が，各命令を解析する上での単位ループ処理時間となる．この値と同定までのループ数との積値が，1 命令を同定するために要する時間となる．これは 6.4.2 節で記すこととする．

6.4.2 命令単体結果

同定アルゴリズムの検証システムを，それぞれの命令種類における単体の命令に適用した結果を本節に記す．命令種類別の結果を表 6.9 に記す．

表 6.9 において，平均ループ数とは，解析の対象となった命令が命令種類のそれであった場合に，同定に至るまでの平均回数を示している．単位ループ所要時間とは，その命令の同定を行う際にかかる時間を全体の流れ 1 ループ単位で示したものである．前説で述べた ARM プロセッサ上における処理時間と，表 6.8 の「解析部分」の加算値を示している．データ処理演算命令以外の加算において情報落ちが発生しているため，このような結果となった．1 命令同定所要時間とは，平均ループ数と単位ループ所要時間の積値であり，その命令を同定する際にかかる時間を意味する．現段階では，データ処理命令比較系とデータ転送命令シングル系に対応できていないため，これらの命令についての結果は得られていない．また，

表 6.9: 命令種類別の同定所要時間

命令種類	平均 ループ数	単位ループ 所要時間 [ms]	1 命令同定 所要時間 [ms]
分岐	7.67	1.46×10^1	1.12×10^2
CLZ	6.67	1.46×10^1	9.74×10^1
乗算	6.73	1.46×10^1	9.83×10^1
データ処理演算	9.98	2.03×10^1	2.03×10^2
データ転送マルチ	5.67	1.46×10^1	8.28×10^1

データ処理命令演算系において条件コードフラグを変更するタイプで一部，正常な同定が行えていない問題点もある．6.3.4.1 節で述べているが，フラグ情報を全数探査に利用していないためである．

各命令種類において，ある程度の数の命令サンプルを選出しそれらの同定を試みて，合計ループを命令数で割ることで 1 命令あたりの平均ループ値を算出した．この平均値はサンプルの合計ループ回数を母数で除算するという単純な平均であり，各命令を同定することに要するループ回数の値分布がどのようになっているかは調査していない．

表 6.9 の結果より，最も多く時間を要するデータ処理命令の演算系の同定を試みれば，1 命令あたり平均で $2.03 \times 10^2 ms$ という時間で同定することができる結果となった．1000 命令を 3 分 23 秒で同定できる結果となり，これは充分に現実的な時間である．ただし，本評価を行う上で対応できていない命令のうち，データ処理命令の比較系に関して，平均ループ数が他の命令種に比べ，1 桁から 2 桁あるいは 3 桁増えるのではないかと現段階では予想している．比較命令に対する同定アルゴリズムの性能向上で平均値を減少することができるとしても，他の命令種よりも非常に大きな値となることが予想され，同様に 1 命令同定に要する時間も大きな値となることが予想される．したがって現段階での同定アルゴリズムでは，比較命令の同定に突出した時間を要することになると予想している．また，同じく対応できていないデータ転送命令のシングル系に対しては，命令の性質上，比較命令よりは少ないループ数になると考えている．

同定アルゴリズムの成功率に関して結果を述べる．全ての命令種において，1 千万分の 1 かそれ以下の確率で条件フィールドの同定に失敗する結果が出ている．この原因は条件フィールドの同定に乱数を用いており，その乱数値によって失敗するケースが存在するためである．全ての命令に共通する現象であるため，以降で記す結果ではこの要因による失敗を省いた結果とする．

解析の対象となった命令が，分岐命令，CLZ 命令であった場合に関しては，全てのパターンにおいて一致する機械語命令を導出できていた．

乗算命令に関しても同様で，全てのパターンにおいて一致もしくは同等な機械

語命令を導出できていた.「同等な」とは文字どおりの意味で，例えば乗算の場合，乗数と被乗数が入れ替わっても同等な式であるが，機械語レベルでみると異なるのでこのような表現を使用している．

データ転送命令マルチ系では，前提条件の主記憶書き換えにより，全てのパターンにおいて一致する機械語命令を導出できていた．ただし，マルチ転送命令で 1 つのレジスタしか転送を行わないパターンは，データ転送命令のシングル系と同等であるため，シングル系とみなしてこの結果には含まれていない．

データ処理命令演算系に関しては，条件コードフラグの更新を行わない命令であれば，全てのパターンにおいて一致もしくは同等な機械語命令を導出できていた．フラグを更新する命令パターンでは失敗するケースが存在している．

以上，分岐，CLZ，乗算，データ転送マルチの命令に関しては全て同定可能なことが判明した．データ処理演算に関しては，現段階ではフラグ更新しないパターンについてのみ同定可能である．失敗及び未対応な命令に対応したアルゴリズムの作成は今後の課題である．

6.4.3 実プログラム結果

同定アルゴリズムの検証システムを，実際のプログラムで使われている命令列に適用した結果を本節に記す.

適用する実際のプログラムには，Stanford ベンチマークの各アプリケーションのうち Intmm の適当な箇所の同定を試みた結果の一部を掲載する．その結果を表 6.10 に記す.

表 6.10 に記した項目の説明を行う.「同定結果の命令」とは,「PC」のアドレスに格納されている「実際の命令」に対し，同定アルゴリズムの検証システムを適用した結果を示す.「同定までのループ回数」で同定に至るまでに要したループ数を示し,「機械語一致」で，実際の命令と同定結果の命令を機械語レベルで比較した場合の真偽を示す.

同定アルゴリズムの検証システムが対応できていない命令については，結果を記していない．CMP 命令は比較命令であり，ループ回数が括弧付きの (5) 回となっているのは，図 6.1 における，比較命令であると仮定する段階に到達した回数を示している．同様に，STR 命令はストアシングル命令であり，転送シングル命令と仮定する段階に到達した回数を示している.

登場している命令の種類について簡単に説明を行う．STMDB と LDMIA は 6.3.5 節で示したデータ転送命令のマルチ系であり，DB と IA の部分でアドレシングモードを表している．MOV，ADD，SUB，RSB，CMP は 6.3.4 節で示したデータ処理命令である．BL と BLE は 6.3.1 節で示した分岐命令で，BL はリンク付き分岐，BLE の LE 部分は 6.1 で示した条件付き実行を表している．アセンブリ言語で表現する場合，条件の識別子を命令識別子の最後に付与することで条件付き実行の表現を行い，AL(Always) 条件の場合は通常表記しない.

表 6.10 に記した結果について説明を行う．両端に位置しているストアマルチ命令とロードマルチ命令の動作から，掲載した部分はサブルーチンの 1 つだと思われる．ループ回数に記されている値のうち，100 回以上の多い回数を要している命令がある．これは，データ処理命令を同定するための全数探査において，命令パターンの候補を 1 つに絞りきれなかった状況を表している．現段階でのアルゴリズムでは 100 回全数探査しても絞りきれなければ，残りの候補は同等な命令だと判断し，候補の 1 つを選出して結果としている．絞りきれずに残った候補がどのような命令であるか例を示す．

PC:0x0001e698 に格納された実際の命令の MOV r8, #0x01 は，r8 に定数の 1 を代入する命令である．同定中に残った候補命令は，MOV r8, #0x01 と MOV r8, r15 LSR #0x10 と MOV r8, r15 ASR #0x10 であった．2 番目と 3 番目の命令は，r15(PC) を右シフトすることで定数の 1 を生成し代入を行う命令であるため，識別が行えなかったのである．この命令がこのアドレスにある限り，これら 3 つの命令は同等な命令であるため，結果から識別することは不可能である．通常，PC をこのように使用しないとして全数探査から除外したとすれば，ループ数は 11 回で済んでいたということがわかっている．現段階での 100 回という設定が大きすぎることが原因と考えられ，適する制限回数の見極めは今後の課題である．または，特殊な命令パターンとして初めから除外しておくことも 1 つの手段である．100 回を超えている他の命令も，同様の事由によるものであった．

機械語レベルでの一致結果をみると 1 つを除き全て一致していた．前述のデータ処理の全数探査で打ち切った命令も一致しているのは偶然の結果である．ただ 1 つ異なる命令があるが，結果を見てみるとレジスタ同士の加算命令でレジスタの指定が逆になっているだけである．つまりこれらは数学的に同等である演算であるため，結果としては正しく同定できているとしても問題ない．

6.4.2 節の表 6.9 の単位ループ時間と表 6.10 のループ回数から，対応していない命令を除く他 19 命令を同定することに要する時間を算出すると，合計で $1.29\times10^{1}sec$ となった．19 命令にこの時間を要すると考えると，1000 命令では 11 分 22 秒という結果となる．6.4.2 節の結果と大きく異なっているが，ループ回数が 100 回を超えている命令が存在することで，平均ループ数が増大したためである．19 命令で合計 648 ループとなり，平均は 34.1 回であった．だが，1000 命令に 11 分 22 秒を要するとしても，まだ現実的に可能な時間の範囲内であると考える．

表 6.10: Intmm における適用結果

PC	実際の命令	同定結果の命令	同定までのループ回数	機械語一致
0x0001e690	STMDB r13!, {r4-r8,r14}	STMDB r13!, {r4-r8,r14}	4	Success
0x0001e694	MOV r7, #0x00	MOV r7, #0x00	25	Success
0x0001e698	MOV r8, #0x01	MOV r8, #0x01	105	Success
0x0001e69c	MOV r6, #0x29	MOV r6, #0x29	6	Success
0x0001e6a0	MOV r5, #0x01	MOV r5, #0x01	105	Success
0x0001e6a4	BL 0x0001E150	BL 0x0001E150	6	Success
0x0001e6a8	MOV r1, #0x78	MOV r1, #0x78	5	Success
0x0001e6ac	MOV r4, #0x00	MOV r4, #0x00	25	Success
0x0001e6b0	BL 0x0001FCE4	BL 0x0001FCE4	6	Success
0x0001e6b4	RSB r0, r0, r0 LSL #0x04	RSB r0, r0, r0 LSL #0x04	6	Success
0x0001e6b8	ADD r3, r6, r5	ADD r3, r5, r6	10	Fail
0x0001e6bc	SUB r4, r4, r0 LSL #0x03	SUB r4, r4, r0 LSL #0x03	6	Success
0x0001e6c0	ADD r5, r5, #0x01	ADD r5, r5, #0x01	104	Success
0x0001e6c4	SUB r4, r4, #0x3C	SUB r4, r4, #0x3C	104	Success
0x0001e6c8	CMP r5, #0x28	—	(5)	—
0x0001e6cc	STR r4, [r7, +r3, LSL #0x02]!	—	(3)	—
0x0001e6d0	BLE 0x0001e6a4	BLE 0x0001e6a4	8	Success
0x0001e6d4	ADD r8, r8, #0x01	ADD r8, r8, #0x01	104	Success
0x0001e6d8	CMP r8, #0x28	—	(5)	—
0x0001e6dc	ADD r6, r6, #0x29	ADD r6, r6, #0x29	7	Success
0x0001e6e0	BLE 0x0001e6a0	BLE 0x0001e6a0	8	Success
0x0001e6e4	LDMIA r13!, {r4-r8,r15}	LDMIA r13!, {r4-r8,r15}	4	Success

第7章　おわりに

本報告書では，暗号化されたプログラムと，それを復号するプログラム鍵との対応付けに仮想記憶機構を利用したプログラム保護システムを提案した．提案システム上に存在する実行プログラムは，ページ単位でプログラム鍵と対応付けられているため，1つのプロセスの中に，異なるプログラム鍵で暗号化されたプログラムが存在することができる．これにより，マルチタスク環境下のOSプログラムやシェアードライブラリのライブラリプログラムのような複数のプログラムで共有されるようなプログラムでも暗号化することができる．

また，提案システムで用いる共通鍵暗号としてTriple-DESとAES-128を取り上げ，それぞれの復号回路を設計し評価を行った結果，AES-128復号回路の方が，同程度の面積で処理能力が高い回路が設計できた．そして，提案システムのフィージビリティ・スタディとして設計したAES-128復号回路やRSA復号回路がプロセッサの性能やチップ面積に与える影響を調査した結果，性能低下及び面積の増加はプロセッサとして許容できる範囲内であることが分かった．

しかし，本提案システムのように，オペランドデータの暗号化を行っていないセキュアプロセッサに対し，プログラムの保護能力の評価として，データ解析による同定アルゴリズムの考案・検証を行ったところ，現段階での同定アルゴリズムでは対応できていない部分が存在するが，環境さえ整えれば充分に現実的な時間内でほぼ正確な同定が完了することが判明した．この結果はセキュアプロセッサのプログラム保護能力が不充分であることを示している．また厳密に同定できていなくても一定以上の知識をもった技術者が解析結果を参考にすることで，アルゴリズムの推定は充分に可能と言える．

したがって，本提案方式の更なる改良を行う必要性があることが確認できた．

今後の課題としては，以下の点が挙げられる．

- **入出力データの解析によるプログラム解読への対応**

  提案システムではデータの暗号化は行っておらず，主記憶上のデータは平文のままで処理される．これまでは，データの入出力の解析によって，プログラムを解読することは，暗号解読に匹敵する時間が必要になり，難しいと考えていたが，OSを改ざんすることができるような環境において1命令ごとに割り込みを発生させ，退避されるコンテキストを改ざんする方法で，現実的な時間内で機械語命令を1命令ずつ同定することが可能であるということが判ってきている．1つの対応策として，コンテキストを含むデータの暗号

化機能をサポートすることを検討している．その場合，データを書き戻す必要があるため，データキャッシュと主記憶間にAES暗号回路を追加することになるが，AES復号回路と多くの部分で共有化するように設計することで，面積の増加は抑えられる．つまり，ハードウェア量はほぼ変わらないと考えられる．したがって，今回の提案システムにおける面積の評価は有効なものであるといえる．ただし，暗号化の処理による遅延は，性能に影響を与えるため，今後，AES暗号回路を追加することによる，性能への影響を検討していきたい．

現段階において，解析されにくいセキュアプロセッサの案としては次のようなものが考えられる．

- ソフトウェアによる耐タンパー化を行う
- 前提条件とした項目を成し得ない
- オペランドデータを暗号化する

ソフトウェアによる耐タンパー性向上 [25] としてプログラムの難読化を行い，プログラム命令列が同定できても，その時点からのリバースエンジニアリングを行いにくくする方法も考えられる．ソフトウェアによるコード暗号化手法もあるが，コードをデータとして扱う時点で復号時に暗号化されていないコードが外部に漏れるため，これは意味をなさない．

前提条件の中で攻撃者に特に有利な条件は,「処理の逐一停止」である．2命令ごとでしか停止できない場合，それだけで考えられる命令の組合せが以前に比べて2乗に激増する．つまり最低でも2命令とか実行のたびに異なる命令数を処理しないと停止しないようなプロセッサを設計することで，解析の難度が大幅に上昇と考えられる．

また，データが暗号化されていない事由で解析されるため，データを暗号化すればデータによる解析はそもそも行えなくなるが，XOMが行っているようにOSなどの特権状態においても機能制約をする必要がある．

この検討は，本研究の後半に手がけた項目であり，予想外の結果を得たため，十分な検討はなされていない．この前提条件の吟味も含めて，もっと広い視点からの攻撃方法を考え，それに対する対策を検討する必要がある．

- **コンパイラの改良**

  5.10節で述べたように，現在のコンパイラ環境では，出力するオブジェクトファイルはデータが.dataセクションに完全に分離されていない．今後，コンパイラを改良することで対応する必要があると考えている．

- **FPGA実装による動作確認**

Stanford Benchmark のプログラムが正しく動作することを論理シミュレーションにより確認できれば，次の段階として FPGA への実装と動作確認を行う予定である．

- **RSA 復号回路に対する処理時間解析攻撃への対策**

  RSA 復号回路の処理時間は，処理するデータに依存して変化する．この処理時間を解析することで秘密鍵 d を得ようとする攻撃法が存在する．図 4.14 で示したバイナリ法のアルゴリズムでは $d_i = 0$ の場合は $P = P{\cdot}CmodM$ が行われない．また，図 4.20 で示したモンゴメリ法のアルゴリズムでは，条件分岐があり，STEP3 で法 M を加算する場合としない場合がある．暗号文 C は第三者が自由に選ぶことができるため，あるイタレーション中に加算が行われる暗号文と，行われない暗号文を用意しておき，それぞれの暗号文に対する復号処理時間を求める．その差から d_i を推定される可能性がある．対策として，モンゴメリ法での演算処理が一定になるようにすることが考えられる．これは，事前に処理時間を定めておき，1 つの処理ごとにカウントを取り，処理が終了しても，定めた処理時間までは終了するのを待つという方法で実現できる．今後，このような処理時間解析攻撃への対策も検討していく予定である．

関連図書

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, Nov. 2000.

[2] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pp. 166–178, May 2003.

[3] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[4] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *In 17th ACM International Conference on Supercomputing*, June 2003.

[5] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[6] 橋本幹生, 春木洋美. 敵対的な OS からソフトウェアを保護するプロセッサアーキテクチャ. 情報処理学会論文誌, Vol. 45, No. SIG 3 (ACS 5), pp. 1–10, March 2004.

[7] 城本正尋, 田端猛一, 酒井智也, 島田貴史, 窪田昌史, 川端英之, 北村俊明. 公開鍵暗号を用いてプログラムの保護を行うプロセッサの提案. 情報処理学会論文誌, Vol. 47, No. SIG 18 (ACS 16), pp. 55–64, November 2006.

[8] National Institute of Standard and Tecnology. DATA ENCRYPTION STANDARD (DES). FIPS PUB 46-3, October 1999.

[9] National Institute of Standard and Tecnology. Announcing the ADVANCED ENCRYPTION STANDARD (AES). FIPS PUB 197, November 2001.

[10] 森岡澄夫, 佐藤証. 共通鍵暗号 AES の低消費電力論理回路構成法. 情報処理学会論文誌, Vol. 44, No. 5, pp. 1321–1328, May 2003.

[11] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communication of the ACM*, Vol. 21, No. 2, pp. 120–126, February 1978.

[12] 揖元. 初等整数論入門. 培風館, 2000.

[13] 櫻井隆雄. 親子剰余算器を用いた左向きアレイ法の実装方式. 東京大学大学院情報理工学系研究科修士論文, 2004.

[14] P.L.Montgomery. Modular multiplication without trial division. *Math, Computation*, Vol. 44, pp. 519–521, 1985.

[15] 高木直史. 算術演算の VLSI アルゴリズム. コロナ社, 2005.

[16] 榎本忠義. CMOS 集積回路. 培風館, 1996.

[17] J.Hong and C.Wu. Radix-4 modular multiplication and exponentiation algorithms for the rsa public-key cryptosystem. *DAC 2000*, pp. 565–570, 2000.

[18] 神永正博, 渡邊高志. 情報セキュリティの理論と技術. 森北出版, 2005.

[19] J. R. Levine. *Linkers & Loaders*. オーム社, 2001.

[20] TIS Committee. *Tool Interface Standard(TIS) Executable and Linking Format(ELF) Specification Version1.2*, May 1995.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[22] 門田暁人, 高田義広, 鳥居宏次. ループを含むプログラムを難読化する方法の提案. 電子情報通信学会論文誌, Vol. J80-D-I, No. 7, pp. 1–11, July 1997.

[23] Incorporated Trusted Computing Group. TCG Specification Architecture Overview, April 2004.

[24] T Kawabata, T Tamai, M Hashimoto, and T Miyamori. Security enhanced embedded processor using local memory protection mechanism. *Cool Chips IX*, pp. 143–157, 2006.

[25] 石間宏之, 亀井光久, 齊藤和雄. ソフトウェアの耐タンパー化技術. *IPSJ Magazine*, Vol. 44, No. 6, pp. 622–627, June 2003.

[26] ARM アーキテクチャリファレンスマニュアル. ARM Limited, 2005.

[27] ARM Limited. ARM アーキテクチャリファレンスマニュアル, 第 ARM DDI 0100DJ-00 版, Feb. 2000. All rights reserved.

[28] Steve Furber, アーム株式会社監訳. 改訂 ARM プロセッサ 日本語版. CQ 出版株式会社, 第 2 版, June 2002. ISBN4-7898-3357-7.

[29] ARM メモ. http://www.bomber.co.jp/chaola/docs/ARM/.

[30] ARM Ltd. ARM922T. http://www.arm.com/products/CPUs/ARM922T.html.

[31] Intel®. *Intel® XScale™ Microarchitecture User's Manual*, Mar. 2003.

[32] 山下淑子. ARM アーキテクチャを対象とした性能評価ソフトウェアシミュレータの開発. 広島市立大学情報科学部情報工学科卒業論文, 2005.

[33] 田端猛一. 組み込みプロセッサを対象としたマイクロアーキテクチャの評価. 広島市立大学情報科学部情報工学科卒業論文, 2006.