社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－DPS－138 (42)
2009－CSEC－ 44 (42)
2009／3／6

セキュアソフトウェア開発環境 DFITS (Data Flow Isolation Technology for Security)

林 良太郎 † 橋本 幹生 † 春木 洋美 † 藤松 由里恵 † 中西 福友 †

† 株式会社東芝 研究開発センター

我々は，保護・非保護のメモリを使い分けるセキュアプロセッサ上で，不正な解析・改変に耐える安全なソフトウェアを構築するための開発環境 DFITS を提案する．DFITS は，暗号処理に注目してソースコードにおけるデータの流れを解析することにより，ソースコード中の変数を，機密情報などの保護必要変数と入出力などの保護不要変数とに自動分類し，分類結果に応じて変数を適切なメモリへ自動配置する．従来はセキュリティの専門知識をもつプログラマにしかできなかったこの作業を自動化することにより，安全なソフトウェアを効率的に作ることが可能となる．

Secure Software Development Environment DFITS (Data Flow Isolation Technology for Security)

Ryotaro Hayashi † Mikio Hashimoto † Hiroyoshi Haruki †
Yurie Fujimatsu † Fukutomo Nakanishi †

† Corporate Research and Development Center, TOSHIBA Corporation

We propose a development environment DFITS for developing secure softwares against illegal analysis and modification, where the softwares are running on security processor architecture which supports a secure and an non-secure memory areas. DFITS analyzes data flows of source code focusing on the cryptographic operations, detects the confidential data, and assigns the confidential data to the secure area, and the other data to the non-secure area automatically. By supporting such a work which only the programmers with specialist knowledge of security can do, DFITS provides easier development of secure software.

1 はじめに

1.1 ソフトウェア保護

PC や情報家電から自動車，携帯電話に至るまで，我々の身の回りの多くのものには，それを制御・活用するためのソフトウェアが搭載されている．このようなソフトウェアはユーザー端末において利用されるため，悪意のあるユーザーにより，ソフトウェア中の機密情報を読み取られたり改ざんされたりする危険にさらされる．例えば DVD 再生ソフトウェアの場合，コンテンツ復号鍵を読み取られたり，ライセンスチェック部分をスキップするようにコードが改ざんされてしまうと，DVD 再生ソフトウェアの著作権保護システムが破綻することになる．実際，PC 向け DVD 再生ソフトウェアのコンテンツ復号鍵が解析されるという事例が報告されている．また，不正な解析によりソフトウェアベンダ独自の実装技術が盗まれるといった可能性も考えられる．このような脅威に対して，ソフトウェアがもつ機密情報を悪意のあるユーザーによる読み取りや改ざんから守り，ソフトウェアが正しく実行されることを保証するためのソフトウェア保護技術が必要である．

1.2 セキュアプロセッサ

ソフトウェア保護を実現するハードウェア基盤として，セキュアプロセッサ（耐タンパプロセッサ）がある．これは，プロセッサチップ内に埋め込まれた鍵による暗号処理によりプログラムがもつ機密情報の保護を行うことができるプロセッサである．その１つである LMSP (License-controlling Multi-vendor Secure Processor) [1] は，マルチタスク環境において，プロセスごとに隔離された保護メモリ領域を提供する．同時に，どのプロセスも参照可能な非保護メモリ領域も提供する．保護メモリ上に配置されるプロ

セスの構成要素（命令，データ，コンテキスト）はLMSPのチップ内の鍵により暗号化および改ざん保護される．保護メモリ上の情報は当該プロセスのみがアクセス可能であり，そのプロセス以外からは，たとえOS特権を利用しても読み取ったり改ざんしたりすることはできない．

プロセスの構成要素のうち，命令は暗号化された状態で外部メモリに置かれ，プロセッサ内部で復号および解釈実行される．コンテキストに関しては，LMSPはハードウェアコンテキストスイッチを備え，レジスタ値を暗号化して外部メモリに退避できる．命令とコンテキストの暗号処理は，キャッシュメモリ入出力および割り込み発生時にプロセッサハードウェアによって処理される．通常のプログラムは自身の命令やコンテキスト情報を直接参照することはないため，暗号処理はプログラマからは意識されずプログラマ透過的である．

1.3 セキュアソフトウェア開発の困難性

ここでさらにプロセスが参照するデータもすべて保護メモリ領域に配置すれば，プロセスの構成要素はすべて保護メモリ内に置かれるため，メモリを通じた外部からの解析・改ざんの可能性を排除して安全なプロセス実行が実現できる．しかしながら，データをすべてを保護すると，メモリ上に配置されたプロセスに関する情報が外部から一切読み書きできなくなるため，（OSを介した）他のアプリケーションとのやりとりも一切できないこととなり，これでは基本的なソフトウェアの機能要件がみたされない．実用プログラムにおいては非保護メモリへの参照が必要不可欠である．すなわち，セキュアプロセッサを用いて実用的かつ安全なソフトウェアを構成するには，プログラム中の各データ（変数）に対してそれをどのように保護すべきかを判断し，かつその保護要件に応じて変数が保護・非保護メモリ領域に適切に配置されるようにプログラムを作りこまなければならない．

しかしこれはプログラマにとって非常に困難な作業である．まず，このような作業を行うにはセキュリティに関する専門知識が必要となるが，そもそもプログラマが皆セキュリティに明るいとは限らない．また，プログラム中には中間変数を含めれば膨大な変数が存在するため，保護・非保護の判定ミスやメモリへの配置ミス等のヒューマンエラーが入り込む恐れがある．

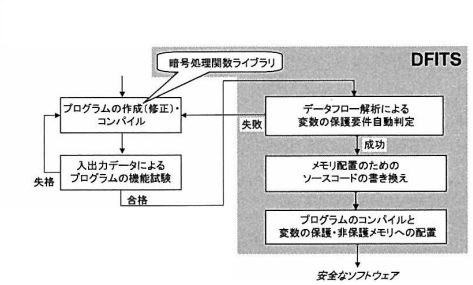

図 1: DFITSを用いたプログラム開発フロー

さらに厄介なことに，これらのエラーは攻撃がないときは顕著に現れず，かつソフトウェアの機能には影響しないため，通常の機能試験ではまったく検出されない．

1.4 セキュアソフトウェア開発環境

我々は，保護・非保護のメモリを使い分けることができるセキュアプロセッサ上で安全なソフトウェアを生成するためのソフトウェア開発環境DFITS (Data Flow Isolation Technology for Security) を提案する．DFITSは，ソースコードにおけるデータの流れ（データフロー）を暗号処理に注目して解析することにより変数を保護・非保護に自動分類し，分類結果に応じて各変数をメモリへ自動配置する．データフローに注目して変数の隔離 (Isolation) を行うことがDFITSの名前の由来である．

DFITSを用いたセキュアプログラム開発フローを図1に示す．ここでは，実装しようとするセキュリティプロトコルの仕様は形式的手法などを用いて安全性が検証済みであることを前提とする．DFITSはそのようなセキュリティプロトコルを安全に実装するための支援技術である．開発フローは，機能検証を行うフェーズと，変数の保護要件判定とメモリ配置を自動的に行い実行可能形式プログラムを生成するフェーズの2つに大きく分かれている．

機能検証フェーズでは，プログラマは実装対象のセキュリティプロトコルとデータ入出力などの関連機能をプログラムとして記述し，実行形式を作成して機能試験を行う．機能試験に失格した場合は，プログラムの修正を行い，再度試験を行う．このとき，プログラマは変数のメモリ配置を意識する必要はない．ただし，暗号化や署名などの暗号処理関数（セキュリティプリミティブ）に関しては，セキュリティ専門家によりメモリ管理も含めて安全性が保証されたライブラリを使用する．

次に機能試験に合格したプログラムのソースコードに対して，各変数の保護要件を判定し，それに応じて各変数が適切なメモリに配置された実行形式プログラムを作成する．DFITS はこれらを自動的に行うものである．なお，秘密鍵をそのまま外部に出力するようなセキュリティ的に矛盾のあるソースコードが入力された場合，適切なメモリ割り当てが不可能であるため，DFITS はエラーを出力する．この場合，プログラマは再度プログラムを修正する必要がある．

以上の手順により生成される実行可能形式プログラム（ソフトウェア）は，安全性が検証されているセキュリティプロトコルを，変数の適切なメモリ配置により安全に実装したものとなるため，結果として安全なプログラムとなるはずである．従来，後半のフェーズにはセキュリティの専門的知識とプログラミング技術が必要だったが，DFITS によりこれを自動化することが可能となる．これにより，機能要件をみたし，かつヒューマンエラーのない安全なソフトウェアを効率よく構築することができる．

以下，第 2 節で DFITS について詳しく紹介し，DFITS の試作および実装検証結果を第 3 節に示す．そして，関連研究について第 4 節で述べた後，第 5 節でまとめを行う．

2 DFITS

本節では DFITS について説明する．まず DFITS が扱うセキュリティ型である保護属性とそれを用いたメモリ自動配置について述べる．次に DFITS が提供する暗号処理関数であるセキュリティ関数について述べ，最後に保護属性自動決定について説明する．

2.1 保護属性とメモリ自動配置

DFITS では，各変数がもつデータの保護要件を，機密性および完全性の 2 つの観点から扱う．そのために，DFITS では「保護属性」と呼ばれるデータの秘匿および改ざんレベルに対応した型を定義する．定義される保護属性は exposed，fixed，hidden，verified，concealed，confidential の 6 つである．

exposed 属性の変数は非保護メモリ上に配置される．よって誰でも読み書きが可能であり，外部からの入力は exposed 変数が用いられる．

fixed, verified 属性の変数は共に改ざん検証機能つきメモリに配置される．この 2 つは（メモリ配置の意味では同じ扱いだが）データフロー解析において使い分けられる．verified はその完全性が確認された変数であるのに対し，fixed はその変数の完全性が保証されていない．例えば，署名検証をパスする前の変数は fixed，パスした後の変数は verified となる．なお，入力は exposed に限定されると述べたが，出力は外部から読むことができればよいため，exposed, fixed, verified のいずれかが用いられる．

hidden, concealed, confidential 属性の変数は，すべて暗号化および改ざん検証機能付きメモリに配置される．これらもデータフロー解析において使い分けられる．confidential は機密性と完全性が確認された変数，concealed が機密性のみ確認された変数，hidden は機密性と完全性のいずれも保証されていない変数である．

DFITS（における保護属性自動決定）では，セキュリティの観点から，変数間の演算（四則演算，代入など）は同じ保護属性をもつ変数間のみに限定している．この仕組みにより，保護要件に応じたデータフローの隔離が行われる．例えば confidential 変数がもつ機密データのコピー先も必ず confidential 変数となり，それが exposed 変数に代入され外部に漏洩するといった危険性を排除している．

保護属性を用いることで，それぞれの変数をコンパイル時に適切なメモリへ配置することができる．その実装方法については 3 節で述べる．

2.2 セキュリティ関数

DFITS では保護属性により属性の混在を防いでいるが，セキュリティプロトコルを実行するためには異なる保護属性をもつ変数間の演算が必須となる．例えば，暗号文を復号する場合は，外部入力である暗号文と，機密データである秘密鍵を同時に用いて演算を行う．そこで DFITS には，異なる属性をもつ変数間の演算を行うための手段として，暗復号，署名生成・検証，ハッシュ関数などの一方向性を持つ暗号処理関数（セキュリティプリミティブ）が，セキュリティ関数として定義されている．

セキュリティ関数の実装はセキュリティの専門家によって行われ，内部演算がすべて暗号化および改ざん保護された領域で行われるように設計される．よって，セキュリティ関数内部の変数はプログラマには意識されず，保護属性も付与する必要がない．プログラマは，属性変換

分類		入力(第1引数)		鍵値(第2引数)		出力(第3引数)	
		意味	保護属性	意味	保護属性	意味	保護属性
公開鍵暗号化	A	平文	confidential	公開鍵	verified	暗号文	verified
	B		concealed		verified		exposed
公開鍵暗号文復号	A	暗号文	verified	秘密鍵	confidential	平文	confidential
	B		exposed		confidential		concealed
共通鍵暗号化	A	平文	confidential	秘密鍵	confidential	暗号文	verified
	B		concealed		confidential		exposed
共通鍵復号	A	暗号文	verified	秘密鍵	confidential	平文	confidential
	B		exposed		confidential		concealed
署名生成	A	入力文	confidential	秘密鍵	confidential	署名	confidential
	B		verified		confidential		verified

分類		入力1(第1引数)		入力2(第2引数)		鍵値(第3引数)		出力(第4引数)	
		意味	保護属性	意味	保護属性	意味	保護属性	意味	保護属性
署名検証	A	入力	concealed	署名	concealed	公開鍵	verified	検証済	confidential
	B	メッ	exposed		exposed		verified	メッ	verified
	C	セージ	concealed		exposed		verified	セージ	confidential

分類		入力(第1引数)		出力(第2引数)	
		意味	保護属性	意味	保護属性
ハッシュ(1入力1出力)	A	入力	exposed	出力	exposed
	B		verified		verified
	C		concealed		concealed
	D		concealed		exposed
	E		confidential		confidential
	F		confidential		verified

図 2: セキュリティ関数

を伴う計算を DFITS が提供するセキュリティ関数に限定することで，安全にセキュリティプロトコルを実装することができる．

セキュリティ関数は，その処理内容に応じて入出力の保護属性が限定されている．図 2 に公開鍵／共通鍵暗号，署名，ハッシュ関数の入出力定義を示す[1]．この定義を見ると，例えば公開鍵暗号化において，暗号化のための公開鍵は機密性は無いが完全性は保証されている必要があるため，verified に限定されている．一方，平文は機密保護が必要だが，完全性は必須ではないため，完全性の有無によって 2 つのバリエーションがある．出力の暗号文は（暗号化されているため）機密保護が不要となり，完全性は入力のものをそれぞれ引き継いでいる．

署名検証では，検証済メッセージが出力されているが，これは入力メッセージと同じ値をもち，保護属性のみが異なる変数である．これは，検証前と検証後のメッセージを（保護属性の違いから）分離して扱うためである．

また，セキュリティ関数ではないが，入力関数の引数は書き込み可能な exposed，出力関数の引数は読み取り可能な exposed, fixed, verified のいずれかに限定されることに注意しておく．

2.3 保護属性自動決定アルゴリズム

ここからは，保護属性を決定するアルゴリズムについて説明する．

2.3.1 基本的な保護属性自動決定

既に述べたように，DFITS には，(1) セキュリティ関数以外での異なる保護属性間の演算禁止，(2) セキュリティ関数における入出力の保護属性限定の 2 つのルールがあり，これを使って保護属性を決定していく．図 3 の上段は，外部から入力された暗号文とそれに付与された署名を検証し，検証後に復号を行うプロトコルのデータフローを示したものである．このプロトコルの場合，まず署名検証関数の 2 つの入力が外部からのため，それぞれが exposed となる．これにより，図 2 における署名検証関数の入出力定義 B が選択され，他の変数の保護属性が決定する．すると，検証済暗号文が verified となるため，それとデータフローでつながった一時変数も（属性が変化しないため）verified となる．これにより，公開鍵暗号文復号については図 2 における入出力定義 A が選ばれ，図 3 の下段のように，全ての変数の保護属性が決定する．

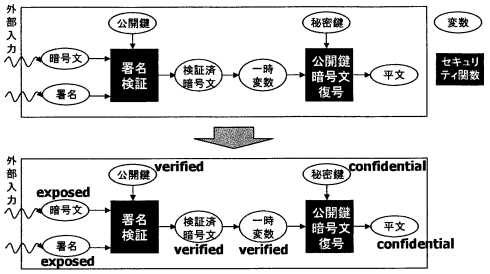

図 3: 保護属性自動決定（基本）

2.3.2 完全性の間接的検証の検出

DFITS は，さらに複雑なプロトコルに対しても保護属性を自動決定できるように，複数暗号処理に注目した完全性の間接的検証を検出する．以下では，完全性の間接的検証について具体的な例を用いて説明する．

図 4 は，ハイブリッド暗号を利用したメッセージ転送プロトコルの一部である．暗号文と署名を受信すると，受信した暗号文の復号を行い，そのハッシュ値と署名により署名検証を行う．検証が成功したら，ハッシュ関数に通す前の共通鍵を使用して平文の暗号化を行い，暗号文を外部に送信する．

このデータフローにおいて保護属性を決定しようとすると矛盾が生じる．共通鍵暗号化の鍵は入出力定義から confidential でなければならない．一方，この共通鍵は外部からの入力を（検証せずに）復号したものであるから，完全性が確認できていない．よって，共通鍵の保護属性を一意に決定することは不可能である．

[1] ここで示したものは入出力定義の代表的な例であり，セキュリティ関数として具体的に実装する暗号プリミティブの種類・強度によって，適切な入出力定義は変化する．

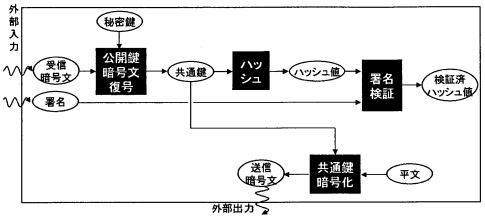

図 4: ハイブリッド暗号プロトコルのデータフロー

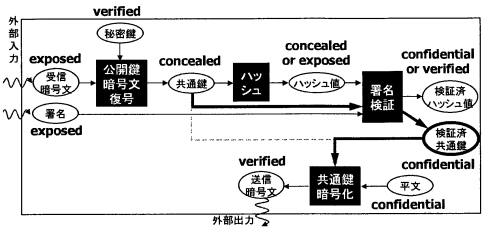

図 5: 完全性の間接的検証を考慮した拡張データフローと保護属性

一方，このデータフローをセキュリティの観点で解析すると，まず署名検証により「ハッシュ値」の完全性が確認されている．ここで，ハッシュ関数は衝突困難性（おなじ出力値をとる 2 種類の入力を見つけることが困難である）をみたすので，共通鍵が（ハッシュ値が同じになるような）別のものにすりかえられているという危険性が排除される．よって，ハッシュ値の完全性が検証されると，同時にハッシュ関数の入力である共通鍵の完全性も確認できる．すなわち，署名検証を行った後，共通鍵の完全性は（ハッシュ値の検証によって）間接的に確認されており，それを共通鍵暗号化の鍵として使ってもセキュリティ上問題はない．

以上の解析より，このデータフローは安全であるにも関わらず，共通鍵の完全性が間接的に検証されるという事実が単一のセキュリティプリミティブに閉じた基本的なデータフロー解析のみでは適切に解釈できないため，保護属性を決定できなかったということがわかる．なお，このような事例はメールの送受信プロトコルである PGP や S/MIME，暗号通信に標準的に利用される SSL，TLS 等多くのプロトコルで見られる．

このようなプロトコルに対しても保護属性を自動決定するために，DFITS は完全性の間接的検証を検出し，それにあわせてデータフローを拡張して保護属性を決定する機構を備える．

DFITS には，各関数に対して依存関係と呼ばれる関係が定義されている．これは，決定項と従属項のペアからなり，それぞれの項は当該関数の引数により構成される．依存関係は「決定項の変数の完全性が検証されると，従属項の変数の完全性が（間接的に）検証される」ことを意味している．依存関係は，具体的には「関数において，決定項の変数値が同一となるような従属項の変数値を 2 種類（以上）求めることが計算量的に困難である」と定義できる[2]．

例えば前述のハッシュ関数の場合，出力値が同じになるような入力値を複数求めるのは困難であるため，出力が決定項，入力が従属項という依存関係が定義できる．さらに，入力が定まると出力が一意に定まることから，入力が決定項，出力が従属項という依存関係も定義できる．

DFITS は，この依存関係を用いて，完全性が間接的に検証される変数を検出し，その変数を検証前変数と検証後変数に分離する．具体的には，依存関係と署名検証関数に注目し，依存関係の決定項変数が署名検証関数に入力される部分を見つけると，対応する従属項変数もその署名検証関数で検証されるようにデータフローを書き換える．

図 5 は，図 4 のデータフローを上記のルールに従って書き換えたものである．DFITS は，ハッシュ関数における依存関係の決定項であるハッシュ値が署名検証において検証されることを検出する．そして，ハッシュ関数における依存関係の従属項である共通鍵が署名検証において検証されるように，共通鍵が署名検証に入力され，かつ検証後の検証済共通鍵が署名検証から出力されるようにデータフローを書き換える．さらに，共通鍵暗号化に使われる鍵を，(コントロールフロー解析等により）検証済共通鍵に置き換える．この書き換えられたデータフローは，前述の基本的な保護属性自動決定により，保護属性を決定することができる．注目すべきは，検証前の共通鍵 (concealed) と検証済み共通鍵 (confidential) が分離されていることである．なお，ハッシュ値と検証済ハッシュ値はそれぞれとりうる保護属性の可能性が複数あるが，これはいずれの保護属性でもセキュリティ上問題がないことを意味している．

[2]決定項の変数値が定まると従属項の変数値が一意に定まる場合ももちろんこれに含まれる

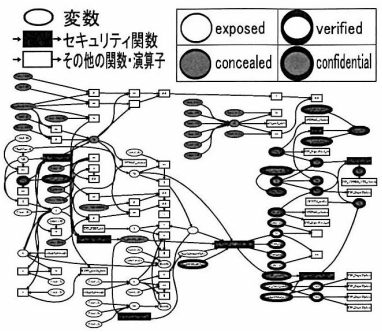

図 6: OpenSSL のデータフロー解析結果（一部）

3 試作による実装検証

これまでに説明した DFITS の仕組みについて，我々はメモリ自動配置と保護属性自動決定についてそれぞれ試作を行った．本節では試作とその適用による実装検証結果について説明する．なお図 1 で示すように，この 2 つを一連で利用するためには，保護属性自動決定におけるデータフロー書き換えに応じたソースコード書き換えが必要となるが，この部分の試作は未実施であり，今後の課題である．

まず保護属性に対応した変数格納メモリの自動配置については，保護属性毎にメモリ配置を行う C++クラステンプレートを用意し，各変数について決定された保護属性に対応するクラステンプレートを利用することで実現した．これをソースコードと共にコンパイルすることで，変数が保護属性ごとに用意されるメモリ領域に配置される．我々は，簡単な保護属性つきソースコードを作成し，それがクラスライブラリと共にコンパイルできること，並びにそれぞれの変数が保護属性ごとに隔離されたメモリ領域に配置されることを確認した．

次に保護属性の自動決定については，データフロー解析に Elsa (The Elkhound-based C++ Parser) [2] と呼ばれる C++構文解析支援ツールを利用した．適用実験では，公開されている SSL のソースコード (OpenSSL) の復号と署名検証を行う一連の処理部分に限定して DFITS を適用した．現在の DFITS は構造体，関数ポインタなどの解析能力に限界があるため，ソースコードの一部に手作業による書き換えを行ったものに対して DFITS を適用した．

DFITS 適用により得られた保護属性つきデータフローの一部を図 6 に示す．セキュリティ専門家が OpenSSL の仕様 (RFC) を手作業で解析して決定した変数の保護属性と，DFITS が出力した上記変数の保護属性が一致することを確かめ，この例における DFITS の保護属性自動決定の妥当性を確認した．なお，2.3.2 節で説明した完全性の間接的検証の検出を省いた DFITS を同ソースコードに適用したところ，保護属性は適切に決定できなかった．

4 関連研究

DFITS と同様にプログラミング言語に対する静的解析を用いてソフトウェアのセキュリティを保証する研究として，情報流解析 [3] が挙げられる．一般に情報流解析では，プログラムへの入力と出力の関係をもとに，機密情報漏洩を厳密に解析する．よって，プログラムの内部変数にアクセスすることなく入出力関係から機密情報を推測するような攻撃をよくモデル化している．

一方 DFITS では，メモリの解析によりプログラム内部の一時変数などを取り出す攻撃モデルを想定している．また，情報漏洩に関する解析は情報流解析より単純な（厳密でない）ものとなっている．これは，解析に適度な柔軟性を持たせることで，実用的なセキュリティプロトコルへの適用を可能とするためである．

5 まとめ

セキュリティプロセッサ上における安全なソフトウェアの開発環境 DFITS を提案した．今後の DFITS の改良により，セキュリティソフトウェアの安全性と生産性の向上が期待できる．今後は DFITS の能力向上および適用範囲拡大とともに，DFITS が行う変数隔離の正当性評価方法についても検討を進めていく．

参考文献

1) 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：コンピューティングシステム, vol. 45, No. SIG 3(ACS 5), 2004.

2) `http://www.cs.berkeley.edu/~smcpeak/elkhound/`

3) D. Volpano, C. Irvine, G. Smith: A sound type system for secure flow analysis. J. of Computer Security, Vol. 4, No. 2–3, pp. 167–187, 1996.