サンドボックスシステムにおける投機的な安全性検査

尾上浩一[†]　大山恵弘[†,††]　米澤明憲[†,††]

サンドボックスシステムは信頼できないアプリケーションを安全に実行する用途において有用である．現在提案されているいくつかのサンドボックスシステムは，アプリケーションが発行するシステムコールを捕捉し，安全性が保証されるようにシステムコールの実行を制御するというものである．既存のシステムの一つの問題は，システムコール捕捉後の安全性検査によって課されるオーバヘッドである．本論文では，安全性検査を投機的に実行するサンドボックスシステムを提案し，その設計と実装について述べる．提案システムは，将来行われるアプリケーションの動作を予測し，マルチプロセッサを利用して投機的な安全性検査をアプリケーションと並列に実行する．それにより安全性検査のオーバーヘッドを削減する．将来の動作は，過去の実行におけるシステムコールのプロファイルを利用して予測する．我々は提案するシステムを Linux 上に実装し，予備評価を行った．

Speculative Security Checks in Sandboxing Systems

Koichi Onoue ,[†] Yoshihiro Oyama [†,††] and Akinori Yonezawa[†,††]

Sandboxing systems are useful for secure execution of untrusted applications. Some sandboxing systems proposed so far assure security by intercepting system calls invoked by an application and controlling their execution. A problem in existing systems is the overhead of security checks performed after system call interceptions. In this paper, we propose a sandboxing system that executes speculative security checks, and describe design and implementation of the system. The proposed system predicts the future behavior of a sandboxed application and executes speculative security checks in parallel with the application, thus reducing the overhead. Future behavior is predicted based on profiles of system calls in past executions. We have implemented the system on Linux and made a preliminary evaluation.

1. はじめに

現在，各ユーザはネットワーク等を利用することにより，様々なアプリケーションを容易に取得，使用することが可能となっている．しかし，これらのアプリケーションの安全性は保証されていないことが多い．安全性の保証されていないアプリケーションを実行することにより，各ユーザの実行環境に悪影響を及ぼす可能性がある，最悪の場合，システム全体に致命的な影響を与える可能性がある．現在では，システムの権限を乗っ取ることを意図的した悪意あるコードを含んだアプリケーションもしばしばみられる．

安全性の保証されていないアプリケーションを実行することによるシステムへの被害を防ぐための一つの効果的な手法として，**サンドボックスシステム**を利用することが考えられる．サンドボックスシステムとは，指定したアプリケーションがファイルやネットワーク等の資源に対して行える操作を制限し，そのアプリケーションの実行環境を他のアプリケーションの実行環境から隔離するシステムである．サンドボックスシステムの利用により，安全性の保証されていないアプリケーションによるシステムへの被害を最小限に抑えることができる．これまでに多くのサンドボックスシステムが提案されている[1)]．本研究ではシステムコールの捕捉に基づくサンドボックスシステム（たとえば文献 2)~13) などのシステム）を対象とする．

サンドボックスシステムが持つ問題点はアプリケーションの実行時に行う安全性検査に関するオーバーヘッドである．システムコールの捕捉に基づくサンドボックスシステムでは，アプリケーションがシステムコールを発行したらアプリケーションの実行を停止させ，システムコールの安全性を検査した後に，アプリケーションの実行を再開させる．安全性検査にかかる時間はそのままオーバヘッドとしてアプリケーションの実行時間に加わる．安全性検査が単純な処理ばかりであれば，そのオーバヘッドは小さく抑えられる．し

† 東京大学
The University of Tokyo
†† 独立行政法人科学技術振興機構，戦略的創造研究推進事業
Japan Science and Technology Agency, CREST

かし，近年，安全性検査のために複雑な計算や時間のかかる処理を行うシステム[14),15)]も提案されてきている．また，資源の仮想化を通じて安全性を保証するサンドボックスシステム[3),6),8),9),11)]も提案されている．それらのシステムでは，安全性検査や仮想化処理のオーバヘッドを削減することが求められる．

本論文ではサンドボックスシステム上で実行されるアプリケーションの性能を向上させるための手法を提案する．提案するサンドボックスシステムは，アプリケーションが将来行うであろう動作を予測し，安全性検査を投機的に実行する．それにより，安全性検査のためにアプリケーションが停止する時間が短縮され，オーバヘッドが削減される．具体的には，提案手法では，マルチプロセッサを利用し，投機的な安全性検査をアプリケーションと並列に実行する．アプリケーションがシステムコールを実際に発行した際に，そのシステムコールが予測と一致しているかどうかを検査する．もし一致していれば，投機的な安全性検査の結果にもとづいて即座にアプリケーションの実行を再開させたり，アプリケーションを強制終了させたりする．我々は提案システムを設計，実装し，予備評価を行った．

本研究がもたらす一つの利点は，安全性検査のオーバヘッドの削減により，重い安全性検査処理を行うサンドボックスシステムの利用を快適にすることである．さらなる利点は，有用であるがオーバヘッドが大きすぎるため採用されていない安全性検査アルゴリズムを，実用的な時間で利用できるようにすることである．

本論文は以下のように構成される．2 章では本論文で想定するサンドボックスについて述べる．3 章では提案するシステムの設計について述べる．4 章では提案するシステムの実装について述べる．5 章では関連研究について述べる．6 章ではまとめと今後の課題について述べる．

2. 対象とするサンドボックスシステム

本論文の対象とするサンドボックスシステムの動作について述べる．サンドボックスはアプリケーションによって発行されるシステムコールを待ち，捕捉する．システムコールの捕捉後，システムコールの引数等を取得し，安全性検査を行う．サンドボックスが安全性検査を行っている間，サンドボックス内で実行されるアプリケーションは停止している．

本論文ではサンドボックスシステムという言葉を広範囲のシステムを指すために用いている．本論文で対象とするサンドボックスは侵入検知処理や資源仮想化処理を含みうる．また，本論文で安全性検査と呼ばれている処理には，実資源をもとにして作られた仮想的な資源をアプリケーションに提供するための仮想化処理も含まれる．

本研究は，安全性検査の処理にある程度長い時間がかかることを仮定している．たとえば以下のようなサンドボックスシステムを想定する．

- それまでの実行履歴を考慮する，単純でないアルゴリズムにもとづいてシステムコールの実行の許可，不許可を決定するサンドボックスシステム[14),15)]．
- 計算機資源を仮想化し，仮想資源へのアクセスを実資源へのアクセスに置き換える処理を行うサンドボックスシステム[3),6),8),9),11)]．実資源が遠隔計算機に存在し，サンドボックスがそれをアプリケーションに対して透明な形で取得，提供することもありうる．
- 操作しようとしている資源の中身に応じてシステムコールの実行の許可，不許可を決定するサンドボックス．たとえば，ファイルの中にウィルスと思われるデータが含まれていたら，ファイルのオープンや読み出しのためのシステムコールを失敗させるなどの処理を行うサンドボックスシステム．

これらのシステムでは，Janus[5)]のような単純な構造を持つサンドボックスシステムに比べて，安全性検査のオーバヘッドが大きくなる．

本研究の主眼は広範囲のサンドボックスシステムに適用可能な一般的な枠組みを示すことと，そのための評価を行うことにある．よって，本研究では，サンドボックスが実行する具体的な安全性検査の内容については特に指定しない．

3. 設　　計

安全性検査を投機実行するサンドボックスシステムを説明する．

3.1 基 本 構 成

提案するサンドボックスシステムの構成を図 1 に示す．提案システムは，既存のサンドボックスシステムと同様の処理を行う**モニタプロセス**と，投機的に安全性検査を行う**投機実行プロセス**の二つのプロセスから構成される．アプリケーションはモニタプロセスの子プロセスとして実行される．アプリケーションが実行されたときの提案システムの基本的な動きを図 2 に示す． アプリケーションが発行したシステムコールはモニタプロセスによって捕捉される．モニタプロセスは捕捉したシステムコールの安全性検査に関する処理の制御を行う．モニタプロセスは必要に応じて投機実

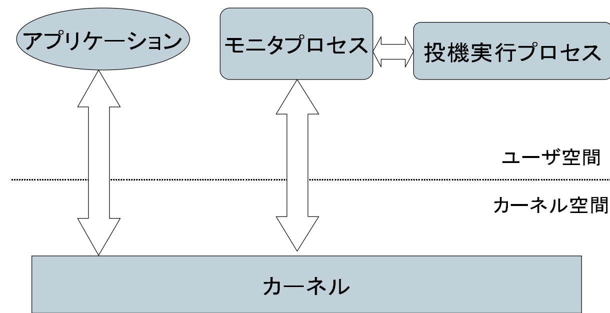

図 1　提案するサンドボックスシステムの構成

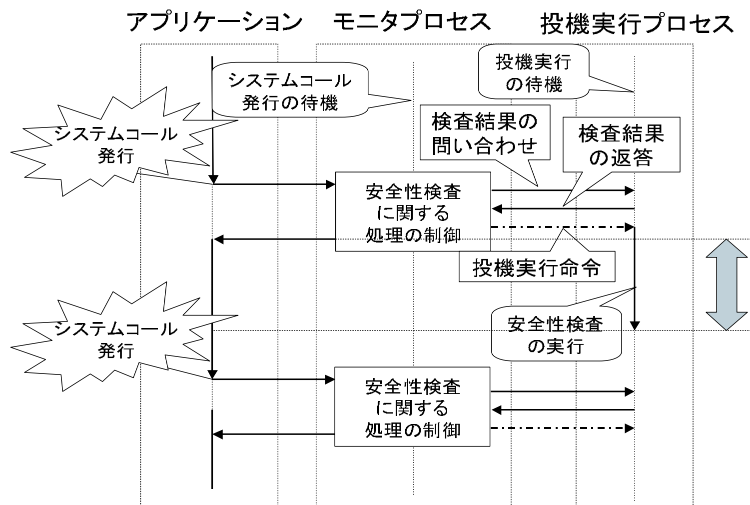

図 2　提案システムにおける状態遷移

行プロセスと通信を行ってシステムコールの実行を制御する．モニタプロセスはアプリケーションでシステムコールコールが呼び出されると，そのとき得られる情報をもとに安全性検査の問い合わせし，その結果を用いて，システムコールの制御を行う．次に，同じ情報を用いて，呼び出されるシステムコールを予測し，システムコールで必要な安全性検査の投機実行を開始する．安全性検査の問い合わせ，投機実行の命令という順序にした要因は，問い合わせ結果に依存して，投機実行命令をする必要がなくなる場合があると考えられるからである．

図 2 中で，本研究がもたらす利点であるアプリケーションと安全性検査の並列に実行される時間を右端の太い上下矢印で表している．また，モニタプロセスが最初のシステムコールで投機実行命令を発行し，次のシステムコールで投機実行命令をを発行しない例を示している．

3.2　安全性検査の投機実行

提案システムにおける安全性検査の処理の制御を図 3 に示す．まず，基本的な動きを述べる．モニタプロセスはシステムコールが発行されたら，安全性検査を行う必要のあるシステムコールであるかどうかを調べる．検査の必要のあるシステムコールの場合は，そのシステムコールに対する安全性検査が投機的に行われているかどうかを調べる．検査の必要が無い場合には投機実行命令の発行の処理へ遷移する．投機的に実行されている場合には以下の処理を行う．まず，投機実行プロセスが投機的な安全性検査を完了しているか調べる．完了している場合には，その結果に基づいて，捕捉したシステムコールの制御を行う．完了していない場合，投機実行プロセス による安全性検査の完了を待機する．完了後，捕捉したシステムコールの制御を行う．一致していない場合には以下の処理を行う．投機実行プロセスによる投機実行を（まだ終わっていなければ）中断するように投機実行プロセスに伝える．その後，モニタプロセスが自ら安全性検査を実行し，その結果を用いてシステムコールの制御を行う．モニタプロセスはさらに，捕捉したシステムコールの情報

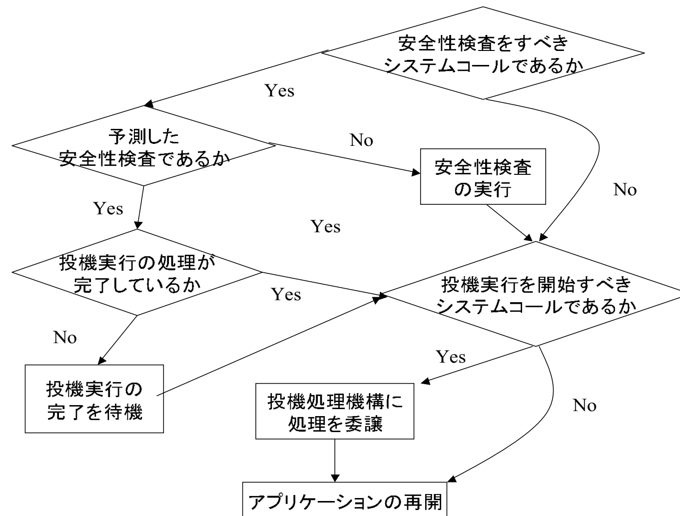

図 3　安全性検査に関する処理の制御

を用いて，次に発行されると予測するシステムコールの安全性検査に関する情報を投機実行プロセスに伝える．予測方式は 3.3 節で述べる．

どのシステムコールを捕捉するかはトレードオフの関係にある．本研究では安全性検査に関する処理を行うシステムコールに加え，行わないシステムコールを捕捉するとしているのは，予測の精度を高めたり，投機実行開始の地点を増やすためである．もちろん，安全性検査を行うシステムコールのみを捕捉する場合に加え，余計な捕捉が入ってしまい，オーバーヘッドが大きくなる可能性もある．

提案システムでは一部のシステムコールについては安全性の検査を行わないようにすることもできる．たとえば sbrk() の実行に関して安全性を特に検査せず，必ず実行を許可するようにすることができる．そのようなシステムコールが発行されたら，モニタプロセスは即座にシステムコールの実行を再開させる．また，そのようなシステムコールが次に実行されると予測されたら，投機実行は行わない．

投機的に実行すべき安全性検査が複数あるときは，各々の投機実行を並列に行う．そのような場合が発生すると考えられるのは，次に発行されると予測されるシステムコールの候補が複数あるときや，あるシステムコールの実行に対して複数の観点から安全性検査を行う必要があるときである．たとえば資源仮想化のためのプログラムモジュールと侵入検知のためのプログラムモジュールの両方を通じてシステムコールの検査を行うときである．

3.3　将来呼び出されるシステムコールの予測方法

提案システムでは将来実行される可能性のある安全性検査を高精度で予測することが重要となってくる．予測が外れた場合，行われた処理が無駄になってしまう．

以下では，次に発行されるシステムコール（引数情報も含む）を予測する方法を述べる．その方法では，アプリケーションが過去に発行したシステムコール列のプロファイルをもとに，そのアプリケーションが発行するシステムコール列を表現するオートマトンを作成する．オートマトンは各ユーザが作成する．そして，そのオートマトンを利用して，必要な安全性検査の投機実行命令を発行開始時点と実行すべき命令を予測する．投機実行に関する決定は各ユーザが決定する．単純にそのオートマトンをもとに次に発行されるシステムコールを予測する方法もある．

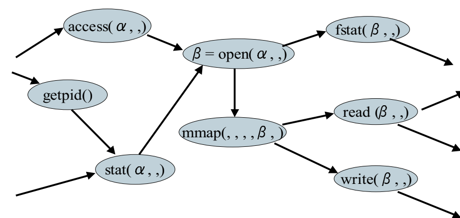

図 4　オートマトンの例

サンドボックスシステムに与えるプロファイルを生成するために，各ユーザはアプリケーションを複数回，システム上で実行する．生成されたプロファイルをもとに，アプリケーションの発行するシステムコール列

に関するオートマトンを作成する．作成されるオートマトンの例を図 4 に示す．図 4 中の α，β は各々，ファイル名，ファイル記述子を表す．たとえば，ユーザはこのオートマトンからファイル名 α を open() するために，getpid()，stat() のシステムコール列が発行されたとき，stat() 時に open() の安全性検査を行うように設定することができる．また，getpid() 時に open() に関する安全性検査を発行するようにも設定することができる．

4. 実　　装

提案システムの実装に関する説明を行う．モニタプロセスはアプリケーションの実行開始前に投機実行プロセスを作成する．作成したサンドボックスシステムはアプリケーションが発行したシステムコールを ptrace() を用いて捕捉する．ptrace() を用いた場合，アプリケーションが呼び出した，すべてのシステムコールを捕捉する．このため，3.2 節で述べたような，安全性検査を行うシステムコール以外を捕捉することによる影響が小さいと考えられる．各システムコールごとに安全性検査の確認，投機実行を開始するための関数ポインタを用意した．安全性検査の処理の制御のためにどちらか一方だけを用いることも可能である．現在，投機実行プロセスの生成には clone() を用い，モニタプロセスとアドレス空間を共有するようにした．これはモニタプロセス で得たアプリケーションに関する情報を投機実行プロセスで用いるためである．

5. 提案システムの評価

提案した安全性検査を投機実行するサンドボックスシステムの実験を行った．実験環境は CPU Pentium IV 3.2 GHz （Hyper Threading 有効），メモリ 2G バイトの Linux 2.6.6 である．本論文では，以前述べたように，特定のサンドボックスシステムを対象としていない．そのため，実験では安全性検査の負荷は擬似的なもので代用することにした．

実験では，安全性検査を投機的に行う場合と行わない場合とで，stat()，open() 文字列のコピー，open()，read()，close() をくりかえし 1000 回実行する．文字列のコピーと open() で行われる安全性検査の処理時間を以下の 3 通りで変化させる．

- テスト A：
  文字列のコピーと安全性検査の負荷は，どちらも 4K バイトの strncpy() を 10 回．
- テスト B：
  文字列のコピーと安全性検査の負荷は，どちらも 4K バイトの strncpy() を 50 回．
- テスト C：
  文字列のコピーと安全性検査の負荷は，どちらも 4K バイトの strncpy() を 100 回．

実験結果を表 1 に示す．表 1 は各テストにおけるプログラム全体の時間を示し，指標は 1 より小さければ提案システムの方が有用であるということを表している．この実験結果から，どの程度アプリケーションと安全性検査に関する実行が並列に行われれば提案システムが有用であるかがわかる．現在の実装では投機実行を行う場合と行わない場合の境界はテスト B の負荷より少し小さいあたりである．今後，実装を洗練させることで提案システムの有用性を高めていきたい．

6. 関 連 研 究

提案システムと同じくシステムコールの実行制御によって安全性を保証するサンドボックスシステムは過去に非常に多く提案されている[2)~13)]．しかしながら，本論文で述べたような安全性検査の投機実行を行うサンドボックスシステムはこれまでに提案されていない．

文献 16) のシステムでは，アプリケーションと並列に走る投機的スレッドが安全性検査の処理を行う．アプリケーションプログラムの中で，安全性を検査する関数を投機的であると指示すると，特殊なハードウェアがその関数とその関数呼び出し以降の部分を並列に実行する．その研究は，アプリケーションのコードと安全性検査の処理を並列に実行して性能を向上させるという基本アイデアにおいて本研究と共通している．しかし，二つの研究は問題設定が異なる．文献 16) は，アプリケーションに埋め込まれた検査コードのオーバヘッドの削減という問題に対して，特殊ハードウェアによる複数の命令列の並列実行を提案している．本論文は，サンドボックスの安全性検査のオーバヘッドの削減という問題に対して，マルチプロセッサによる検査コードの投機的実行を提案している．

Shadow guarding[17)] は，アプリケーションを実行する主プロセスと，そのアプリケーションの実行の安全性の検査を行う影プロセスを，通信させながら並列に実行することにより，小さいオーバヘッドで安全性検査を行う方式である．影プロセスが実行するプログ

表 1　実験結果（秒）

	投機実行		指標
	有	無	(有/無)
テスト A	0.205	0.186	1.102
テスト B	0.648	0.705	0.919
テスト C	1.202	1.356	0.886

ラムは，元のアプリケーションを抽象化して，安全性検査に関係する部分のみを実行するようにしたものである．影プロセスはポインタおよび配列のアクセスの検査やメモリリークの検出を行う．Shadow guarding はアプリケーションを抽象化して得られた検査コードを並列に実行するものである．一方，本研究はサンドボックスの安全性検査を並列に実行する．

7. まとめと今後の課題

投機的な安全性検査を行うサンドボックスシステムを構築する方式を述べた．我々は提案方式に基づいて実際にシステムを実装し，予備実験を行った．その実験においては，安全性検査のための処理がかなり重いものであり，かつ，それがアプリケーションの実行とオーバラップすれば，投機実行の効果は出るという結果が得られた．しかし，そのような条件が成り立たない状況では，投機実行によって逆にオーバヘッドが増大した．現在は実装がまだ未熟なため，投機実行の効果を出ない場合が多いが，実装を改良することにより，多くのアプリケーションで投機実行の効果が出るようにすることを予定している．

今後の課題としては，実際に既存のサンドボックスに適用し，評価を行いたい．また，複数の安全性検査の投機実行を並列に行う有用性についても調査したい．ファイルへの書き込みのような状態を変化させてしまう処理の投機実行に関しても考察して行きたい．OS の内部に変更を加えることにより性能を向上させたい．

参 考 文 献

1) 大山恵弘: ネイティブコードのためのサンドボックスの技術, コンピュータソフトウェア, Vol. 20, No. 4, pp. 55–72 (2003).
2) Acharya, A. and Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications, *Proceedings of the 9th USENIX Security Symposium*, Denver (2000).
3) Alexandrov, A., Kmiec, P. and Schauser, K.: Consh: Confined Execution Environment for Internet Computations, http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps (1998).
4) Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain: Parsimonious Server Security, *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, New Orleans, pp.355–367 (2000).
5) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E. A.: A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker, *Proceedings of the 6th USENIX Security Symposium*, San Jose, pp.1–13 (1996).
6) Liang, Z., Venkatakrishnan, V. N. and Sekar, R.: Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs, *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)* (2003).
7) Coleman, M.: Subterfugue, http://subterfugue.org/.
8) Kato, K. and Oyama, Y.: SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation, *Software Security – Theories and Systems*, Lecture Notes in Computer Science, Vol. 2609, pp. 112–132 (2003).
9) 大山恵弘, 神田勝規, 加藤和彦: 安全なソフトウェア実行システム SoftwarePot の設計と実装, コンピュータソフトウェア, Vol. 19, No. 6, pp. 2–12 (2002).
10) 東村邦彦, 松原克弥, 相河亨, 加藤和彦: モーバイルオブジェクトシステム PLANET のプロテクション機構, 第 1 回インターネットテクノロジーワークショップ論文集 (WIT '98), 高知工科大学, 日本ソフトウェア科学会 (1998 年).
11) 石井孝衛, 新城靖, 板野肯三: プロセストレース機能を用いた世界 OS の実現, 情報処理学会論文誌, Vol. 43, No. 6, pp. 1702–1714 (2002).
12) 榮樂恒太郎, 新城靖, 板野肯三: システム・コールに対するラッパ/リファレンス・モニタ SysGuard の設計と実現, 情報処理学会論文誌, Vol.43, No.6, pp. 1690–1701 (2002).
13) 品川高廣, 河野健二, 高橋雅彦, 益田隆司: 拡張可能コンポーネントのためのカーネルによる細粒度軽量保護ドメインの実現, 情報処理学会論文誌, Vol. 40, No. 6, pp. 2596–2606 (1999).
14) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156–168 (2001).
15) 阿部洋丈, 大山恵弘, 岡瑞起, 加藤和彦: 静的解析に基づく侵入検知システムの最適化, 情報処理学会論文誌, Vol. 45, No. SIG 3 (ACS 5), pp. 11–20 (2004).
16) Oplinger, J. T. and Lam, M. S.: Enhancing Software Reliability with Speculative Threads, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 184–196 (2002).
17) Patil, H. and Fischer, C.: Efficient Run-time Monitoring Using Shadow Processing, *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*, pp. 119–132 (1995).