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

2005－ARC－164 (5)
2005／8／3

実行の振舞いを鍵情報とする不正プログラムの動的検出方式

井上弘士†‡ 岩佐崇史§

本稿では，コンピュータ・システムの安全性向上を目的とした，動的プログラム認証方式を提案する．また，その安全性に関する定性的評価，ならびに，コスト/性能オーバヘッドに関する定量的評価を行う．本方式では，実行の振舞いを共通の秘密鍵情報として利用する事で，1)低い性能オーバヘッド，ならびに，2)連続的なプログラム認証，を可能にする．アプリケーション発行側では，共通秘密鍵から決定される「プログラム実行の振舞い」を実現するオブジェクト・コードを生成する．一方，利用者側では，専用プロファイラを用いて鍵となる実行の振舞いを動的に検出する．もし，「鍵としての実行の振舞い」が検出できなかった場合にはプロセッサに実行停止割り込みを発行する．

Run-Time Intrusion Detection based on Dynamic Execution Behavior

KOJI INOUE †‡ AND TAKAFUMI IWASA §

To challenge the security problem, we propose a hardware-base intrusion detection technique which regards the dynamic program-execution behavior as a certification key. Based on secret key information, we determine an execution behavior. Then an object code which generates the determined execution behavior at run time is constructed by a secure compiler. While the program execution, a secure profiler monitors the execution behavior. If the secure profiler can not see the determined behavior, it alarms the microprocessor for terminating the current program execution. Since the viruses do not know the behavior required to continue the execution on the microprocessor, we can detect and prohibit the malicious attacks at the beginning of its execution.

1. はじめに

近年，コンピュータ・ウイルスなどの悪質プログラム(以下，不正プログラムと呼ぶ) による被害が深刻な問題となっている．このような不正プログラムは，ユーザが気づかないうちにコンピュータ内部に進入し，突然暴走することでその被害を拡大する．今後，社会システムの更なる情報化を進めていくためには，不正プログラム検出ならびに実行防止技術の確立が必要不可欠となる．そこで我々は，不正プログラム問題の解決策として「実行の振舞いを鍵情報とする動的プログラム認証方式」を提案している．本方式では，プログラム発行者と利用者が共通秘密鍵を有する場合を前提とし，プログラムのコンパイル時に鍵情報を実行振舞いとしてオブジェクト・コ

† 九州大学大学院システム情報科学研究院 〒816-8580 福岡県春日市春日公園 6-1
‡ 科学技術振興機構さきがけ 〒332-0012 埼玉県川口市本町 4 丁目 1 番 8 号
§ 福岡大学工学部電子情報工学科 〒814-0133 福岡県福岡市城南区七隈 8-19-1

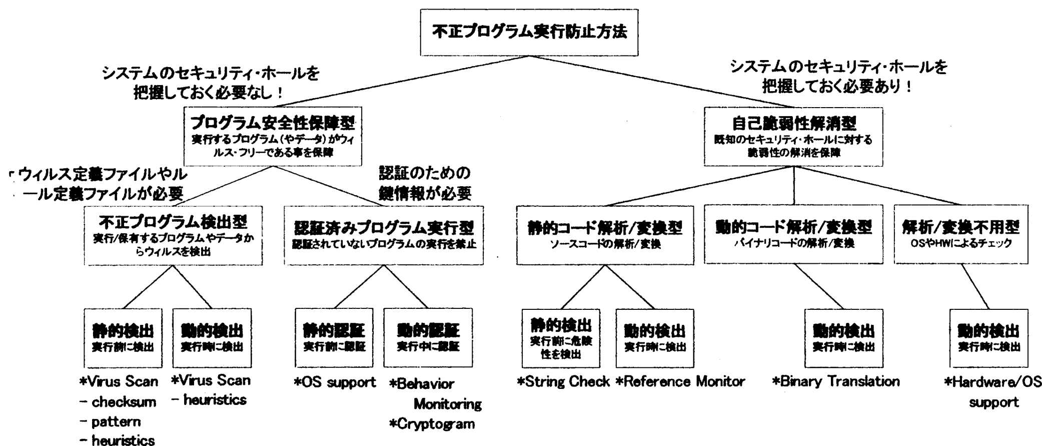

図1：不正プログラム実行防止技術の分類

ード中に挿入する．そして，当該プログラムが実行される際，「鍵情報としての実行の振舞い」を常に監視し，鍵となる振舞いが検出されなかった場合にはこれを不正プログラムと判断する．つまり，プログラム実行という最終段階に鍵情報を適用することで，静的プログラム認証では解決できない不正プログラムの突然の暴走も防ぐことが可能となる．

このような動的プログラム認証方式では，如何にして「コンパイル時におけるプログラム実行振舞いの制御可能性」を高めるかが極めて重要となる．通常，プログラム実行の流れは入力データに依存するため，コンパイル時に実行の振舞いを完全制御することは難しい．そこで，本研究では，基本ブロック・サイズの統一による実行振舞い制御可能性を検討する．また，基本ブロック・サイズを統一した場合に発生するコード・サイズの増大やそれに伴う性能低下に関して定量的評価を行う．

以下，第2節では不正プログラム実行防止技術を分類・整理し，第3節では関連研究について述べる．第4節では本稿で提案する動的プログラム認証方式について説明し，その安全性に関する考察を行う．さらに，第5章ではベンチマーク・プログラムを用いた定量的評価を行い，コード・サイズならびに性能オーバヘッドを明らかにする．最後に第6章で簡単にまとめる．

2. 不正プログラム実行防止技術の分類

これまでに様々な不正プログラム実行防止技術が考案されてきた．また，その一部はすでに実用化されている．不正プログラム実行防止技術は主に，以下2つの方式に大別できる．

- **自己脆弱性解消型**：システム内部に存在するセキュリティ・ホールを活用する不正プログラムの実行を防止する．つまり，静的もしくは動的にセキュリティ・ホールへの攻撃（または攻撃可能性）を検出し，それに対する防御を行う．例えば，バッファ・オーバフロー脆弱性に基づくスタック・スマッシングを検出する方法などが多く提案されている[3][4][7][8]．本方式の利点は，対処済みのセキュリティ・ホールを攻撃する全ての不正プログラムに関して，その実行を防止できる点である．当然，防御の対象となるセキュリティ・ホールは既知でなければならないが，システム内部の全てのセキュリティ・ホールを事前に把握する事は極めて難しい．

- **プログラム安全性保障型**：実行対象となるプログラムが不正コードを含まない事を保障する（安全性を保障する）．つまり，システム内部に如何なるセキュリティ・ホールが存在するとしても，実行対象となるプログラムが安全であれば問題は発生しないという考えに基づいている．本方式では，システム内部に存在するセキュリティ・ホールを事前に把握しておく必要がないといった利点がある．実際，実用化されている不正プログラム実行防止技術の多くは本方式に分類される．

プログラム安全性保障型は，更に以下のように分類できる．

- **不正プログラム検出型**：ウィルス定義ファイルやルールファイル等に基づき，システム内部に不正プログラムが存在しないか検査する．本方式ではアプリケーション・プログラムに対して特別な処理を加える必要がないため，その適用は比較的容易である．実際，ウィルス・スキャン等の技術は実用化が進んでおり，すでに多くのコンピュータ・システムに搭載されている．しかしながら，本方式の最大の欠点は未知の不正プログラム（ウィルス定義ファイルやルールファイルに登録されていない不正プログラム）の検出ができない点にある．

- **認証済みプログラム実行型**：アプリケーション・プログラムの安全性が保障されたコードのみを実行する[2][5][6]．つまり，プログラムを実行するコンピュータ・システムにおいて，安全であると認証されたプログラムのみが実行可能となる．したがって，未知の不正プログラムに対しても対応することができる．ただし，プログラムの実行に関して，静的または動的なプログラム認証処理が必要となる．

3. 関連研究

より高い安全性を実現するためには，第2節で説明した自己脆弱性解消型とプログラム安全性保障型を組合せて運用するのが望ましい．本稿では，プログラム安全性保障型，特にその中でも，認証済みプログラム実行型に着目する．なお，ここでは，アプリケーション発行者側と利用者側で共通の秘密鍵を有する事を前提とする．

プログラム認証の実現方式は主に静的方式と動的方式に分類できる．これらは，「いつ，プログラム認証を行うか？」によって異なる．静的方式の場合，アプリケーション発行時に正しいプログラムである事を証明する鍵（または証明書）を作成し，プログラムを実行する際にOS（オペレーティング・システム）が照合する方法が考えられる．つまり，OSレベルでのプログラム認証である．しかしながら，最近のウィルスはOSを書き換えるような物も少なくない．また，プログラム起動時にのみ鍵の照合が行われるため，正規のアプリケーション・プログラムを実行中に突然悪質プログラムが暴走し始めた場合にはそれを停止する手段がない．現在のプロセッサは，いったんプログラムの実行を開始すると，それが悪質プログラムであるか否かは関係なくその実行を完了しようとする．そのため，OSレベルでのプログラム認証を通過した悪質プログラムはコンピュータ・システム内で暴走し多大な被害を与える．

一方，動的方式では，プログラム・コードの暗号化がある．この方式は主にソフトウェアの違法コピー防止を目的として考案された．本技術は，不正プログラムの実行防止にも応用できる．つまり，プロセッサは秘密鍵によって復号化したコードのみを実行する．言い換えると，秘密鍵を用いて暗号化されたプログラム・コードのみが正しく実行される．よって，不正プログラムも正しい秘密鍵で暗号化されない限り，プロセッサはその実行を阻止できる．例えば、XOM (eXecute-Only Memory)ではオフチップ・メモリのデータは全て暗号化されている事を前提としており，それを実現するためのマシン・モデルを提案している[5]．その他，多くの研究でも暗号化を前提としたセキュア・プロセッサを提案している[1][9]．暗号化方式の利点は，暗号アルゴリズムによって保障される（例えばAESなど）安全性の強度をそのまま実行コードに適用できる点にある．しかしながら，復号化に伴う性能オーバヘッドが問題となる．この問題を解決するため，多くの研究ではL2キャッシュを前提とし，データ（実際はライン単位）がオンチップ・キャッシュにフィルされる際に復号化を行っている．

暗号化以外の方法としては，実行対象プログラムの振舞いに着目した方法がある．事前にプログラムの静的解析を行い，実行に関する情報を採取する（ここでは実行ルールと呼ぶ）[2][6]．利用者は，実行対象プログラムに加え，この実行ルールもアプリケーション発行者より入手する．そしてプログラム実行中，実行ルールに基づく振舞いをするか動的にチェックする．もし実行ルールに定義されていない動作が検出された場合には不正プログラム（つまり，未認証プログラム）が実行されたと判断する．例えば，文献[6]ではシステム・コールの呼出しシーケンスを実行ルールとして定義する．もし，実行ルールで未定義のシステム・コール・シーケンスが検出された場合には不正プログラムと判定する．こ

れは，重大な被害を引き起こすアタック・コードは多くの場合システム・コールを利用するという点に着目している．

4. 実行振舞いを鍵情報とする動的プログラム認証方式

4.1 基本アイデア

不正プログラムの暴走に関する本質的問題は，マイクロプロセッサ自信が実行対象プログラムの安全性を全く考慮せず，ただ単にその実行を完了しようと試みる点にある．したがって，この問題を解決するためには，不正プログラム等の認定されていない（鍵情報を持っていない）プログラム・コードは必ず実行できない機能をプロセッサ自身に搭載する必要がある．

第3節で述べた動的プログラム認証方式では，実行中に対象プログラムが安全か否かを判定する．しかしながら，暗号化に基づく方式では復号化に伴う実行時間オーバヘッドが発生する．L2 キャッシュを搭載する高性能な単一プロセッサではそのオーバヘッドを十分隠蔽可能である．しかしながら，組込み用プロセッサのように小容量L1キャッシュのみを搭載する場合や，L2 キャッシュを共有するCMP方式の場合は，L1キャッシュ・フィル時に復号化を行う必要があり，性能オーバヘッドがより顕著になる．一方，実行ルールに基づく動的認証方式では，その多くがソフトウェア実装であり極めて大きな性能低下を伴う．また，実行ルールを満足する攻撃コードに関しては全く対応できない．何らかの方法で攻撃者がソースコードやその実行履歴を入手した場合，実行ルールにあわせたアタック・コードを生成可能である．

そこで本稿では，性能オーバヘッドが小さく，かつ，ソースコードに依存しないプログラム認証方式として，実行の振舞いを鍵情報とする動的プログラム認証方式を提案する．提案方式の基本動作を図2に示す．ここでは，利用者側とアプリケーション発行側で共通の秘密鍵を保有していると仮定している．アプリケーション発行側では，この秘密鍵から決定される「プログラム実行の振舞い」を実現するオブジェクト・コードを生成する．一方，利用者側では，秘密鍵から決定される「プログラム実行の振

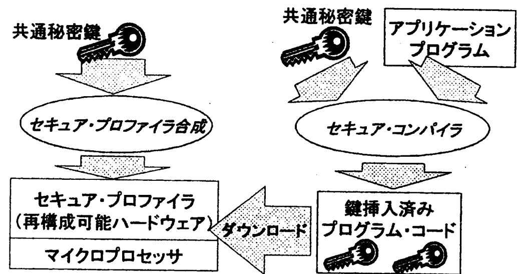

図2：実行振舞いを鍵情報とするプログラム認証

舞い」を検出するための専用プロファイラを生成し，再構成可能ハードウェア上に実装する（複数の秘密鍵を有する場合等にも対応できるよう，専用プロファイラの実装デバイスは再構成可能ハードウェアとしている）．そして，アプリケーション・プログラム実行時，このプロファイラによって常に実行の振舞いを監視する．もし，「鍵としての実行の振舞い」が検出できなかった場合には即座にプロセッサに実行停止割り込みを発行する．なお，ここではプログラム発行者側と利用者側でのデータ転送は暗号化技術等により安全性は保たれていると仮定する．

4.2 鍵情報としてのメモリ参照パタン

鍵情報となる実行の振舞いに関しては様々なものが考えられる．本稿では，その一例としてメモリアクセス・パタンに着目する．メモリアクセス・パタンはマイクロプロセッサ・コア内部に手を加える事無く，外部で観測可能なためである．具体的には，ある特定アドレスへのロード命令を「鍵ロード命令」と呼び，「N 命令毎に鍵ロード命令が必ず実行される」という実行の振舞いを鍵情報とする．具体的には以下のような処理の流れとなる．

1. アプリケーション発行者は，プログラムのコンパイル時，必ず N 命令毎に鍵ロード命令が実行されるようコードを生成する．
2. 利用者側では，プログラムの実行を開始する前に鍵検出ハードウェア（再構成可能ハードウェア）を準備する．ここでは，実行命令カウンタ，ならびに，メモリ参照モニタが必要となる．
3. プログラム実行を開始すると同時に，鍵検出ハードウェアによって実行の振舞いをモニタす

る．もし，N 命令毎に鍵ロード命令が実行されない（つまり，特定アドレスへのロードが発生しない）場合は不正プログラムの実行と見なす．

4.3 静的な実行振舞い制御

第 4.2 節で説明したように，本提案方式ではコンパイル時にプログラム実行の振舞いを制御しなければならない．しかしながら，実際には以下のような非決定的要因により実行命令数や命令実行パタンは異なる．

- 条件分岐命令の実行に伴う制御フローの変動
- 予測技術に基づく投機実行による実行命令数の変動
- アウト・オブ・オーダ実行による実行命令順序の変動

4.3.1 条件分岐による制御フロー変動への対応

プログラム実行の流れは入力データに依存する．一般に，各基本ブロック・サイズはそれぞれ異なるため，実行振舞いを生成するよう各鍵命令を一定間隔に挿入することが困難となる．条件分岐命令の実行結果によって実行命令数が異なるためである．この問題を解決するため，プログラムの全基本ブロック・サイズを同一サイズに統一する．そして，各基本ブロックの同一位置に鍵ロード命令を挿入することで，固定命令数毎に鍵ロード命令が実行される．基本ブロック・サイズを統一するために冗長な NOP 命令を挿入する必要があり，コード・サイズの増加ならびにプログラム実行時間の増大といった欠点がある．

4.3.2 投機実行による実行命令数の変動

マイクロプロセッサが予測技術に基づく投機実行をサポートする場合，予測結果によって実行命令数が異なる．これに対応するためには，実行振舞い検出回路側にて実行命令数カウントを調節する必要がある．つまり，予測ミスに伴う実行のやり直し（ロールバック）が発生した際，それに応じて実行命令カウンタの値を増減する．

4.3.3 OOO 実行による実行命令パタンの変動

本方式では N 命令毎に鍵ロード命令が実行されなければならない．しかしながら，OOO 実行においては実行される命令のオーダが異なるため，静的な実行の振舞いは極めて難しくなる．OOO プロセッサを対象とする場合，命令でコード時，または，コミット時に実行の振舞いを検出する方法が考えられる．

4.4 安全性に関する考察

本節では，提案方式における安全性（不正プログラム実行の検出可能性）を定性的に考察する．

- **鍵情報としての実行振舞いの強度**：攻撃者に対して鍵ロード命令出現間隔 N，ならびに，鍵ロード命令参照アドレスを見破られた場合，それを正しく反映した攻撃コードに対しては不正実行を検出できない．しかしながら，本提案方式では共通秘密鍵に基づき鍵ロード命令出現間隔 N ならびに鍵ロード参照アドレスを決定する．したがって，共通秘密鍵で実現できる安全性をある程度維持できる．一方，プログラム実行中にメモリ・バスを観測する事で鍵情報を推測される可能性がある．しかしながら，本提案方式では実行振舞い検出回路はプロセッサと同一チップに搭載される．したがって，実際には鍵ロード命令の実行時にオフチップ・アクセスを実施する必要はない．

- **鍵ロード命令実行間隔**：攻撃者が鍵情報となる実行の振舞いを知らない場合でも，鍵ロード命令出現間隔 N より短い（実行命令数の少ない）アタック・コードの実行は検出できない．しかしながら，第 4.3.1 節で説明した制御フロー変動による対応においては，各基本ブロックを 5〜30 命令程度のサイズに統一する．各基本ブロックに 1 個の鍵ロード命令を挿入する場合，鍵ロード命令出現間隔は 5〜30 命令程度となる．よって，不正プログラムを実行するためには，その実行命令数を 5〜30 命令程度に抑制しなければならない．しかしながら，このような少数命令での攻撃は非常に難しいと推測される．

5. 性能/コスト・オーバヘッドの評価

組込み用途で多く使用される StrongARM プロセッサを想定し，提案手法適用における性能ならびにコード・サイズのオーバヘッドを評価した．実際には SimpleScalar/ARM[10]を使用している．命令発行はインオーダであり，分岐予測は採用していない．ベンチマーク・プログラムは SPECint95 の 129.compress を用いた。

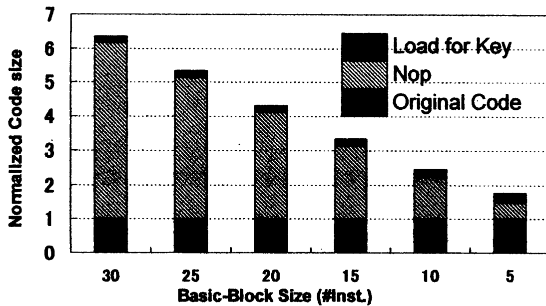

図 3: コードサイズ・オーバヘッド

図 3 に提案手法適用に伴うコードサイズ・オーバヘッドを示す. 全ての結果は, 本手法適用前のコード・サイズで正規化している. この結果より, 統一する基本ブロック・サイズの減少に伴い, 挿入される鍵ロード命令数の割合が大きくなっている事が分かる. 逆に, 統一する基本ブロック・サイズを大きくすると, 冗長な NOP 命令の挿入による影響が顕著となる. 基本ブロック・サイズを 5 命令に統一した場合でも約 80%コード・サイズが増加した.

一方, 性能オーバヘッドに関する実験結果を図 4 に示す. 全ての結果は, 本手法適用前のプログラム実行時間で正規化している. この結果より, 統一基本ブロック・サイズを 5 命令に統一した場合の性能オーバヘッドは 9%程度であることが分かる. 以上より, コード・サイズの増加は依然として大きいが, 不正プログラムの実行防止を優先する場合には 10%以下の性能低下は許容範囲であると考える.

6. おわりに

本稿では, 不正コードの実行防止を目的とした動的プログラム認証方式を提案した. また, 安全性に関する考察, ならびに, コード・サイズ/性能に関するオーバヘッドを評価した. その結果, コード・サイズ増加率は大きいものの, 10%以下の性能低下で提案手法を実装可能であることが分かった. 今後, より詳細な安全性に関する評価, ならびに, プロトタイプ・システムの設計を行う予定である.

謝辞

本研究を遂行するにあたり, 多くのご意見を頂いた科学技術振興機構さきがけプロジェクト「情報基盤と利用環境」領域関係者各位に感謝します. なお, 本研究は一部, 文部省科学研究費補助金(課題番号: 14GS0218, 17680005) による.

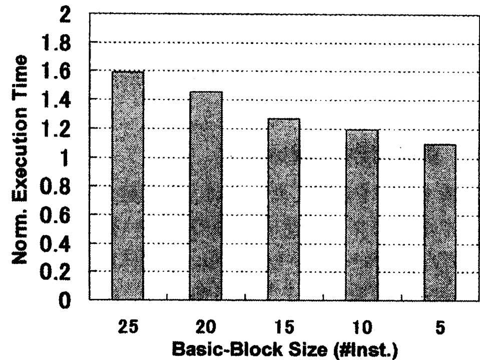

図 4 : 性能オーバヘッド

参 考 文 献

[1] G. Edward Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas,"Efficient Memory Integrity Verification and Encryption for Secure Processors," Int. Symp. on Microarchitecture, pp.339-350, Dec. 2003.

[2] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff, "A Sense of Self for Unix Processes," Proc. of the 1996 IEEE Symposium on Security and Privacy (S&P), pp.120-128, 1996.

[3] K. Inoue, ``Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," ACM Computer Architecture News, vol33, no.1, pp.81-89, Mar. 2005.

[4] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," Int. Conf. on Security in Pervasive Computing, Mar. 2003.

[5] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[6] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," IEEE Symposium on Security and Privacy (S&P01), pp.156-168, May 2001.

[7] U.Erlingsson and F.B.Schneider, "SASI Enforcement of Security Policies: A Retrospective," Proc. of the workshop on New security paradigm, 1999.

[8] D.Wagner, J.S.Foster, E.A.Brewer, and A.Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," Proc. of the Network and Distributed System Security Symposium, Feb. 2000.

[9] J. Yang, Y. Zhang, and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," Proc. of the Int. Symp. on Microarchitecture, pp.351-360, Dec. 2003.

[10] SimpleScalar Tool Sets, http://www.simplescalar.com/.