コンパイラと OS の連携による強制アクセス制御向け静的解析

森山　壱貴 †　　表　雄仁 †　　桑原　寛明 ‡　　毛利　公一 ‡　　齋藤　彰一 ††
上原　哲太郎 ‡‡　　國枝　義敏 ‡

† 立命館大学大学院　理工学研究科　　‡ 立命館大学　情報理工学部
†† 名古屋工業大学大学院　工学研究科　　‡‡ 京都大学　学術情報メディアセンター

あらまし　多くのセキュア OS は，正常なアクセスを定義したポリシーに基づく，強制アクセス制御機構を備えている．しかし，適切なポリシーの記述は通常，非常に困難である．そこで，本研究では，コンパイラと OS の連携によって強制アクセス制御を行う手法を提案する．コンパイラの静的解析技術により，システムコールに着目してソースコードから正常なアクセスを表現するポリシーを自動生成し，OS は生成されたポリシーに基づいて強制アクセス制御を実現する．本稿では，生成するポリシーを検討し，静的解析技術を用いたポリシーの生成手法について述べる．

Static Analysis for Mandatory Access Control by Collaboration of Compiler and OS

Ikki Moriyama†　　Yuji Omote†　　Hiroaki Kuwabara‡　　Koichi Mouri‡
Shoichi Saito††　　Tetsutaro Uehara‡‡　　Yoshitoshi Kunieda‡

†Graduate School of Science and Engineering, Ritsumeikan University
‡College of Information Science and Engineering, Ritsumeikan University
††Graduate School of Engineering, Nagoya Institute of Technology
‡‡Academic Center for Computing and Media Studies, Kyoto University

Abstract　Almost all the secure OS have some kind of mandatory access control structure based on the policy which defines each right access. However, it is generally very difficult for users to describe the policy appropriately. In this study, we suggest a technique for mandatory access control by collaboration of compiler and OS. In the proposed system, compiler generates the policy that expresses right access focused on the system call from a given source code by static analysis technique. And OS realizes mandatory access control based on the generated policy. In this paper, we examine the policy to generate, and explain on generation technique of the policy by using static analysis technique.

1　はじめに

近年，報告されるソフトウェアの脆弱性を狙った攻撃で，比較的高い割合を占めるものはバッファオーバーフローによる攻撃であり，高い権限で実行されているプロセスが攻撃されると，悪意のあるコードがその権限で実行される危険性がある．そのため，重大なセキュリティ侵害を引き起こす可能性が高い．こうした脅威からコンピュータの被害を最小限に抑えるため，OS

レベルでの強固なセキュリティ対策が求められている [1].

現状の問題の一つとして，ファイルへのアクセス権をユーザが任意で設定可能なことが挙げられる．このため，多くのセキュア OS は強制アクセス制御機構を備えている．強制アクセス制御とは，各リソースに対し正常なアクセスを定義したポリシーを予め用意し，それに基づいたアクセス制御を全プロセスに強制する機構である．これにより，バッファオーバーフローなどの攻撃によってプログラムの制御を不正に奪取された場合でも，ファイルへのアクセスを禁止できる．すなわち，脆弱性の影響がシステム全体へ拡散することを抑制して，被害を限定化・局所化でき，リスクを最小に抑えることができる．しかし，アクセス制御を定義するポリシーの記述が複雑になるという欠点を抱えており，高い安全性を得るためには，煩雑なアクセス制御設定をアプリケーションごとに行う必要があるため活用されているとは言えない [2]．この煩雑さを軽減する対策として，対象プロセスを実行し，アクセスされたファイルや発行されたシステムコールのログを取ることによってプログラムの挙動を解析し，ポリシーを自動生成 (学習) する機能が提案されている．しかし，完全なポリシーを生成できるわけではなく，その解析結果を雛形としても，必要十分なポリシーの作成は，なお困難な作業である．

本研究では，コンパイラと OS の連携によって強制アクセス制御を行う手法を提案する．コンパイラの静的解析技術により，システムコールに着目してソースコードから正常なアクセスを表現するポリシーを自動生成し，OS はポリシーに基づいて実行時の動的解析と監視機構により強制アクセス制御を実現する [3]．双方の組み合わせによりポリシーの記述作業を削減し，ヒューマンエラーを減少させることによって信頼性の高いシステムを構築することを目指す．プログラムの挙動をその原型であるソースコードからコンパイラの静的解析技術を用いて解析することにより，実行コードから挙動を解析した場合と比較して，詳細なアクセス制御情報を生成することが可能となる．また，システムコールに着目することで，例え侵入されても被害の影響範囲を最小限にし，プログラムの脆弱性に対するリスクを大幅に低減することができる．

本稿の構成を以下に示す．2 章で関連した研究について述べる．3 章で提案手法について述べる．4 章で提案システムのコンパイラ側の設計とプロトタイプの実装について述べる．最後に 5 章でまとめと今後の課題を述べる．

2 関連研究

2.1 ソースコードの静的解析による脆弱性検出

ソースコードの解析技術は，バッファオーバーフローやメモリのリークといったソースコードに内在する脆弱性を検出できるため，ソースコードの安全性や信頼性が向上する．また，セキュリティに関する知識や経験に依存する割合を減らす意味でも有用であると言える．セキュリティに関するソースコードの静的解析の技術としては大きく分けてパターンマッチング技術と，構文解析技術とがある [4].

パターンマッチング技術では，バッファオーバフローを利用した攻撃によって利用されやすい関数をあらかじめデータベース化しておき，ソースコードにそれらの関数が含まれるかどうかを検出する．脆弱性が含まれる既知の関数が検出された場合，ユーザに警告を促す．パターンマッチング技術を用いた研究には，Flawfinder[5] などが存在する．

従来の構文解析技術では，ソースコードを構文解析し，抽象構文木に変換することによって，ソースコード内部の論理的な矛盾点を発見することを目的とした技術である．これに分類されるツールには Splint[6] などが存在する．構文解析技術では，関数呼び出しや変数の関係を詳細に検討することが可能である．

ソースコードの静的解析技術を異常検知に適用した研究としては，文献 [7][8] がある．これらの手法では，ソースコードを静的に解析するものの，プログラムの異常検知は動的に行う．

2.2 コンパイラによるセキュアコード生成技術

セキュアコード生成技術では，コンパイラを拡張し潜在的にバッファオーバフロー脆弱性を内包するソースコードから，それを除去した安全な実行コードを生成する．このようなものには， StackGuard [9]，Stack Smashing Protector [10] などの研究が存在する．それぞれ，実行コードの生成過程において若干の違いはあるが，どちらもリターンアドレスの前にカナリアと呼ばれる値を挿入して，関数呼び出しから戻る際にその値が書き換えられているかをチェックすることによってバッファオーバフローの発生を検知するという手法が用いられている．また， Stack Smashing Protector では内部で用いられている変数や引数の保護機能もあり，カナリアをチェックするだけでは検出できない，関数ポインタを持つ変数を狙ったバッファオーバフロー攻撃に対する保護も可能である．

コンピュータウィルス感染や情報漏洩の危険性のない，対攻撃耐性を持ったコードを生成する高安全 C コンパイラの研究も報告されている [11]．この研究では，機密情報の流れを監視する情報流解析機構を導入する．さらに，各メモリブロックの有限の定義域を取り払い，全アドレス空間に対して仮想的に有効とすることで，すべてのメモリアクセスを必ず成功させるといった手法が用いられている．これにより，バッファオーバーフローなどのメモリ脆弱性を突いた攻撃を受けても安全に動作を継続し，バッファオーバーフローが原理的に起こらないようになる．

3 提案手法

3.1 コンパイラと OS の連携によるセキュアシステム

本システムは，静的解析によりシステムコールに着目した強制アクセス制御情報を生成するコンパイラと，実行時の動的解析と監視を行う OS から成り立つ．システムコールに着目することで，プロセスが侵入者の手に渡った場合，ファイルへのアクセス，他のプロセスとの通信，任意のコード実行といった情報漏洩やデータ改竄などを引き起こす可能性がある挙動を監視・制御することができる．それは，ファイル・ネットワークへのアクセスにはシステムコールを必要とするためである．したがって，システムコールがプログラムの意図する形で発行されていることを監視すればシステムを防御でき，未知の攻撃を防ぐ効果がある [8]．コンパイラは，プログラムが呼び出すシステムコールがどのような状況で発行されるか，どのようなリソースにアクセスするかといった制御情報を生成する必要がある．その制御情報を基に OS による実行時の高度な解析と監視が可能となる．

プログラムの原型であるソースコードは，発行されるシステムコールに関する詳細な情報を保有し，コンパイラにより実行コードにコンパイルされる．また，システムコールは OS のカーネルの機能を呼び出すために使用される機構であるため，OS によって監視・制御を効率的に行うことができる．そこで，コンパイラではソースコードに記述されたシステムコールの種類や呼び出し位置，システムコール発行に伴う引数の取り得る値の範囲や引数に関連するデータの流れを静的解析し，その解析情報からポリシーを生成する．OS ではこうして生成されたポリシーを利用して，システムコールに関する実行時の動的解析により不正な実行，すなわちソースコードに記述されていないシステムコールの発行やアクセスを許可されていない引数の範囲を検出し停止させる．

提案するシステムの構成を図 1 に示す．点線内がコンパイラのシステム構成を示す．コンパ

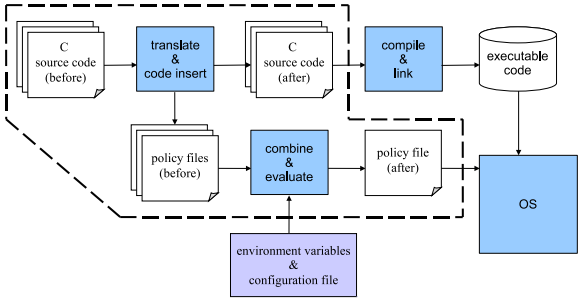

図 1: システムの構成図

イラによる処理フローとして，C 言語を対象としたソースコードを静的解析することで，強制アクセス制御情報であるポリシーと，エラー処理コードや実行時にポリシーの精度を向上させるためのコードが挿入されたソースコードを生成する．ソースコードの分割コンパイルによりポリシーが複数ある場合にはポリシーを１つの結合さる．また，実行時に使用する環境変数や設定ファイルがある場合には，それらの値をポリシーに反映させ，引数の取り得る値をより限定し精度を向上させる．

以上により，本提案システムでは高度な解析能力を有するコンパイラと OS とが，従来にない強固な連携を果たすことでソフトウェアシステムの高度な信頼性を実現する．

3.2 コンパイラによる強制アクセス制御情報の生成

コンパイラでは，ソースコードからシステムコールに着目した強制アクセス制御情報をポリシーとして生成する．生成されるポリシーの内容を以下に示す．

(1) 発行可能なシステムコールの種類

(2) システムコールが呼び出される位置

(3) システムコールの引数が取り得る値の範囲

(4) システムコールが返す値の追跡情報

(1) により，プログラムが発行できるシステムコールの種類を制限することができる．(2) では，システムコールがプログラムのどこから呼び出されるかを特定し，システムコールの発行シーケンスの監視や実行されるはずない位置からのシステムコールの呼び出しを禁止できる．(3) により，システムコールに対する引数の取り得る値の範囲を正規化して定義し，ファイルやネットワークなどのリソースにアクセス可能な範囲を限定する．(4) は，システムコールが返すファイルディスクリプタや値が他のシステムコールの引数に使用される場合の追跡情報を定義したものである．

コンパイラは，ソースコードから制御フローとデーターフローを解析し，制御フローからシステムコールの発行シーケンスを解析する．また，データの流れからファイルアクセスに伴うパスの範囲を抽出・解析する．ソースコードの解析において，ファイルパスを表す文字列を抽出し，当該文字列がプログラム中でどのように加工されるかを解析することにより，正規の挙動で可能なファイルを限定する．これにより，乗っ取られたプロセスが任意のファイルに不正アクセスする事態を防ぐことができる．

3.3 OS よる実行時の動的解析と監視・制御システム

コンパイラが解析した結果を利用して，OS はシステムコール発行のたびに，比較検査・実行可否判定を行う．また，静的には解析できない実行時に決定されるファイルパスについては，実行時のファイルパス確定時点でポリシーの引数の取り得る範囲を参照し，引数の値が規定された範囲内であるかを解析する．もし解析結果にないシステムコール発行や引数であった場合には，当該プロセスを停止させる．

4 設計とプロトタイプの実装

4.1 システムコールの種類と発行位置の解析

システムコールを解析する場合，ソースコードに直接記述されているシステムコールとライブラリ関数の内部で使用されるシステムコールを考慮する必要がある．ライブラリ関数中のシステムコールに関しては予め解析しておき，ソースコードに現れるライブラリ関数に対応した解析結果を適応させてポリシーを生成する．また，システムコールの発行位置として，どの関数から呼び出されるかを特定するため，関数の呼び出しグラフを構築する．また，関数内部のどの位置から呼び出されているかを特定するために，制御フローグラフを構築し，発行シーケンスと制御フローの関連付けを行う．

4.2 引数に対するデータフロー解析

システムコールの実引数に対するデータフロー解析では，主に使用されるファイルパスのアクセス範囲を解析する．引数の取り得る値を限定することで，アクセス可能なファイルパスを絞り込むことができる．対象がファイルパスであるため，特に文字列に対するデータフロー解析を重点的に行う．制御フローグラフを基にしたデータフロー解析と，コンパイラの基本的な最適化として定数伝播と畳み込みを行い，引数の取り得る値の範囲を抽出し正規化する．また，ファイルディスクリプタなどのシステムコールに使用される変数の流れを追跡可能にする情報を OS 側に提供することで，システムコールと引数の関連を実行時に解析可能にし，同じ変数を使用して，一連の処理を行うシステムコール群に対し，適切な動的解析が可能となる．

4.3 プロトタイプの実装

C 言語を対象としたデータフロー解析や最適化を行うための制御フローグラフを持つコンパイラをプロトタイプとして設計し実装を行う．プロトタイプでは，ANSI-C 準拠のフロントエンドと定数伝播や定数の畳み込みを行う基礎的な最適化機能を実装した．さらにソースコードに記述された関数単位で，システムコールを抽出し，その種類と引数のリストを生成する．ファイルへのアクセス，他のプロセスとの通信，任意のコード実行を行うシステムコールの抽出を名前を基に識別する．プロトタイプで使用したサンプルコードと生成されたポリシーをそれぞれ図 2 と図 3 に示す．

プロトタイプが生成するポリシーは，サンプルコードに記述された各関数内のシステムコールについて，種類と引数を列挙したものを関数単位でブロック化し，それらを *syslist* というブロックで表現している．サンプルコードの 33 行目では，file_copy 関数内の open システムコールが `to_name`，`O_WRONLY|O_CREAT|O_TRUNC`，0666 の 3 つの引数を取っている．この部分の内容がポリシーの 10 行目表現されている．2 番目の引数については，畳み込みにより 577 となっている．4.1 節で述べた発行シーケンスと制御フローの関連付けはまだ実装されていないため，システムコールの発行位置を詳細に特定できないが，その関数内部で発行可能なシステムコー

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <fcntl.h>
 4  #include <unistd.h>
 5
 6  void file_copy(char *from_name, char *to_name);
 7
 8  int main(int argc, char *argv[])
 9  {
10    if (argc != 3) {
11      fprintf(stderr, "Usage: %s from to\n", argv[0]);
12      exit(1);
13    }
14    file_copy(argv[1], argv[2]);
15    execve("/bin/ls", argv, NULL);
16    return (0);
17  }
18
19  #define BUFFSIZE 1024
20
21  void file_copy(char *from_name, char *to_name)
22  {
23    int from_fd, to_fd;
24    char buff[BUFFSIZE];
25    int rcount;
26    int wcount;
27
28    from_fd = open(from_name,O_RDONLY);
29    if (from_fd == -1) {
30      perror(from_name);
31      exit(1);
32    }
33    to_fd = open(to_name, O_WRONLY|O_CREAT|O_TRUNC, 0666);
34    if (to_fd == -1) {
35      perror(to_name);
36      exit(1);
37    }
38    while((rcount = read(from_fd, buff, BUFFSIZE)) > 0) {
39      if((wcount = write(to_fd, buff, rcount)) != rcount) {
40         perror(to_name);
41         exit(1);
42      }
43    }
44    close(from_fd);
45    close(to_fd);
46  }
```

図 2: サンプルコード

```
 1  #
 2  # Policy file generated by prototype.
 3  #
 4  syslist {
 5    main {
 6            execve  "/bin/ls"       argv    NULL
 7    }
 8    file_copy {
 9            open    from_name       0
10            open    to_name         577     0666
11            read    from_fd         buff    1024
12            write   to_fd           buff    rcount
13            close   from_fd
14            close   to_fd
15    }
16  }
```

図 3: ポリシーのプロトタイプ

ルの許可リストと見ることができる．すなわち，現在のこのポリシーを使用して実行・監視した場合，ポリシーに記述されていない種類のシステムコールを禁止することができる．

5 おわりに

本稿では，コンパイラの静的解析技術と OS の動的解析・監視機能の連携による，システムコールに着目した強制アクセス制御方式を提案し，特にコンパイラの静的解析技術を用いたポリシー生成手法を述べた．本システムでは，コンパイラの静的解析技術を利用して，従来にない新たな観点からセキュリティ強化を実現する．コンパイラは，システムコールの種類，発行位置，引数の取り得る値の範囲，変数の追跡情報を定義した強制アクセス制御情報をポリシーとして生成する．OS は，ポリシーに基づいて実行時の動的解析と監視機構により強制アクセス制御を実現する．

今後の課題として，データフロー解析によってシステムコールの引数が取り得る値の範囲を求めることが挙げられる．これにより，不正なファイルアクセスを実行時に検出することが可能になる．また，システムコールに着目することで，どのような種類の攻撃からシステムを守ることができるのか明らかにすることも今後の課題である．

将来的には，致命的な脆弱性を含むプログラムに対しても，より厳密なデータフロー解析を行うことで，バグを作り込んでも安全に実行可能なシステムへ発展させることも可能であると考えている．

参考文献

[1] 総務省. セキュア OS に関する調査研究会報告書, 2004. http://www.soumu.go.jp/s-news/2004/040428_1.html.

[2] 情報処理振興事業協会セキュリティセンター. オープンソースソフトウェアのセキュリティ確保に関する調査, 2003. http://www.ipa.go.jp/security/fy14/reports/oss_security/.

[3] 表雄仁, 森山壱貴, 桑原寛明, 毛利公一, 齋藤彰一, 上原哲太郎, 國枝義敏. コンパイラと OS の連携による強制アクセス制御向けプロセス監視手法. *Computer Security Symposium 2007 (CSS2007)*, 2007(投稿中).

[4] バッファオーバフロー対策技術に関する報告書. http://itslab.csce.kyushu-u.ac.jp/ssr/tatara_report.html.

[5] David A. Wheeler's. Flawfinder. http://www.dwheeler.com/flawfinder/.

[6] Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities. *In Proceedings of the 10th USENIX Security Symposium*, pp. 177–189, August 2001.

[7] D.Wagner, D.Dean. Intrusion Detection via Static Analysis. *In Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156–168, May 2001.

[8] 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, May 2004.

[9] C.Cowan, C.Pu, D.Maier, H.Hinton, J.Walpole, P.Bakke, S.Beattie, A.Grier, P.Wagle, Q.Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *In Proceedings of the 7th USENIX Security Conference*, pp. 63–78, January 1998.

[10] IBM. Stack-Smashing Protector. http://www.trl.ibm.com/projects/security/ssp/.

[11] 古瀬淳, 米澤明憲. VITC: 対攻撃耐性コード生成コンパイラ. 日本ソフトウェア科学会第 22 回大会論文集, September 2005.