3A-3

動的リンク機構を利用したバイナリコードパッチ機能の設計

大津 金光 † 横田 隆史 † 馬場 敬信 †
† 宇都宮大学工学部情報工学科

1 はじめに

多様に変化する現代社会においてコンピュータシステムのソフトウェアプログラムにはその機能や性能についての様々な要求が増している．また，近年はコンピュータウィルス等の存在によりプログラムコードの安全性を脅かされている状況である．そのため，プログラムコードは機能面、性能面、安全面で常に改良され続ける必要があるが，利用者の手に渡り一度使われ始めたプログラムコード (一般には機械語バイナリコード形式であることが多い) の置き換えは容易ではなく，古いコードがいつまでも使い続けられる状況にある．

これを背景に，我々は既存のプログラムのバイナリコードを対象として実行時にバイナリコードの解析および変換を行なうことで，プログラム中に潜む危険なコードの改修や，プログラムの機能・性能の改良を実現する手法を検討している．この手法の実現にはバイナリコードの一部を置き換える手段が必要となるが，対象とするバイナリコードのプログラムファイルにコード書き換えによる恒久的な変更 (ダメージ) を加えることは避けるべきであると我々は考えた．そこで，本研究では実行時にバイナリコードに対してパッチ当て処理を行なう方法を採用する．本稿では，現代の OS が備える動的リンク機構を利用し，プログラム実行時にバイナリコードに対してパッチ当てを行なう機能 (以下，動的パッチ機能) について設計を行なう．

2 設計方針

本研究では対象バイナリコードのプログラムファイルに対して恒久的な変更を加えず，実行時にバイナリコードの一部をパッチ当てにより置き換えることを基本としている．これを実現する類似の手法としては既に *livepatch*[1] が開発されている．これはデバッガと類似の方法で他のプロセスから UNIX の *ptrace* システムコールを用いて対象プロセスのメモリ空間にアクセスし，コードの書き換えを行なうものである．そのため，対象プロセスとは別にコードの書き換えを行なうプロセスが必要になる．また，コードの書き換えを行う毎に *ptrace* システムコールを発行する必要があり，それに伴うオーバーヘッドが発生する．この方法には OS の時間的・空間的資源を消費するという点で問題があり，特にバイナリ変換による高速化を目的とする場合にはこのオーバヘッドの影響は小さくないと考えられる．そこで本研究では対象プロセス内でのコードの自己書き換えによりバイナリコードのパッチ当て機能を実現する．

また，実行時のバイナリコードのパッチ当て機能の実現のために OS カーネル内のコードに修正を加えることはカーネルのバージョンに依存する問題があり，また機能導入の障壁を高めることになるため，本研究では OS が提供する標準的な機能を利用し，ユーザレベルコードのみにより実現する．

3 機能設計

本稿では一般に広く普及している IA-32 プロセッサ [2] をターゲットとし，実装実験の容易さの観点から OS として Linux が動作している環境を前提に設計を行なう．

3.1 起動時期と方法

まずは、動的パッチ機能の起動時期と起動方法を決定する．ユーザプログラムの実行中にバイナリコードにパッチを当てる処理をいつどのようにして機能させるかについては様々な選択肢があるが，対象プログラムの実行開始に必要な共有ライブラリのリンク処理が全て完了した段階でパッチを当てられるようにするため，本研究ではユーザプログラムの実行開始地点である *_start* 関数†に実行がおよんだ段階でパッチ当て処理を行なうこととし，その処理の起動には OS が提供する動的リンク機構 [3] を利用することにした．

ユーザプログラムの *_start* 関数が実行される前に，実際はプログラムが要求する共有ライブラリをリンクする作業を行なうために動的リンカ (以後，*ld.so*) が同じプロセス空間内で実行されている．動的パッチ機能を共有ライブラリ (以後，動的パッチ機能を提供する共有ライブラリを *libdynpatch.so* と呼ぶ) の形で実装しておき，環境変数 *LD_PRELOAD* を指定することで *ld.so* によって強制的に *libdynpatch.so* を組み込ませる．

共有ライブラリには初期化処理を目的としてリンク処理完了後に (*_start* 関数実行前に) 実行するコードを含めることができる．この初期化処理コードの呼び出し機構を利用して動的パッチ当て処理を起動する．現在の IA-32 Linux 環境で一般的な *ELF* 形式 [4] のバイナリコードの場合は *.init* セクションや *.ctors* セクションを使うことで実現できる．

3.2 2 段階処理

LD_PRELOAD の指定によって *libdynpatch.so* が読み込まれ，その初期化処理コードが実行された段階では他の共有ライブラリの初期化処理が全て完了している保証はない．そこで，*libdynpatch.so* の初期化処理の段階では動的パッチ機能の本体処理を行なわず，*_start* 関数の先頭に動的パッチ機能の本体処理コードへジャンプさせるフックコードを埋め込む処理だけを行ない，後で *_start* 関数が実行された際に，埋め込まれたフックコードによって動的パッチ機能の本体処理が起動されるようにする．これによって，全ての共有ライブラ

Design of a Dynamic Binary Code Patching Function using Dynamic Linking Mechanism

†Kanemitsu Ootsu, Takashi Yokota, and Takanobu Baba
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)

† 実際のシンボル名はプラットフォームによって異なるが，本稿では総称として *_start* 関数と呼ぶことにする．

リのリンクと初期化が完了している状態で動的パッチ処理が行なえる．

3.3 実行開始位置の特定

_start 関数はソースコードやコンパイラによって配置アドレスが変わるため，*_start* 関数の先頭にフックコードを埋め込むためにはそのアドレスを特定する必要がある．これには，*_start* 関数がユーザプログラムの実行開始アドレスに配置されることを利用する．プログラムの実行開始アドレスは実行可能バイナリファイルのヘッダ情報から取得できるが，このヘッダ情報はプログラムコードのメモリへのロード時に共に読み込まれているため，ヘッダ情報が読み込まれているメモリ領域を調べることで実行開始アドレスを特定する．

3.4 フックコードの埋め込み

_start 関数の先頭にフックコード (動的パッチ処理コードへのジャンプ命令) を埋め込むための領域が必要であるが，バイナリコードは既に配置アドレスが確定しており簡単に移動することができない．そのために *_start* 関数の先頭のいくつかの命令を上書きすることでコードを埋め込む．その際，元のプログラムの動作を変えないようにするため，上書き前の命令コードを保存しておき，動的パッチ処理完了後再び *_start* 関数に制御を戻す直前に実行するよう動的にコードを生成する．

3.5 コード領域の書き換え

現在の OS はコード領域を読み出し専用に設定しており，ユーザプログラムが勝手に書き換えることを禁じている．そのためユーザプログラムがコード領域に対して強引に書き込みを行なうと保護違反を起こしプログラムの実行が終了する．この問題は OS が提供する *mprotect* システムコールを用いることで解決できる．本システムコールによって指定したメモリ領域の保護属性を変更することができるため，対象コード領域に対し書き込み属性を付与することでコード書き換えが可能になる．

3.6 パッチ処理本体の呼び出し

_start 関数の先頭のフックコードにより動的パッチ処理が起動する際，後で再び *_start* 関数に制御を戻す時に備えてレジスタの値を保存しておく必要がある．特に，スタック上にはプログラム起動時のコマンドライン引数や環境変数情報が置かれておりスタックポインタの値が書き変わると正しい実行ができなくなる．

3.7 自己書き換えに伴なう一貫性問題

対象プロセッサのアーキテクチャに依存するが，命令コードを書き換える際，書き換えられる命令が命令キャッシュ中に存在する場合は書き換えの効果が反映されない場合がある．また，演算パイプラインの存在により，直前に行なった命令書き換えの影響が反映されない期間が存在する．そのため，プログラムが意図しない動作を起こすことがある．この問題を避けるためには書き換えた命令コードの一貫性を保持するための処理が必要であり，命令キャッシュの無効化やパイプラインのフラッシュ処理を行なう必要がある．ただし，本稿で対象としている IA-32 プロセッサではハードウェアによって自動的に一貫性が保持されるので明示的な処理は不要である．

4 実装実験

本稿で設計した動的パッチ機能を実装し，以下の 3 つの環境で実行した．括弧内の 32bit/64bit の表記は，カーネルについて表わしたものであり，実験用のプログラムや *libdynpatch.so* 自体は 32bit モードで動作する．また NX-bit とは最近の IA-32 プロセッサに装備されている保護機能の一つで，これによってデータ領域中に配置された命令コードの実行を禁止できる．本動的パッチ機能では一部の命令コードをデータ領域中に動的に生成して実行するため，この機能の影響を受ける可能性がある．

1. Intel Pentium 4 3.2GHz, Vine Linux 3.2 (32bit, NX-bit **なし**)
2. Intel Pentium 4 3.2GHz, Fedora Core 5 (32bit, NX-bit **あり**)
3. AMD Athlon64X2 4800+, CentOS 4.4 (64bit, NX-bit **あり**)

上記のすべての環境において実行時にプログラムファイル自体を改変することなくバイナリコードにパッチを当てることができ，元となるバイナリコードには無かった機能を追加できることを確認した．また，NX-bit を使用していても，*mprotect* システムコールによりパッチコードの領域に対して実行許可属性を付与することで保護違反を起こすことなく実行が可能であることを確認した．

5 おわりに

IA-32 プロセッサ上で Linux OS が動作している環境をターゲットとし，実行時にバイナリコード対してパッチを当てる機能の設計を行ない，実装を行なった．実験の結果，設計通りに動作することを確認できた．本機能は現在の UNIX 系 OS には標準的に備える機能や仕様のみを利用して実現されているため，同様の機能が提供されていれば他の OS でも機能の実現が可能であると考えられる．

今後は，今回実装を行なったパッチ当て機能を汎用的に利用できるようにソフトウェアインターフェースを決定しフレームワークとして実現する予定である．また，Intel64 や AMD64 といった 64 ビット環境での実装も行なう予定である．さらに，非 UNIX 系 OS，例えば Windows 系 OS 上での本機能の実現も今後の課題である．

謝辞 本研究は，一部日本学術振興会科学研究費補助金（基盤研究 (B)18300014，同 (C)16500023，若手研究 (B)17700047）および宇都宮大学重点推進研究プロジェクトの援助による．

参考文献

[1] livepatch - Live Patching for Linux, http://ukai.jp/Software/livepatch/

[2] **インテル**, “IA-32 **インテルアーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル**,” 2001.

[3] John R. Levine, “Linkers & Loaders,” Morgan Kaufmann Publisher, 2000.

[4] TIS Committee, “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2,” 1995.