メモリ効率を考慮した動的アップデート手法の提案と実装
Dynamic Software Update with Efficient Memory Usage

山田司 †, 福田浩章 †
Tsukasa Yamada†, and Hiroaki Fukuda†
† 芝浦工業大学　工学部
†Faculty of Engineering, Shibaura Institute Technology

要旨

脆弱性対策として, ソフトウェアは定期的にアップデートを行い, 最新の状態にしておかなければならない. 通常, ソフトウェアのアップデートでは再起動を必要とし, 実行停止から再起動までアップデート対象となるプログラムの動作は停止する. これはソフトウェアの運用において可用性を損なわせる. この問題は, プログラムの実行中にアップデートを行う, 動的アップデートにより改善することができる. しかし, 一般的な動的アップデート手法では新しい関数を書き込むための heap 領域をアップデートする度に毎回確保するため, 古い関数がメモリ中に残され, 非効率的にメモリを使用してしまう.

そこで本研究では, 古い関数が存在する領域を再利用する動的アップデート手法を提案する. 提案手法では更新対象プロセス中の関数名やそのサイズ等の情報をファイルに記録し, このファイル内容を基にアップデートを行う. そして, 提案手法を利用して新しい関数のサイズを変化させ, 各サイズでの更新対象プロセスの停止時間を測定した. その結果から, 新しい関数のサイズと停止時間の関係を確認した. また, アップデートが正常に動作することも確認した.

1. はじめに

ソフトウェアにおいて, プログラムの設計上のミスが原因となって発生する欠陥を脆弱性と呼ぶ. この脆弱性は外部からの不正アクセスやウイルス感染を引き起こす原因となる. そのためソフトウェアは定期的にアップデートする必要がある. ソフトウェアをアップデートする場合は, 実行しているプログラムを中断し, プログラムを修正後に再起動する必要がある. この中断と再起動の間の停止時間はソフトウェアの運用において可用性を損なわせる. また, 中断すると実行中のデータを失うため, 再起動した時には初期状態からプログラムが実行されることになる. これは 24 時間連続でサービスを提供しなければならないサーバで望ましくない.

プログラムを実行したまま変更する手法として動的アップデートがある. これによりアップデート前後でのデータの消失を防ぎ, 停止時間の削減により可用性の高いシステムを実現することができる.

一般的な動的アップデート手法では新しい関数を heap 領域に書き込み, 古い関数が存在する領域は使わなくなる. つまり, 既存の動的アップデートを繰り返すとメモリ中に使わない領域が増加し, 非効率的にメモリを使用してしまう. そこで本研究では, 使わない領域を再利用する動的アップデート手法を提案する.

2. 関連研究

本節では, 動的アップデートに関連した研究について述べる. なお, 本稿では, 動的アップデート処理を行うプロセスを制御プロセスと呼ぶ.

2.1.livepatch

Linux kernel ではあるプロセスから他のプロセスの監視, 制御, 変更を行う手段として, ptrace システムコールが提供されている. livepatch[1] ではこれを使い, 更新対象プロセスを一時停止させている間にプロセスの内容を書き換えることで動的アップデートを行っている. livepatch によるアップデートは主に, ptrace システムコールによるアタッチ処理, メモリ確保処理, パッチ書き込み処理, jmp 命令書き込み処理, デタッチ処理の 5 つの処理から構成される. 図 1 に関数 func を関数 new_func に動的アップデートした場合の関数の呼び出し関係を示す. まず制御プロセスは ptrace システムコールを使い, 自プロセスに動的メモリ確保を行う mmap2 システムコールを用いた命令コードを更新対象プロセスに実行させ, heap 領域を確保する (図 1 の (1)). 次に確保した領域に関数 new_func を書き込む (図 1 の (2)). 最後に関数 func の先頭を関数 new_func の先頭への jmp 命令に書き換える (図 1 の (3)).

2.2.pannus

pannus[2] では, mmap2 システムコールを改良して特定のプロセスのメモリにアクセスできるシステムコールを追加実装し, 制御プロセスから更新対象プロセスの heap 領域を直接確保している. これにより, livepatch よりも短い停止時間でメモリ確保を行っている.

2.3. 関連研究の問題点

livepatch, pannus いずれも 2 回目以降のアップデートでも 1 回目と同様に, heap 領域を確保し, そこへ新しい関数を書き込む. つまり, アップデートを繰り返すと, メモリ中に占める古くなった関数が存在する領域の割合が増加し, 非効率的にメモリを使用してしまう問題がある.

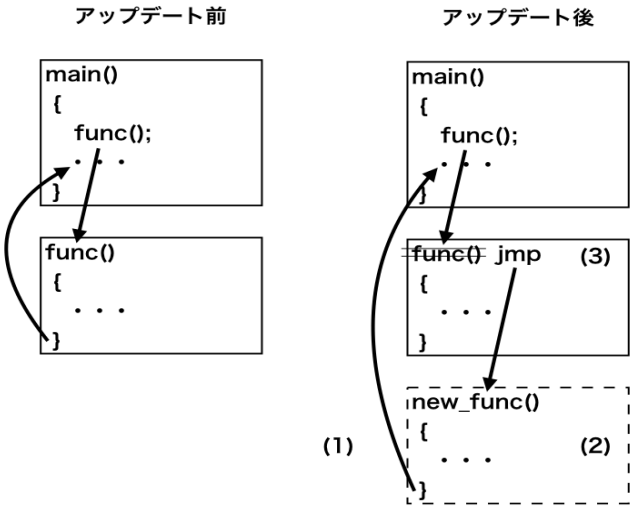

図 1: livepatch による動的アップデート

3. アプローチ

本研究では, livepatch にメモリ管理機能を追加し, 1 回目のアップデートで不要となる関数が存在する領域を 2 回目以降のアップデートで再利用する. なお, 本節では, 関数 func1 と func2 を備えるプログラムを例に図 2 と図 3 を用いて提案手法を述べる.

3.1. 処理手順

古い関数が存在する領域を再利用するには更新対象プロセスの初期状態を知る必要がある. まず制御プロセスは更新対象プログラムのシンボルテーブルを調べることで, 全ての関数のアドレスやサイズ等の情報を取得し, ファイルとして保持する. 以降, このファイルをマップファイルと呼ぶ. 図 2 に提案手法でアップデートした場合のメモリ空間の変化を示す. 1 回目で関数 func1 を関数 new_func1 にアップデートする場合, 更新対象プロセス中に関数 new_func1 のサイズ以上の heap 領域を確保し, そこへ関数 new_func1 を書き込む. この時点で関数 func1 が存在する領域は不要となる. 不要となった領域のアドレスやサイズは空き領域情報としてマップファイルに反映する. 2 回目で関数 func2 を関数 new_func2 にアップデートする場合, マップファイルを参照し, 関数 new_func2 のサイズ以上の空き領域が存在すればそこへ関数 new_func2 を書き込む. 存在しなければ, heap 領域に書き込む.

3.2. マップファイル

提案手法では, 更新対象プロセス 1 つに対して, symbol_map と free_map の 2 種類のマップファイルをアップデート前に 1 度生成し, 以降アップデートの度にこれらのファイルを更新する. symbol_map には関数名とその関数のメモリ空間における始点, 終点アドレスを書き込み, free_map にはアップデートにより空き領域となった領域の始点, 終点アドレスを書き込む. 図 3 にアップデートによるマップファイルの変化を示す. まず, アップデート前に 2 つのマップファイルを生成し, symbol_map には関数名 func1, func2 と, それぞれの関数の始点, 終点アドレス a1, a2, a3, a4 を用いて書き込む (図 3 の (1)). この時, 対

象のメモリ空間内に空き領域は存在しないため free_map に書き込む情報はない. 1 回目のアップデートでは, symbol_map 内の関数 func1 の名前と始点, 終点アドレスを関数 new_func1 のものに上書きし (図 3 の (2)), free_map には jmp 命令の終点アドレス a5 の次のアドレスからアドレス a2 までを空き領域情報として書き込む (図 3 の (3)). 2 回目のアップデートでは, symbol_map に関数 func2 に関して 1 回目と同様な上書き (図 3 の (4), (6)) をした後, free_map に関数 new_func2 の終点アドレス a8 の次のアドレスからアドレス a2 までを空き領域情報として上書きする (図 3 の (5)).

3.3. メモリの解放

提案手法では, livepatch にある 5 つの処理にメモリ解放処理を追加し, mmap2 システムコールで確保した領域が不要となれば, その領域を解放する. 解放には, 指定範囲のマッピングを除去する munmap システムコールを使い, 更新対象プロセスに mmap2 システムコールを実行させる時と同様に, munmap システムコールを実行させることで解放する.

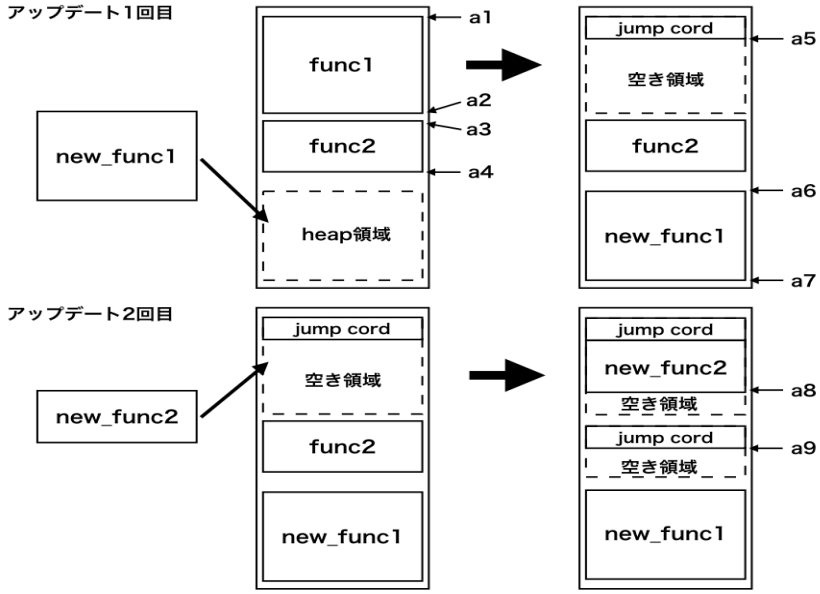

図 2: メモリ空間の変化

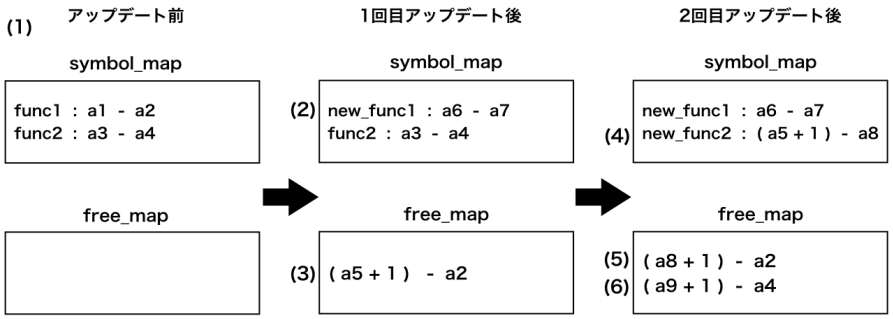

図 3: マップファイルの変化

4. 評価

停止時間の測定のために, パッチのサイズを 10KB から 100KB まで変化させ, 100 回動的アップデートを行った. その際に得られた停止時間の中央 80 個の平均を算出した. 測定箇所は, アタッチ処理 (attach), メモリ確保処理 (mem alloc), パッチ書き込み処理 (write patch), jmp 命令書き込み処理 (write jump), メモリ解放処理 (mem free), デタッチ処理 (detach) の 6 つに分け, 測定結果を図 4 に示す. 6 つの処理の中で処理時間がパッチのサイズに依存するのはパッチ書き込み処理のみであることがわかる. パッチ書

き込み時間の変化を図 5 に示す. 図 5 より, パッチ書き込みによる停止時間はパッチのサイズに比例していることが確認できた. また, アップデート処理が正常に動作していることも確認した. 今後は, 提案手法を用いた場合のメモリ効率を既存手法と比較する予定である.

patch size(KB)	time(ms)						
	total	attach	mem alloc	write patch	write jump	mem free	detach
10	1.409	0.017	0.037	1.314	0.005	0.026	0.007
20	2.558	0.016	0.034	2.468	0.005	0.027	0.006
30	3.779	0.017	0.036	3.681	0.005	0.032	0.006
40	5.068	0.017	0.039	4.958	0.005	0.039	0.007
50	6.411	0.021	0.038	6.298	0.005	0.039	0.007
60	7.621	0.017	0.038	7.511	0.005	0.040	0.007
70	8.927	0.017	0.038	8.818	0.005	0.039	0.007
80	10.065	0.017	0.035	9.960	0.005	0.039	0.007
90	11.798	0.017	0.044	11.679	0.006	0.042	0.007
100	12.519	0.017	0.044	12.396	0.006	0.046	0.007

図 4: パッチのサイズによる各停止時間の変化

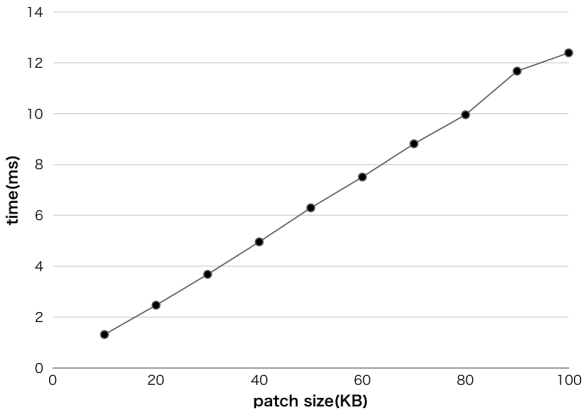

図 5: パッチ書き込み時間の変化

5. まとめ

一般的な動的アップデート手法では新しい関数を書き込むための heap 領域をアップデートの度に毎回確保するため, 古い関数がメモリ中に残され, 非効率的にメモリを使用してしまう. そこで本研究では, メモリ効率を向上させる動的アップデート手法を提案した. 提案手法は, 繰り返されるアップデートに伴って増加する古い関数が存在する領域を再利用することにより実現した. これにより, アップデートの度に毎回新しく領域を拡張する必要がなくなる. 現状の評価として, パッチのサイズと停止時間の関係を明らかにした. 今後はメモリ効率の測定を行う予定である.

参考文献

[1] http://ukai.jp/Software/livepatch/
[2] https://osdn.co.jp/event/kernel2004/pdf/C02.pdf
[3] Yamato, K., Abe, T. and Corpration, M.: A runtime code modification method for application programs, *Proceedings of the Ottawa Linux Symposium* (2009)
[4] 小澤駿, 齋藤彰一, 「プログラム領域の copy on write を抑制した複数プロセスの動的更新手法の提案」 (2014)