C-008

ハードウェア暗号モジュールによる BOA 防止策の提案

An Approach to the Prevention of Buffer-Overflow Attacks by a Hardware Encryption Module

李 碧波†　上原 稔†　森 秀樹†

Bibo Li　Minoru Uehara　Hideki Mori

1．はじめに

近年、バッファオーバーフローと呼ばれる脆弱性を利用した攻撃（BOA）は、非常に深刻かつ持続的な問題になってきている。既存のバッファオーバーフロー対策の多くは、ソフトウェア方法によって実現されたことから、性能に与える影響が大きい。

本研究では、CPU アーキテクチャに暗号化モジュールを追加することによって、リターンアドレスの完全性を保護するハードウェアレベルの解決手法を提案する。また、SimpleScalar ツールセット[2]を用いて、本手法の実装を行い、提案手法適用による機能及び性能への影響を検証する。

2．関連研究

2.1 バッファオーバーフロー攻撃

C や C++言語などのプログラミング言語で書かれたプログラムでは、プログラムが確保したメモリサイズを越えた文字列が入力されると、領域があふれて(オーバーフロー)しまい、予期しない動作が起きる。バッファオーバーフローの中で、最も危険なスタックオーバーフローが発生すると、リターンアドレスが書き換えられ、悪意のあるコードが実行可能になる。その結果、システム全体が乗っ取られてしまう可能性が発生する。

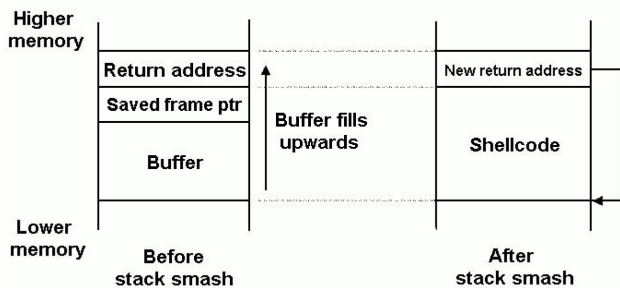

図１．stack smashing attack

2.2 解決手法

スタックオーバーフロー問題の解決手法はソフトウェアによる（静的、動的）手法と、ハードウェアによる手法の２つに分けることができる。

ハードウェアによる手法の利点

1. 性能への影響が小さい
2. 利用者に対して透過的であるため、利用者は攻撃に対する手法の適応を意識する必要がない
3. プログラムに変更を加える必要がないため、動作確認テストの必要がない

† 東洋大学大学院工学研究科情報システム専攻

† Department of Open Information Systems, Graduate School of Engineering

ハードウェアによる解決手法として、SRAS(Secure Return Address Stack)[1] はプログラムの実行中に手続きのリターンアドレスを保存するハードウェアリターンアドレススタックを用いてリターンアドレスの改ざんを防ぐものである。SRAS は限れたエントリ数のスタックであるため、保持されるリターンアドレス数がエントリ数を超えた場合、OS が割り込みをかけ、データをメモリに保存する。この OS による割り込みが SRAS のオーバーヘッドとなる。

3. 提案手法

本研究では、ハードウェアレベルで、リターンアドレスの完全性を保護する解決手法を提案する。手続きがリターンする際、リターンアドレスが書き換えられたかどうかを検証することによって、バッファオーバーフロー攻撃を検出し、攻撃を防ぐことができる。

3.1　方針

本手法では、スタックに２つのリターンアドレスを保存し、その中の１つを暗号化することによって、リターンする際の完全性を検証する。

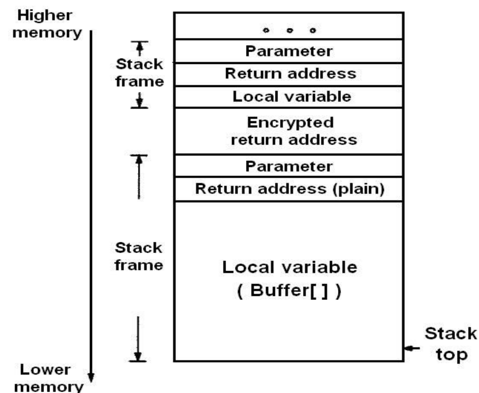

図 2．提案手法

手続きを呼出すと同時に、CPUは暗号化されたリターンアドレス（以下暗号文）をスタックにpushする。平文リターンアドレス（以下平文）は呼出す時に、正常にスタックにpushされる。手続きがリターンする時、スタックから２つのリターンアドレスを取り出し、暗号文を復号化し、平文との比較を行う。（あるいは、平文を暗号化し、暗号文との比較を行う）同じの場合、リターンアドレスが有効であり、実行を続ける。異なる場合、少なくとも中の１つは書き換えられ、バッファオーバーフローが発生することになる。このとき、システムに割り込みを要求し、不正動作を回避することができる。

平文に関するスタック操作が従来通りに行われるため、今回の変更はスタックフレーム間に暗号文を挿入する動作だけである。

暗号化に使われる暗号鍵は、CPU 内部のレジスタに保存され、プログラムごとにランダムな数字である。バッファオーバーフローが発生すると、１つもしくは２つのリターンアドレスが書き換えられるかもしれない。ただ一つの攻撃手段として、２つのリターンアドレスがすべて書き換えられ、平文は攻撃コードのアドレス位置に、暗号文はそのアドレスの暗号形式に書き換えられる。その可能性は非常に低く、また、暗号鍵は乱数のため、リプレイアタックはできない。事実上、バッファオーバーフロー攻撃が不可能になる。

今回の提案では、１つのプログラムスタックに対して、１つ暗号鍵を生成し、プログラムの実行時に鍵の変更はできない。実行中のすべてのプログラムにとって、鍵は同じである。メモリに攻撃者が侵入された場合（可能性は非常に小さい）、平文と暗号文が見つけられても解読できないように、一方向暗号（ハッシュ関数）を使用する。

3.2 基本動作

PISA（Portable Instruction Set Architecture）[2]を用いて動作の流れを説明する。R31はリターンアドレスの保存に用いられる。R29はSP（スタックポインタ）である。

JAL、JALR 命令（**Call** 命令）は Jump すると同時に、暗号文を store する操作を行う。

JR 命令（**Return** 命令）は、return すると同時に、暗号文を load し、比較を行う。

すべての操作は ISA 下にパッケージされ、プログラムから見ると透過的である。

以上の Jump 命令に、新たな操作を追加した後、スタック操作は以下のようになる：

1． 制御は手続きに移行するとき、（例えば JALR 命令は）、暗号文をスタックに push する。この時、SP（スタックポインタ）の R29 は変わらない（状態 1 と称す）。（ローカル変数のない手続きは、スタックフレームを持たない）
2． 手続きに移行した後、手続きはスタックフレームを持つ場合、新たなスタックフレームが作られる。CPU は状態１の場合（R29 は変わってない状態）、R29－１(SP-4)操作を行う。この操作は、スタックフレームが作られる時、新たなスタックポインタを設置する動作を合併することができる。
3． 従来の操作通り、平文は手続きの中でスタックに push される。
4． 従来の操作通り、main 関数に return する前に、平文はスタックから pop され、R31 に保存する。
5． 制御は手続きから戻る時、（例えば JR 命令は）、暗号文は pop され、状態 1 でない場合、R29＋1 (SP+4)操作を行う。
6． リターンアドレスの比較、”=”?継続：中断

上記の流れの中、手順１，２，５，６は従来の手続き呼出しスタック操作に追加した動作である。そのうち、手順２はクロックサイクルを消費しないが、手順１，５，６は各自１クロックサイクルを余分に消費するため、プロセッサ性能への影響が大きい。オーバーヘッドを解消するため、最大限の並行性を考慮しながら、モジュールの実装を行う。

4. 実装

5 ステージのパイプラインを持つ RISC アーキテクチャの CPU では、EX ステージで、ALU、シフト Unit、乗除算 Unit、load store unit の 4 つの実行 Unit がある。Jump 命令は ALU で行い、ロード・ストア命令は load store unit で行われる。

アーキテクチャに暗号化モジュールを追加する。モジュールの中に、32 ビットの Key レジスタが含まれる。暗号化モジュールは、ID ステージでリターンアドレスを暗号化し、MEM ステージ（メモリのロード・ストア）で、暗号文リターンアドレスを復号化し、平文との比較を行う。

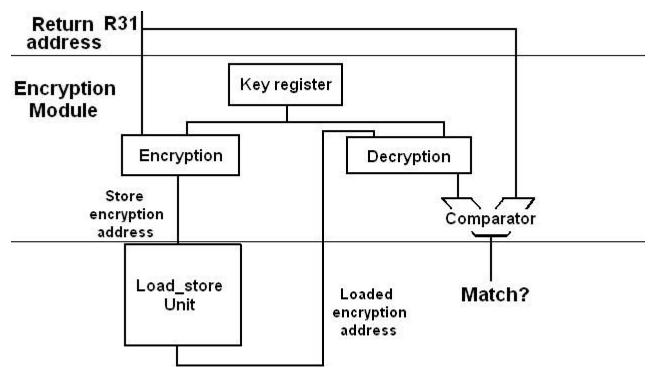

図 3. 暗号化モジュール

ID ステージで、暗号化モジュールは Call と Return 命令を見つけた時、ALU と load store unit にその命令を同時に発行する。ALU は制御の jump を実行し、load store unit は暗号文リターンアドレスの load store を行う（アドレスとデータ情報は暗号化モジュールから送られる）。

5. まとめと今後の課題

本研究で提案する手法がスタックオーバーフローに対して、有効であるかどうかを調べるために機能評価を行う。C の脆弱性を利用した stack smashing attack プログラムを動作させ、提案手法を実装した場合、リターンアドレスの改ざんによる攻撃を防ぐことができることがわかった。

今後の課題として、提案手法による性能低下を検証するため、SPEC ベンチマーク[3]を用いて、実装時の性能への影響を計測する。

参考文献

[1] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks," in Proc. Int. Conf. on Information Technology: Research and Education (ITRE), pp. 243--250, Aug. 2003

[2] T.Austin.Simplescalar.http://www.simplescalar.com

[3] T. Austin. SimpleScalar benchmarks. http://www.simplescalar.com/benchmarks.html