データ改ざん検出による侵入検知システムの一考察

長野 文昭 *　　鑪 講平 *　　田端 利宏 †　　櫻井 幸一 ‡

* 九州大学大学院システム情報科学府
812-8581 福岡市東区箱崎 6-10-1
{nagano,tatara}@itslab.csce.kyushu-u.ac.jp

† 岡山大学大学院自然科学研究科
700-8530 岡山市津島中 3-1-1
tabata@it.okayama-u.ac.jp

‡ 九州大学大学院システム情報科学研究院
812-8581 福岡市東区箱崎 6-10-1
sakurai@csce.kyushu-u.ac.jp

あらまし　近年，メモリ上のデータを不正に改ざんする攻撃が，セキュリティ上最も脅威のある攻撃の一つとなっている．データ改ざん検知手法はこれまでにもさまざまな手法が提案されているが，多くの検知手法はメモリ上のデータの改ざんの一部分しか検知することができないという問題点が存在する．また，既存のデータ改ざん検知手法に秘密データを用いているものがあるが，秘密データを用いる方式では，実行コードにフォーマットストリングバグといったメモリ上のデータを読み取られる脆弱性が存在すると，秘密データを攻撃者に推測されて，データを不正に改ざんされてしまう恐れがある．そこで，本論文ではメモリ上の任意のデータの攻撃者による改ざんを検知するための手法を提案する．また，本提案方式は，秘密データを利用した検知手法に存在する，メモリ上のデータを読み取られると秘密データを推測されるといった脆弱性は存在しない．

キーワード　侵入検知，データ改ざん，バッファオーバフロー，フォーマットストリングバグ

A Note of Intrusion Detection using Alteration of Data

Fumiaki NAGANO*　　Kohei TATARA*　　Toshihiro TABATA†　　Kouichi SAKURAI†

* Graduate School of Information Science and Electrical Engineering, Kyushu University
6-10-1 Hakozaki, Higashiku Fukuoka, 812-8581
{nagano,tatara}@itslab.csce.kyushu-u.ac.jp

†Graduate School of Natural Science and Technology, Okayama University 3-1-1 Tsushima-naka, Okayama, 700-8530
tabata@it.okayama-u.ac.jp

‡Faculty of Information Science and Electrical Engineering, Kyushu University
6-10-1 Hakozaki, Higashiku Fukuoka, 812-8581
sakurai@csce.kyushu-u.ac.jp

Abstract　These days, Attacks which alter data in memory illegally are one of the most serious security threats. Although a lot of detection systems have been proposed so far, most of the systems have the problem that only a part of the alteration of data in memory can be detected. And, some detection systems use secret data. But, if an execution code has a bug like format string bug which enable attackers to read data in memory, data in memory might be altered illegally because the secrete data might be guessed by the attackers. Then, we propose a system which detects the alteration of arbitrary data in memory by attackers. Moreover, this system doesn't have the vulnerability that exist the systems which use secret data.

keyword　Intrusion Detection, Alteration of Data, Buffer Overflow, Format String Bug

1. はじめに

近年，実行コードで使用されるデータを改ざんすることにより侵入を行う攻撃が多数報告されている．その最たる例が，バッファオーバフローの脆弱性を利用した攻撃である．シマンテックインターネットセキュリティ脅威レポートによると，2004年7月から12月の間にシマンテックにより検知された攻撃のなかで，バッファオーバフローを利用した攻撃の数の多さは3位となっている[1]．バッファオーバフローの脆弱性を利用すると，実行コード内の関数の戻りアドレスやポインタ変数を書き換えることで悪意のある攻撃者が任意のコードを実行することが可能となりうる．

バッファオーバフローの脆弱性を利用した攻撃以外にも，フォーマットストリングバグ[2]を利用した攻撃や，Integer Overflow[3]を利用した攻撃など，データをユーザの意図しない値に改ざんすることにより侵入を行う攻撃は多数存在する．また，最近はヒープオーバフロー[4]といった巧妙化したバッファオーバフローを利用した攻撃を利用した侵入も行われている．

よって，データ改ざんを検知する検知手法が必須である．

2. 関連研究

この節では，関連研究について述べる．

境界チェック

バッファの境界をチェックすることでバッファオーバフローの発生を検知し，データ改ざんを防ぐ方式がある[5]．しかし，この方式では，フォーマットストリングバグを利用したデータ改ざんを検知することができないことが報告されている[6]．

ヒープやスタック領域でのコードの実行禁止

メモリ上のある領域のコードを実行不可能にすることでデータ改ざんを利用した，悪意のあるコードの実行を不可能にしようという方式がある．既存の方式として，スタック上のコードを実行不可能にするという方式とヒープ上のコードを実行不可能にするという方式がある[7]．しかし，この方式はフォーマットストリングバグを利用したデータ改ざんなど検知しきれないものが存在するという点や，再帰的なプログラムを実行する場合には適用できないという欠点が報告されている[6]．

変数保護

変数を保護することで，データ改ざんを検知し，攻撃者の意図する攻撃を防ぐ方式がある．StackGuard[8]，StackFences[9]，propolice[10]はカナリアという秘密データを使用することで，Libsafe[11]はライブラリを変更することで，PointGuard[12]はポインタを暗号化することで，それぞれ変数を保護し，データ改ざんを検知している．しかし，これらの方式を使用することで保護される変数以外にも，攻撃に利用される変数が存在する事があるため，これらの方法を攻撃者により回避され，攻撃者の意図する攻撃が可能になる恐れがある．また，カナリアを用いた方法では，フォーマットストリングバグなどの脆弱性を利用して，メモリ上に存在するデータを読み取られると，データ改ざんが可能となり，攻撃者の意図する攻撃が可能になる恐れがある．これは，メモリ上から読み取ったデータを用いてカナリアの値を解読することが可能となるからである[13]．他にも，ポインタ変数を使用するとカナリアを解読することなくデータ改ざんが可能となり，攻撃者の意図する攻撃が可能になる恐れがある[13]．

ランダムなアドレス空間

実行コードに使用されるアドレス空間をランダムにすることで，攻撃者に意図したアドレスをわからせないようにし，攻撃者の意図する変数の改ざんを防ぎ，それにより攻撃の発生を防いでいる方式がある[14]．しかし，多くの場合でアドレスをランダム化してもあまり効果がないことが報告されている[15]．

上記のように．データ改ざん検知手法はこれまでにもさまざまな手法が提案されているが，多くの検知手法はメモリ上のデータの改ざんの一部分しか検知することができないという問題点が存在する．また，既存のデータ改ざん検知手法に秘密データを用いているものがあるが，秘密データを用いる方式では，実行コードにフォーマットストリングバグといったメモリ上のデータを読み取られる脆弱性が存在すると，秘密データを攻撃者に推測されて，データを不正に改ざんされてしまう恐れがある[16]．そこで，本論文ではメモリ上の任意のデータの攻撃者による改ざんを検知するための手法を提案する．また，本提案方式は，秘密データを利用した検知手法に存在する，メモリ上のデータを読み取られると秘密データを推測されるといった脆弱性は存在しない．

3. 提案手法

侵入行為に使用される脆弱なデータに対して，データが改ざんされる可能性がある前に検証子を作成し，侵入行為に使用される脆弱なデータが攻撃者により改ざんされたかどうかを調べる際に，作成した検証子を検証することで，データ改ざんの検知を行う．侵入行為の際に使用される脆弱な変数としては，戻りアドレス，関数ポインタ，関数の引数などがある[17]．以下，まずこの方式

において使用する検証子の構造について説明し，その後検証子を利用した侵入検知手法の方法について説明する．また，本方式では，カーネルメモリは攻撃者によりアクセスすることができないと仮定する．

3.1 検 証 子

侵入検知に使用する検証子のデータ構造として，図1のような検証子構造体を持つデータ構造を利用する[18]．

検証子構造体は，カーネルメモリに K 個存在する．また，ユーザメモリには検証子構造体が配列になった，検証子構造体配列と，検証子構造体がリストになった検証子構造体リストが存在する．ユーザメモリの検証子構造体配列の要素数は $N \times K$ 個である．ただし，N は，配列の要素の数であり，小さな値であるとする．N が小さな値である理由については3.2.3で述べる．まず，検証子構造体のそれぞれの要素について説明する．

検証子構造体の要素は，図2のようになっており，カーネルメモリ上のものと，ユーザメモリ上のものとで異なる．ユーザメモリ上のものも，配列のものとリストのものとで異なる．以下は，それぞれの要素の説明である．

ユーザメモリ上の検証子構造体の要素

- 検証対象ポインタ

被検証データの先頭のアドレスを格納．

- 検証対象データ長

被検証データの長さを格納．

この要素と，検証対象ポインタとで，被検証データを一意に識別できる．

- 検証データ

被検証データを検証するのに使用する値を格納．詳しくは3.2.2で述べる．

- 変更フラグ

被検証データの改ざんが検知されたときにセットされるフラグを格納．

- 制御フラグ

検証子構造体配列の要素にのみ存在するもので，この検証子構造体が使用中かそうでないかを区別するフラグを格納．

- 構造体ポインタ

検証子構造体リストの要素にのみ存在するもので，検証子構造体リストの要素のアドレスを格納する．

カーネル上の検証子構造体の要素

- 総検証データ

ユーザメモリの検証子構造体配列を検証するのに使用する値を格納．詳しくは3.2.2で述べる．

- 変更フラグ

被検証データの改ざんが検知されたときにセットされるフラグを格納．

- 検証対象ポインタ，検証対象データ長，検証データ，制御フラグ

これらは，ユーザメモリに存在する検証子構造体配列のキャッシュである．詳しくは3.2.4で述べる．

3.2 検 証 方 法

次に，実際にどのようなアルゴリズムを用いることで，メモリ上のデータが攻撃者により改ざんせれていないかをチェックする方法を述べる．説明の都合上，検証データのビット長を c とおく．

3.2.1 初 期 化

最初に初期化を行う．すなわち，カーネルメモリ上，ユーザメモリ上に図1のようなデータ構造を作成する．ただし，検証子構造体リストの要素は作成しない．また，初期化の際，カーネルメモリ，ユーザメモリ上に存在する制御フラグは，その検証子構造体が使用されていないことを示すように初期化する．

3.2.2 検証子登録方法

被検証データについて，バッファオーバフロー，フォーマットストリングバグを利用した攻撃などの侵入行為が成功する前に，ユーザメモリの検証子構造体を作成する．検証子構造体配列と検証子構造体リストのどちらの要素を作成するのかについては3.2.7で述べる．ここでは，共通の要素について述べる．

検証子を作成する際に，侵入行為から保護したいデータについて全ての検証子構造体を初期化時に作成してしまうという方法も考えうるが，その方法であると，プログラム実行中に作成された変数，たとえばスタック上に詰まれる戻りアドレスや，ヒープに新たに作成された変数に対して，攻撃者による改ざんをを検知することができない．また，プログラムの一部の間でしか使わない変数を最初から保護し続けることはメモリの有効利用の点から芳しくない．よって，動的に検証子構造体を追加できる必要がある．よってここでは，その方法について述べる．

はじめに，検証データの値について述べる．検証データの値は，被検証データの長さが c より長かった場合は，被検証データを長さ c の区分に分割し，それぞれをXORしたものが検証データである(図3参照)．また，被検証データの長さが c より短かった場合は，適当なパディングを加えた値が検証データとなる．

その後，作成した検証データを含む検証子構造体を，ユーザメモリの検証子構造体のいずれかに格納する．以下，その格納方法について述べる．以降，検証子構造体リストについては考えないものとする．検証子構造体リ

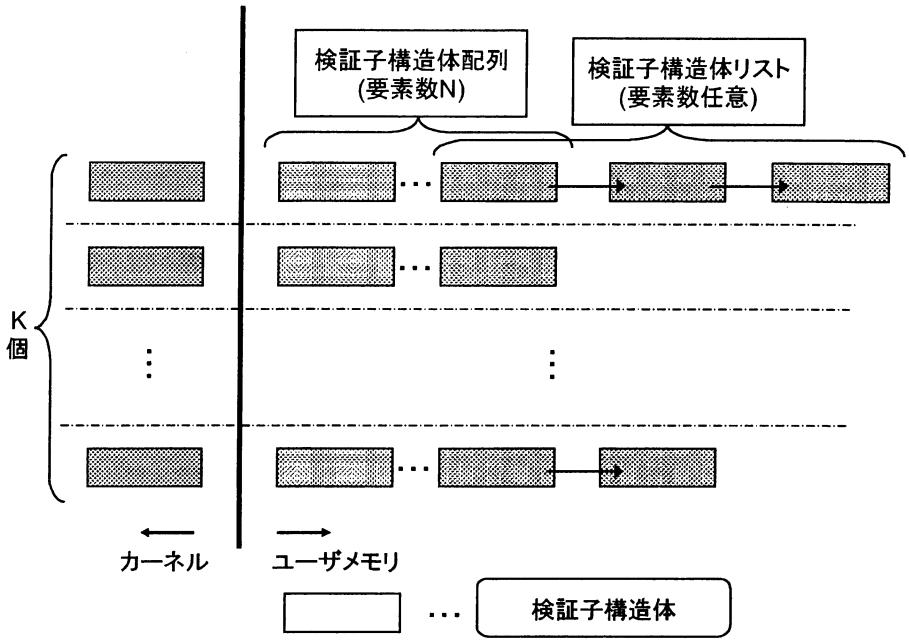

図 1　検証子のデータ構造

カーネルメモリ上の検証子構造体

総検証データ	検証対象ポインタ1	検証対象データ長1	検証データ1	制御フラグ1	検証対象ポインタ2	検証対象データ長2	検証データ2	制御フラグ2	…	変更フラグ

ユーザメモリ上の
検証子構造体配列の検証子構造体

検証対象ポインタ	検証対象データ長	検証データ	変更フラグ	制御フラグ

ユーザメモリ上の
検証子構造体リストの検証子構造体

検証対象ポインタ	検証対象データ長	検証データ	変更フラグ	制御フラグ	構造体ポインタ

図 2　検証子構造体

ストについては，3.2.7 で述べる．

まず，K 個あるうちの，どの検証子構造体配列に登録するかを述べる．どの検証子構造体配列に登録するかのアルゴリズムは，チェイン法によるハッシュ探索法を使用する．この探索法は，挿入，探索，削除ともすばやく行う事ができるという特徴をもつ．具体的に本方式でチェイン法によるハッシュ探索法をどのように使用するかについては，検証対象ポインタの値を 2 で割り，その商を検証子構造体配列の個数である K で割り，その余りによりどの検証子構造体配列を使用するかを決める．たとえば，

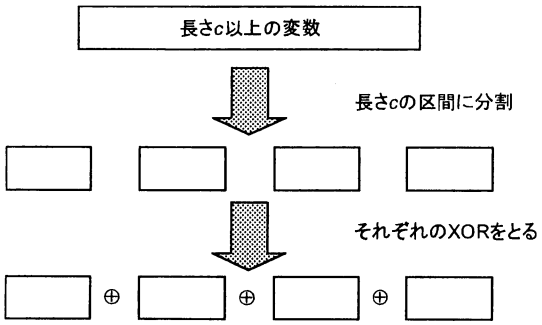

図 3　検証データの値の作成方法

検証対象ポインタの値を 2 で割り，その後，K で割った余りが 1 になった場合，図 1 の上から 1 番目の検証子構造体配列を使用するといった具合である．ただし，検証対象ポインタの値を 2 で割った商が整数にならなかった場合は四捨五入を行うとする．

なぜ，K で割った余りを利用するかは，K で割った余りは，0 から $K-1$ までの数しかないため，一意に，登録する検証子構造体配列を決めることができるためである．また，なぜ，2 で割るかについては，検証対象ポインタの値の偶数と奇数の偏りを抑え，極力，利用する検証子構造体配列をばらばらにするためである．

どの検証子構造体配列に登録するのかが決まったら，次にその検証子構造体配列のどの要素に登録するかについて決める必要がある．それには，検証子構造体配列の要素の制御フラグを参照する．使用されていない検証構造体配列の要素の制御フラグには，初期化時に使用されていないことを示すように初期化している．よって制御フラグを見ればまだ使用されていない検証子構造体配列の要素を探索することができる．よって，その要素に，先ほど作成した検証子構造体を格納する．そして，使用中の検証子構造体配列の要素であることを示すために，格納した検証子構造体の制御フラグを更新する．ここで，まだ使用されていない検証子構造体配列の要素を調べる際には配列の 0 番目の要素から N 番目の要素に向かって調べていくものとする．この理由は 3.2.3 で述べる．これで，ユーザメモリの検証子構造体配列への格納は終了である．

次に，カーネルメモリの検証子構造体について述べる．この構造体は，ユーザメモリの検証子構造体配列が攻撃者により変更されていないかを調べる目的と，検証の際のオーバヘッド削減のため，ユーザメモリの検証子構造体を保存しておくキャッシュの目的がある．キャッシュの目的のほうについては 3.2.4 にて述べる．ユーザメモリの検証子構造体配列が攻撃者により変更されていないかを調べる目的についてはカーネルメモリの検証子構造体は K 個，ユーザメモリの検証子構造体配列も K 個 存在するので，一つのカーネルメモリの検証子構造体につき，一つの検証子構造体配列の検証を行う．

カーネルメモリの総検証データはユーザメモリの検証子構造体配列の値を，図 3 で述べた要領で XOR したものである．これにより，ユーザメモリの検証子構造体配列の改ざんも検知することができる．つまり、たとえ検証子構造体配列を攻撃者に読み取られ、改ざんされたとして、その改ざんを検知することができる。検知手法については次節で述べる．

3.2.3　被検証データ検証方法

ここでは，被検証データの検証方法について述べる．

まず，被検証データの検証データを格納しているユーザメモリの検証子構造体配列の改ざんの有無を検知するために，被検証データを格納している検証子構造体配列の総検証データを図 3 で述べた要領で XOR し，それに対応しているカーネルの検証子構造体の総検証データとを比較を行う．今，仮定により，カーネルは攻撃者によりアクセスされないため，カーネルの総検証データの改ざんはない．よって，カーネルの総検証データとユーザメモリの検証子構造体配列を計算した結果の比較が一致すれば，ユーザメモリの検証子構造体配列が攻撃者により改ざんが加えられていないことが検証できる．もし，比較した値が等しくなかったら，変更フラグに変更が検知されたことを示すフラグを立てる．

改ざんが検知されなかった場合は，ユーザメモリの検証子構造体の検証データを用いて被検証データの検証を行う．具体的には，被検証データを図 3 で述べた要領で XOR し，改ざんされていないことが確証されている検証子構造体に格納されている検証子とを用いて比較を行う．もし，両者が一致すれば，被検証データの改ざんがないことが確証される．もし，両者が一致しなければ，変更フラグ改ざんが検知されたことを表すフラグを立てる．

この際，被検証データがユーザメモリの検証子構造体配列のどの要素に格納されているかを探索する必要がある．探索する方法は検証子構造体配列の N 番目の要素から，1 番目の要素に向かって探索を行う．なぜ，後ろから探索を行うかは，被検証データは，時間的に最後に検証子構造体を作成されたものほど，検証される可能性が高いからである．つまり，被検証データは first in first out の割合が高いからである．たとえば関数を考える．サブルーチンに飛ぶ前に使用されたいたローカル変数は，サブルーチンの内部では使用されることはないのである．検証子構造体配列に検証子構造体を登録するときには，

3.2.2 で述べたように，配列の 1 番目の要素から登録しているので，配列の後ろから探索を行うと，上記の特徴を利用した探索を行うことができる．ここで，N を余り大きな数にしておくと，配列の後ろから探索を行うと，配列の後ろのほうは使用されていない要素ばかりであり，効率が悪いことので，N は小さい数にする必要がある．

3.2.4 キャッシュ

ここでは，キャッシュについて述べる．3.2.3 で述べた，被検証データ検証方法には無駄がある．それは，たとえば，被検証データの検証データを格納している検証子構造体配列に，検証子構造体が多数格納されているときに生じる．3.2.3 で述べた方法では，一つのユーザメモリに存在する検証子構造体の改ざんの有無を検知したいときでも，その検証子構造体が格納されている検証子構造体配列の全ての要素の改ざんの有無を検知することになる．これは明らかな無駄である．よって，この問題を解決するために，キャッシュを用いる．

キャッシュは，ユーザメモリの検証子構造体のコピーをカーネルに登録したものである．これにより，被検証データを検証する際に被検証データの検証子構造体がカーネルに登録されていれば，ユーザメモリの検証子構造体配列の改ざんの有無を検知する必要はない．なぜなら仮定によりカーネルは安全であるからである．全てのユーザメモリの検証子構造体をカーネルに登録することも可能ではあるであろうが，カーネルメモリは通常貴重な資源であるため，全ての検証子構造体を登録することは芳しくない．

3.2.5 検証子構造体の再登録時のオーバヘッド

総検証子をカーネルに登録する際に，毎回総検証子に対応するユーザメモリの検証子構造体配列の XOR を計算し毎回カーネルにアクセスすると，オーバヘッドが大きくなってしまう．よって，オーバヘッドを削減するために，以下の XOR の特徴を利用する．

XOR 演算の特徴として，u，v，w をある数とすると，
$(u \text{ XOR } w) \text{ XOR } (v \text{ XOR } w) = (u \text{ XOR } v) \text{ XOR } w$
という式がなりたつ．

すなわち，新たに検証子構造体を作成する場合には，新たに作成した検証子構造体の検証データの値のみとすでにカーネルに存在している総検証データの値を XOR することで，ユーザメモリの検証子構造体配列の総検証データの値を求めることができる．これにより，毎回新たにユーザメモリの検証子構造体配列の XOR を再計算する必要はない．また，新たに作成した検証子構造体の検証データの値とすでにカーネルに存在している総検証データの値とを XOR した値が，カーネルに存在している総検証データの値とすでに同じであったら，カーネルの総検証データの値を更新する必要はない．すなわち，新たに作成した検証子構造体の検証データの値が 0 であったら，カーネルの総検証データの更新を行う必要はない．たとえば，値が 1010 で，新たに作成した検証データが 0 であったら，新たな総検証データは

1010 XOR 0 = 1010

となり，カーネルの総検証データを更新する必要はない．すなわち，カーネルにアクセスする必要がない．これらにより，オーバヘッドを削減することができる．

3.2.6 検証子構造体削除方法

検証子構造体の削除の方法についてのべる．すでに使わなくなったデータを検証する検証子構造体を検証子構造体配列に登録しておくことはメモリの有効利用上よくない．よって，使わなくなった検証子構造体に対しては削除することが望ましい．よって，使用されなくなった場合は削除を行う．検証子構造体配列からの削除を行うには，検証子構造体の制御フラグを，使用していないように示すように変更すればよい．しかし，これだけでは十分ではない．なぜなら，総検証データの更新が必要であるからである．総検証データの更新には，以下の XOR の特徴を用いる．

XOR 演算の特徴として，u，v，をある数とすると，
$(u \text{ XOR } v) \text{ XOR } v = u$
という式が成り立つ．すなわち，ある数に，同じ数を 2 回 XOR するともとの数に戻るという性質がある．よって，使われなくなった検証子構造体を削除する場合，総検証データを更新するには，削除する検証データを総検証データに XOR することで，総検証データを更新することができる．

3.2.7 検証子構造体配列の要素数

検証子構造体配列の要素数は，初期化時に作成されている．検証子構造体配列の要素数が，初期化時に作成されるということは，その要素数を超える検証子構造体の個数を登録できないということである．これは不都合である．よって，検証子構造体配列の要素数を超える検証子構造体を登録したい場合は，ヒープ領域を利用する．ヒープ領域はプログラムの実行中に動的に割り当てることができるメモリ領域であるので，無限の検証子構造体を登録することができる (もちろん物理的なメモリ領域による限界はある．)．この際，検証子構造体リストを利用する．

最初からヒープ領域を使用しない理由はオーバヘッドの削減のためである．表 1 以下は，実験結果である．(CPU Intel(R) Pentium(R) III 1GHz，メモリ 512MB の環境

で実験) 値は，ある被検証データに対して 1 回，検証子を作成し，かつ検証するのに必要な時間．単位は μ 秒であり，30 回計測した時間の平均を示している．

表 1　ヒープ領域使用，未使用によるオーバヘッドの違い

ヒープ使用	ヒープ未使用
0.81 μ s	0.67 μ s

表 1 からわかるように，ヒープ領域を使用しないほうがオーバヘッドを削減することができる．よって，最初からはヒープ領域を使用しないほうが望ましい．

検証子構造体リストの検証子登録方法，検証子削除方法は，検証子構造体配列と多少異なるので，ここで説明する．

はじめに，検証子構造体リストの検証子構造体登録方法を述べる．検証子構造体配列の全ての要素が使用中であったら，ヒープ領域に検証子構造体を登録する．その際，新しく登録した検証子構造体へのポインタの値を検証子構造体配列の N 番目の要素の構造体ポインタの要素に代入する．そして，以前検証子構造体配列の N 番目の要素の構造体ポインタの値を，新しく登録した検証子構造体の検証子ポインタの要素に代入する (図 4 参照)．

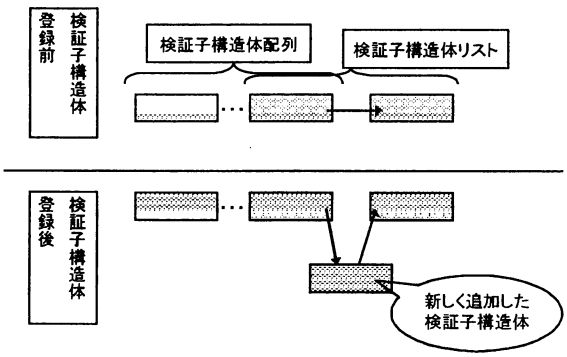

図 4　検証子構造体リストの要素の追加方法

これは，探索時に，被検証データは first in first out の割合が高いという特徴を用いるためである．3.2.3 で，検証子構造体を探索するとき，検証子構造体の N 番目の要素から探索を行うと述べた．この際，検証子構造体リストが存在すれば，まず，検証子構造体リストをたどって行って探索を行う．このとき，リストをたどっていくと被検証データは検証子構造体配列の最後の要素に近いほど，最近登録された検証子構造体になっているので，被検証データは first in first out の割合が高いという特徴をうまく利用することができる．

検証子構造体削除方法については，総検証子の更新の方法は，検証子構造体配列のときと変わらない．ただ，検証子構造体リストの要素を削除する方法は，削除する検証子構造体を指している構造体ポインタの値を，削除する検証子構造体の構造体ポインタの値にすればよい (図 5 参照)．

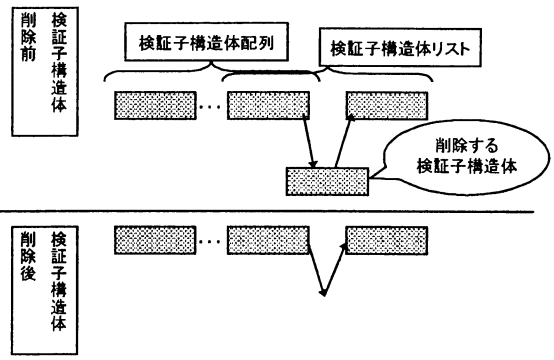

図 5　検証子構造体リストの要素の削除方法

3.3　検証結果利用方法

上記のアルゴリズムが改ざんを検知した時にはユーザアプリケーションに対してシグナルが送られる．そのときの処理はユーザアプリケーションのポリシ次第である．

4.　実　　装

4.1　実装方式

本方式を実装するには，カーネル領域に検証子構造体を作成するためにカーネル領域にアクセスすることと，プログラムの実行中に検証子を作成，検証することが必要である．よって，本方式を実装するには以下の 4 点が必要である．

(1)　カーネルに検証子構造体を登録するシステムコールの実装

(2)　カーネルに登録した検証子構造体を検証するシステムコールの実装

(3)　ユーザメモリに検証子構造体を作成する関数の実装

(4)　ユーザメモリの検証子構造体を検証し削除する関数の実装

(1)，(2) は OS に新たにシステムコールを追加することで可能となる．また，(3)，(4) はコンパイラを改良することで可能となる．また，これらの実装を行うことにより，プログラマが望む位置で任意のデータが攻撃者により改ざんされていないかを検証することが可能となる．なぜならプログラマが望む位置で検証子を作成，検証することができるためである．

4.2 実装結果

この節では，実装結果について説明する．3. 章でのべた方式に対して実装を行った．今回行った実装は，プログラマが任意の位置で任意の値の改ざんを検知するためシステムを実装した．結果，プログラマがデータ改ざん検知対象とした値に対する，フォーマットストリングバグを利用した攻撃，バッファオーバフローを利用した攻撃といった，ユーザの意図しない改ざんに対する攻撃を検知することができた．

また，オーバヘッドは表 2 のようになった．値は，32 ビットの被検証データに対して 1 回，検証子を作成し，かつ検証するのに必要な時間である．(CPU Intel(R) Pentium(R) III 1GHz，メモリ 512MB の環境で実験) 単位は μ 秒であり，30 回計測した時間の平均を示している．

表 2　オーバヘッド

システムコール呼び出しあり	システムコール呼び出しなし
0.67 μ s	0.05 μ s

表 2 からわかるように，システムコールを呼び出さない場合は，オーバヘッドはかなり小さく抑えられることがわかる．

5. まとめ

本論文では，既存のデータ改ざん検知システムの問題を解決するために，メモリ上の任意のデータの攻撃者による改ざんを検知するための手法を提案した．また，本提案方式は，秘密データを利用した検知手法に存在する，メモリ上のデータを読み取られると秘密データを推測されるといった脆弱性は存在しない事を示した．

今後の展望として，攻撃者によるデータの改ざんの検知のみだけでなく，データを改ざん前のデータに戻すことを可能にするシステムの考案を考えている．

今後の課題としては，実際にアプリケーションを動かした上での実験，評価がある．

文　献

[1] symantec. Symantec internet security threat report. http://www.symantec.com/region/jp/istr/.

[2] Tim Newsham. Format string attacks. http://musc.linuxmafia.org/lost+found/format-string-attacks.pdf.

[3] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In 22nd Symposium on Reliable and Distributed Systems (SRDS), Oct. 2003.

[4] Anonymous. Once upon a free(). http://www.phrack.org/phrack/57/p57-0x09.

[5] Richard W M Jones and Paul H J Kelly. Backwards-compatible bounds checking for arays and pointers in C programs. In International Workshop on Automated and Algorithmic Debugging, pp. 13–26, 1997.

[6] 情報処理進行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. http://www.ipa.go.jp/security/fy14/reports/oss_security/part3.pdf.

[7] Openwall Project. Non-Executable User Stack. http://www.openwall.com/linux/.

[8] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In the 7th USENIX Security Symposium, pp. 63–78, 1998.

[9] Andre Zuquete. Stackfences: a run-time approach for detecting stack overflows. In 1st International Conference on E-business and Telecommunication Networks(ICETE), Aug 2004.

[10] Hiroaki Etoh and Kunikazu Yoda. propolice:improved stack-smashing attack detection. In IPSJ SIGNotes Computer Security Abstract 43(12), Dec 2000.

[11] Arash Baratloo adn Navjot Singh and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In the 11th USENIX Security Symposium, Une. 2000.

[12] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In the 12th USENIX Security Symposium, 2003.

[13] Greg Hoglund, Gary McGraw, トップスタジオ訳．セキュアソフトウェア．日経 BP 社, 2004.

[14] Sandeep Bhatkar and Daniel C. DuVarney and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In the 12th USENIX Security Symposium , Aug. 2003.

[15] H. Shacham and M. Page and B.Pfaff and E. Goh and N. Modadugu and D. Boneh. On the Effectiveness of Address-Space Randomization. In the 11th ACM Conference on Computer and Communications Security(CCS), Oct 2004.

[16] 長野文昭, 鑓講平, 田端利宏, 櫻井幸一．データ改ざん検出によるバッファオーバフロー検知システムの提案. 情報処理学会コンピュータセキュリティ(CSEC) 研究会, pp. 81–86, 2005.

[17] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In the 12th USENIX Security Symposium, pp. 105–120, Aug. 2003.

[18] 江頭徹, 稲村雄, 竹下敦．マルチタスク OS におけるメモリの保護-データ改ざん検出手法の提案-. 電子情報通信学会技術研究報告 (ISEC), pp. 27–34, Jun 2004.