論　文

データのスクラッチングと動的復元によるバイナリープログラムの不正コピー防止方式

西垣　正勝[†a)]　曽我　正和[††]　井熊　徹[†††]　田窪　昭夫[††††]

Copy Protection of Binary Programs by Data Scratching and Dynamic Restoration

Masakatsu NISHIGAKI[†a)], Masakazu SOGA[††], Tohru IKUMA[†††], and Akio TAKUBO[††††]

あらまし　ネットワーク経由でバイナリープログラムを配信する場合の大きな課題は不正コピーの防止である．クラッカーによるネットワーク経由の不正コピーに対処するのはもちろんのこと，プログラムの正規購入者が悪意をもって行う不正コピーをも防ぐ技術が必要になる．本論文では，プログラムの一部を意図的に改ざんすることにより，不正コピーを無効化する方法を提案する．プログラム中の改ざん部分はプログラムに付けられた傷に相当する．傷を修復するためのクリーナ情報がなければ，プログラムをそのまま実行したとしても正常な動作には至らない．クリーナ情報はプログラムの正規購入者にのみ，購入者の公開鍵により暗号化された上で送られる．プログラムは傷が付けられた状態で配信され，ユーザのファイルシステムに格納される．更にプログラムは，実行の段階で主メモリにロードされた時点においても傷付いたままである．CPU にはクリーナ情報を用いてプログラム中の傷を動的に復元するための機構が付加されており，プログラムの傷は命令実行の直前で修復され，正しく実行される．なお，クリーナ情報の格納場所を含め，動的復元機構は高度に秘匿されるべきモジュールであり，ユーザからハードウェア的に隠ぺいされている．本論文では，上記不正コピー防止システムの具体的な実現方式を示す．本方式の動的復元機構は既存のアーキテクチャに上位互換的に付加可能である．更に，動的復元のプロセスがプログラムの実行速度を落とすこともない．

キーワード　不正コピー防止，動的復元，スクラッチトプログラム，電子配信，上位互換 CPU

1. まえがき

世界のディジタル社会化，特にインターネットの隆盛により，ディジタルコンテンツの流通が非常に活発となった．今後，電子商取引の基盤が整備され，ディジタルコンテンツの売買は加速度的に増加していくであろう．ディジタル社会を実現する上で，著作権の保護に対する万全の対策は不可欠のものとなる [1]．本論文では，対象をバイナリープログラムとし，その不正コピー防止方式について論ずる．

暗号化技術はプログラムを安全に配信するためには有効である．しかし，現状の計算機環境では暗号化技術をプログラムの不正コピー防止のために利用することは難しい．暗号化技術，スクランブル技術を用い，代金を支払った購入者にしかプログラムをコピーさせないという方法 [2] を採っても，一度購入者にプログラムが渡ってしまえば，その後のコピーを取り締まることはできない．これに対し，近年，ディジタル著作物に ID 情報などを透かし情報として埋め込む電子透かし技術 [3] が脚光を浴びている．しかし，電子透かしは著作権等を証明する手段を提供するものであり，不正コピーの抑止力としては働くが，不正コピーを不可能にする技術ではない．また，電子透かしは，情報中に膨大な冗長成分をもつ画像には適用しやすいがプログラムには適用しにくい，コンテンツがクラッカーにより盗み出されて不特定多数にばらまかれた場合に

† 静岡大学情報学部情報科学科，浜松市
Faculty of Information, Shizuoka University, Hamamatsu-shi, 432-8011 Japan
†† 岩手県立大学ソフトウエア情報学部，岩手県
Faculty of Software and Information Science, Iwate Prefectural University, Iwate-ken, 020-0173 Japan
††† 静岡大学大学院理工学研究科，浜松市
Graduate school of Science and Engineering, Shizuoka University, Hamamatsu-shi, 432-8011 Japan
†††† 三菱電機株式会社情報システム製作所，鎌倉市
Mitsubishi Electric Corp., Information System Factory, Kamakura-shi, 247-8520 Japan
a) E-mail: nishigaki@cs.inf.shizuoka.ac.jp

犯人を特定する力はない，などの問題がある．

結局，ソフトウェア的対策のみによってプログラムの不正コピーを完全に取り締まることは困難と言える．不正コピー防止のためには，何らかのハードウェア的な方策が不可欠となる．これに対する一つの有効なアプローチとして，CPU 全体またはその中の一部をハードウェア的に封印した上で暗号化技術を活用するという方法が提案されている [4]～[9]．この方法においては，プログラムは暗号化されたままの状態で保存されることになり，実行直前に復号され，使用される．そして，暗号化プログラムを復号するための鍵をはじめとする復号に関連する一連の情報や結果は，何らかの方法で CPU 内にハードウェア的に封印される．復号機構がユーザの手を離れるため，クラッカーは不正コピーを行う手段を奪われることになる．

ここで，単にプログラム全文を一括して暗号化しておくだけでは不十分である．プログラムは実行前に全文が復号されるため，実行中はオリジナルプログラムがメモリなどに存在することになる．これを秘密に保つためには CPU とメモリのすべてを封印しなければならず，実装上，非現実的であると言える．すなわち，ハードウェアの改良に対する労力や CPU の上位互換性などから，封印すべき情報を最小限に抑える必要がある．

また，プログラムを 1 ステップごとに暗号化しておき，各ステップの度にその部分のみを動的に復号しながら実行するという方法を採用した場合，復号のオーバヘッドによる CPU の実行速度の低下が懸念される．文献 [4] では秘密鍵を封印した CPU と公開鍵暗号方式 [10], [11] を利用することによりプログラムの不正コピー/不正使用を禁止する基本的なプロトコルが提案されているが，公開鍵暗号方式の復号には時間がかかり過ぎる．対処策として，プログラム 1 ステップごとの暗号化を共通鍵暗号方式により行い，その共通鍵を公開鍵暗号方式により暗号化するという方法 [6] や，公開鍵暗号方式を共通鍵暗号方式によりエミュレートする方法 [5] が考えられる．しかし，現状の共通鍵暗号方式のアルゴリズム（例えば DES [12] など）では攻撃耐力を備えるために鍵のビット長や暗号化の段数をある程度大きく採る必要があり，復号にかかる時間は無視できない．したがって，これらの方法では CPU の実行速度に復号が追従できず，現在の 100 MHz クロックオーダの実行速度を維持することは不可能となる．

この問題に対し，文献 [7], [8] では，換字暗号方式により実用的な実行速度を実現する方法が提案されている．ここでは，換字表を格納した ROM などが封印されることになる．しかし，換字暗号方式はカシスキ検査や一致指数による解読法 [13] が発見されており，攻撃耐力が弱いという致命的な欠点が存在する．文献 [8] では，命令語の置換に加え，命令のアドレス情報を暗号化に利用することにより解読耐性を向上させているが，これは逆に，プログラムのアドレス変更の自由度を奪いかねない（プログラムの動的コードリロケーションが不可能となる）．

以上から，本論文では新たに，プログラムの一部を暗号化することにより不正コピーを無効化する方法を提案する．プログラムの場合は適切な 1 箇所を破壊するだけで暴走するので，一部に対して暗号化を施すことによりプログラムの価値を十分に落とすことができる．これはプログラムに傷を付けることに相当するため，本論文ではこの暗号化を「スクラッチング (scratching)」と呼ぶ．プログラム中の傷（暗号化部分）を検出し，それを動的に復旧（復号）する機構は，CPU に軽微の拡張を施すことにより実装される．「スクラッチトプログラム (scratched program)」は，プログラムが実行される最後の瞬間に CPU 上でその傷が修復され（動的復元され），正しく実行される．オリジナルプログラムはいかなる記憶装置上にも残らない．

スクラッチングは，プログラム中の適切な一部分と任意の定数との排他的論理和をとることにより行われる．すなわち，傷の修復に要する演算は排他的論理和のみとなるため，傷の修復プロセスがプログラムの実行速度を落とすことはない．本方式においては，排他的論理和演算に使用した定数に加え，プログラム中のどの部分が傷付けられているかという点が秘密情報となる．したがって，排他的論理和演算に用いた定数を総当り攻撃により推定することも困難であり，かつ，封印すべき情報も十分に少ない．

本方式の採用により，現在の「オリジナルデータ」をベースに考えられているプログラムの流通（図 1）は，図 2 に示される形態へと移行することになる．スクラッチトプログラムは不完全なデータであるので，クラッカー若しくは悪意のある購入者にコピーを取られたとしてもまったく支障がない．

2. プログラム配信のモデル

（1） 本論文で述べるプログラムとは，インターネットや CD-ROM などによりオンライン/オフライ

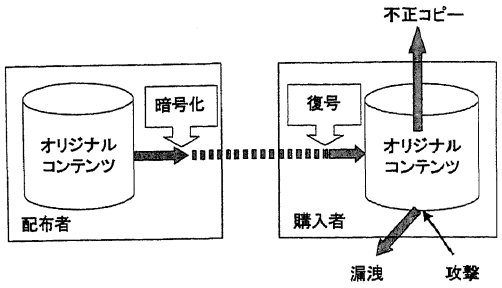

図1 オリジナルコンテンツをベースとした流通
Fig. 1 Distribution of the original contents.

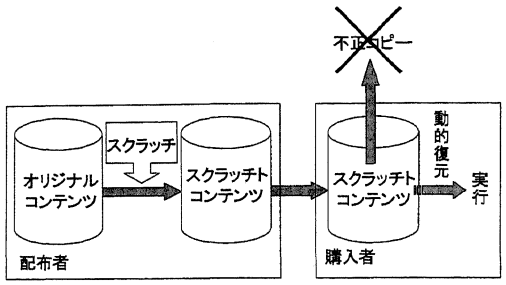

図2 スクラッチトコンテンツをベースとした流通
Fig. 2 Distribution of scratched contents.

ンを通じて配布されるバイナリー形式のプログラムであり，配布を受けたユーザは2次加工を行わないことを前提とする．また，バイナリープログラムを逆アセンブルしてソースプログラムを得るには膨大な労力を要し，経済的に引き合わないことを前提とする．

（2） 本論文で述べるスクラッチングとは，対象プログラムの実効的価値，または商品的価値を意図的に損なうことを目的として，対象プログラムに加える変形を指す（コンテンツの価値を損わないように加工を加える電子透かしとは，この点が決定的に違う）．スクラッチングにより，プログラムは傷付けられ，不完全なデータとなる．

（3） 本論文では公開鍵暗号としてRSA暗号方式[11]を採用して説明を行う．実際には任意の公開鍵暗号方式を用いて構わない．

（4） プログラムに付けられた傷を修復するための秘密情報の配信には，認証用のRSA鍵とは別に，専用のRSA鍵を用いることにする．以下では，このRSA鍵を「配信用RSA鍵」と呼び，認証用のRSA鍵と区別する．配信用RSA鍵は信頼のおける公の機関にて生成され，信頼のおけるCPUメーカに秘密裡に渡されるとする．

（5） 配信用RSA鍵のペアは計算機ごとに生成される．配信用RSA秘密鍵は各CPU内にハードウェア的に封印され，正規ユーザにも隠ぺいされる．具体的には，「セキュアROM」と呼ばれる特別なROMを用意し，そこに秘密鍵を格納する．セキュアROMに対しては，その内容を独立に読み出す機械語命令は用意されていない．一方，配信用RSA公開鍵はCPUメーカによるディジタル署名が付された上で（CPUメーカの認証用RSA秘密鍵にて暗号化された上で）計算機を購入した各ユーザに送付される．

（6） 取り引きにおける三者を次のように名付ける．
プログラムの所有者/配布者：P
プログラムの購入者：U
Uが購入したプログラムを盗もうとするクラッカー：C

（7） 本論文の目的は，Uがオンライン/オフラインを通じて何らかの手段でPから購入し，自己のパソコンに格納したプログラムについて，Uがそれを正規に利用することはできるが，Uがプログラムをコピーしたり，第三者へ配布することを防止する技術を確立することである．それは同時に，Cがインターネット上で，若しくはUのパソコンから，何らかの方法で当該プログラムを盗み出し，不正利用することを防止する技術である．

3. スクラッチングによる不正コピーの防止

3.1 プログラム・スクラッチング

（1） Pは機械語命令1語分の長さの定数Yを用意する．いったん，あるプログラムに対して定数Yを決めたなら，そのプログラムに対しては定数Yは固定される．

（2） Pはプログラムの中から傷を付ける場所を1箇所選ぶ．1箇所とは，連続したn語である．実際に傷が付けられるのは，n語の次の命令Xのオペランド部分となる．Xがオペレーションコード部分を含んでいた場合は，Yの対応部分を0に修正する．

（3） Pは，Xを抜き出し，XとYの排他的論理和Zをもってこれと入れ替えることにより，スクラッチトプログラムを作成する．

Pが手順1で選んだ定数Yと手順2で選んだ場所（n語の先行命令）は，Pのみが秘密裡に管理すべき情報であり，傷を修復しオリジナルプログラムに復元す

るために必要な情報である．よって，この情報を「クリーナ情報」と呼ぶ．もしPが選んだ n 語の先行命令とまったく同じ内容で n 語連続する命令ステップがプログラムの別の場所に存在すれば，Pはその場所の次の命令のオペランド部分も同様に傷付けておかねばならない．

命令Xのオペレーションコードをも定数Yとの排他的論理和をとる対象に含めてしまうと，排他的論理和をとった結果が未使用のオペレーションコードとなる場合があり得る．この場合，実行形式プログラムを逆アセンブルすることにより，傷を付けた箇所が見つけられる恐れが生じる．本方式では，プログラムのオペランド部分のみに傷を付けるため，逆アセンブルを行ってもプログラム中の傷を見つけることは難しい．また，この結果，スクラッチトプログラムは一見通常のプログラムと区別が付かなくなるため，暗号を破ること自体に興味をもつ愉快犯的なクラッカーの攻撃対象になりにくい．なお，スクラッチングを行う者が機械語の知識をもっている場合には，あるオペレーションコードを別のオペレーションコードに適切に変更することによりスクラッチングを施すことは可能である．

先行命令となる n 語は傷が付けられた場所を特定するための情報である．したがって，プログラムの実行段階でその n 語が一意固定的に順次実行される箇所である必要がある．また，プログラムの本質の部分ではないサブルーチンなどにいくら傷を付けたとしても，そのサブルーチンをバイパスするようにプログラムを改造されてしまっては意味がない．よって，傷を付ける箇所はプログラムの中核部分から適切に選出されなければならない．

図3にプログラムのスクラッチング方式を模式的に示す．

3.2 プログラム配信

（1） UはPにプログラム購入の希望を伝えるとともに，Uの使用するCPUの配信用RSA公開鍵をPに送る．なお，配信用RSA公開鍵はCPUメーカによってディジタル署名されている．

（2） PはCPUメーカのディジタル署名をチェックし，送られてきた配信用RSA公開鍵の正当性を検証する．

（3） PはUの配信用RSA公開鍵を用いてクリーナ情報を暗号化する．

（4） Pはスクラッチトプログラムと暗号化したクリーナ情報をUに送る．

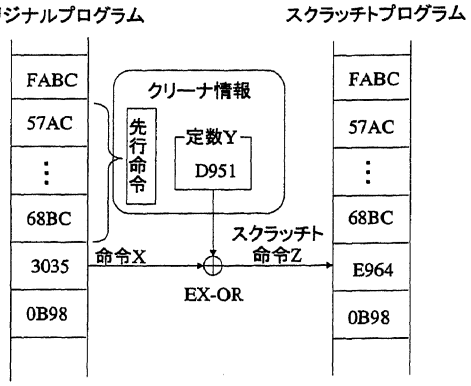

図3 プログラムのスクラッチング方式
Fig. 3 An example of scratching.

（5） UはPから送られたスクラッチトプログラムと暗号化されたクリーナ情報をローカルディスクに保存する．

クリーナ情報のみがユーザごとの配信用RSA公開鍵により個別に暗号化され，配信される．スクラッチトプログラムはどのユーザに対しても同一のものが送られる．このため，プログラム全文をユーザごとに暗号化する方法と比べ，プロバイダの暗号化コストは減少する．また，複数のユーザが結託して，個々に配信されたスクラッチトプログラムをバイナリー比較しても，どこに傷が付けられているかを特定することはできない．

スクラッチトプログラムは不完全な情報であるので，クラッカー若しくは悪意のある購入者にコピーを取られたとしてもまったく支障がない．スクラッチトプログラムが保存されているメディアごとデッドコピーされても問題ない．配信用RSA秘密鍵はCPU中のセキュアROMに格納されており，Uにさえ隠ぺいされている．したがって，配信用RSA公開鍵にて暗号化されているクリーナ情報をUが不正に復号することはできない．逆に，クリーナ情報をUの認証用RSA公開鍵にて暗号化して送ってはいけない．認証用のRSA秘密鍵は各ユーザが保管しているので，Uが不正にクリーナ情報を復号することが可能となってしまうためである．UがPに示したRSA公開鍵が配信用の公開鍵であるかを確認するために，配信用RSA公開鍵には信頼のおけるCPUメーカのディジタル署名が付けられている．

図4にプログラムの配信方式を示す．

3.3 プログラムの動的復元

3.3.1 CPUの拡張

プログラム中の傷を動的に修復するために付加されるCPUの拡張について述べる．図5に模式図を示す．

（1） ユーザに隠ぺいされる特別なレジスタを用意する．これを「セキュアレジスタ」と呼ぶ．セキュアレジスタは既存の機械語命令ではアクセスできない．セキュアレジスタの内容を主記憶上へ読み出す機械語命令は用意されない．したがって，セキュアレジスタ内のデータをのぞくことはできない．

（2） Uの配信用RSA公開鍵で暗号化されているクリーナ情報を復号して，その結果をセキュアレジスタに格納する命令を用意する．この命令を「DecryptSR命令」と呼ぶ．DecryptSR命令を実行するのはCPUとは別途に用意された復号計算用専用装置である．DecryptSR命令によりこの専用装置が起動され，各CPUのセキュアROMに格納されている配信用RSA秘密鍵を使用して一連の復号計算が行われる．専用装置は主記憶やCPUの汎用レジスタを使用せず，復号結果及び復号途中のデータがCPUの管理する記憶装置上に残ることはない．本論文では，この専用装置をファームウェアとみなし，これを起動するDecryptSR命令を「セキュアファームウェア命令」と呼ぶこととする．

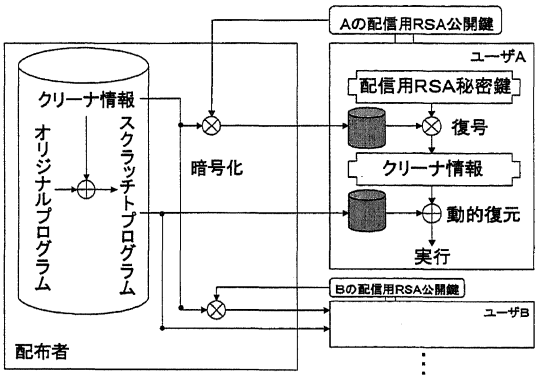

図4 スクラッチトプログラムの配信方式
Fig. 4 Distribution of scratched programs and their patch-data.

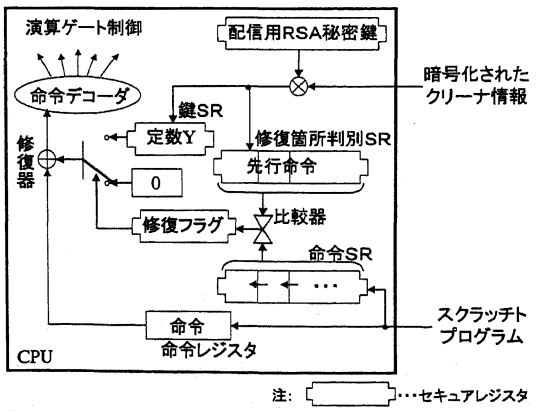

図5 CPUの拡張
Fig. 5 CPU modification for dynamic restoration.

（3） DecryptSR命令によりクリーナ情報が復号された際に，定数Yが格納されるセキュアレジスタを「鍵SR」，先行命令群 n 語が格納されるセキュアレジスタ群を「修復箇所判別SR」と呼ぶ．

（4） プログラム実行中に過去 n 語分の命令を保持するためのセキュアレジスタ群が付加される．これを「命令SR」と呼ぶ．命令SRはシフトレジスタ構成を採っており，過去 n 語分の命令がFIFO式に保存される．

（5） 修復箇所判別SRの内容と命令SRの内容を比較し，両者の一致を検出する回路が付加される．これを「比較器」と呼ぶ．

（6） 比較器が一致を検出すると「修復フラグ」をセットする．修復フラグは，一致を検出した次の命令を実行する際に傷の修復動作を行うように指示を与えるフラグである．修復フラグの内容を保存するレジスタもセキュアレジスタである．

（7） 命令レジスタの内容が命令デコーダに送られる経路に，命令レジスタの内容と鍵SRに格納されている定数Yとの排他的論理和をとる機構が付加される．これを「修復器」と呼ぶ．

これらの拡張は既存のいかなるノイマン形マイクロプロセッサに対しても上位互換的に可能である．換言すれば，修復機構の実現は既存のプロセッサの改良で足りるものであり，また修復機構が追加されたプロセッサは現状機種との上位互換性が完全に保証される．

3.3.2 動的復元方式

（1） Uは当該プログラムを実行する際に，セキュアファームウェア命令DecryptSRを実行してクリーナ情報を復号する．この結果，クリーナ情報（定数Y及び先行命令群 n 語）がそれぞれ鍵SRと修復箇所判別SRに格納される．

（2） Uは当該プログラムを実行する．主記憶から

命令が読み出され，命令レジスタと命令SRに格納される．通常は修復箇所判別SRと命令SRの内容は一致せず，比較器からは0が出力される．この結果，命令レジスタ内の命令がそのまま実行される．

（3） CPUが先行命令群内の最終命令に対応する命令を実行する段階で，修復箇所判別SRの内容（先行命令群 n 語）と命令SRの内容（現命令を含む過去 n 語分の命令）が一致し，比較器から1が出力される．これを受け，修復フラグがセットされる．

（4） 傷が付けられている次命令Zが命令レジスタに読み込まれる．命令レジスタ内ではZは傷付いたままの状態である．しかし，この段階で修復フラグがセットされているため，命令レジスタの内容は命令デコーダのゲート群へ送られる途中で修復器により傷が修復され，命令デコーダに届く．この結果，ZはCPUにより実行される直前に本来の命令Xに復元され，プログラムは正しく実行される．

プログラムは，ファイルされている期間はもちろんのこと，主記憶にロードされた段階においても傷が付いたままである．このように，静的状態でのデータの復元を一切行わず，実行の瞬間にデータを修復することを本論文では「動的復元」と呼ぶ．オリジナルプログラムはいかなる記憶装置上にも残らない．復号された状態のクリーナ情報はセキュアレジスタ内にしか存在しない．セキュアレジスタの内容を読み出す機械語命令は用意されておらず，CやUがクリーナ情報を盗み出すことは不可能である（LSIのパッケージを解いて，プロービングするしかない）．

クリーナ情報はRSA公開鍵暗号方式で暗号化されているため，その復号には時間がかかる．しかし，クリーナ情報は当該プログラムの実行前に前もって復号されるものであるので，プログラムの実行速度を阻害することはない．一方，動的復元は排他的論理和により実現される．排他的論理和の機構はたかだか論理ゲート2段の追加で実現できるので，修復器における動的復元プロセスがプログラムの実行速度を落とすことはない．

なお，例えばプログラム中に条件付きジャンプがある場合，実行時の条件によってプログラムの実行経路が異なり，命令SRに格納される過去 n 語分の命令が変わってくる．よって，条件付きジャンプにかかわる箇所などを先行命令群として選ぶことは避ける．同様の理由で，プログラム実行中に他の命令によってオペランドなどが書き換えられる可能性のある箇所も先行命令群としてはいけない．また，プログラム中に先行命令群と同一となっている命令群が複数存在する場合，すべての箇所の次命令に傷を付ける必要がある．

3.4 実用的観点からの検討

3.4.1 スクラッチング箇所の決定

本方式を効果的に機能させるにおいて，スクラッチングを施す場所の決定が非常に重要となる．プログラムが開発されるにあたり，高級言語によりソースプログラムが作成され，コンパイラによりバイナリープログラムへと変換されるのが主流である．一般にコンパイラの出力するバイナリープログラムは可読性が低いため，実際にバイナリープログラムを解析してその中核部分を見極め，スクラッチング箇所を決定するにはコストがかかる．また，高級言語によりプログラムが作成されるということは，プログラマは必ずしも機械語の知識を持ち合せているわけではない（むしろ，機械語の知識をもたないプログラマがほとんどである）ということを意味する．本方式においては機械語コードに傷をつけるというその性質上，スクラッチトプログラム作成者にはある程度の機械語レベルの知識が要求されることになり，すべてのプログラマが自由にスクラッチトプログラムを作成することができないという問題が残る．

一方，ソースプログラムの解析はバイナリープログラムの解析に比べて容易であり，ソースプログラムのレベルでプログラムの中核部分を判別することは可能であると考えられる（少なくともソースプログラムの作成者本人ならば，ソースプログラム中のどの部分がプログラムの中核であるかは自明であろう）．したがって，ソースプログラム上の中核部分がバイナリープログラムのどこに対応するのかがわかれば都合が良い．高級言語と機械語の命令は1対1対応していないのでソースプログラムにおける中核部分からバイナリープログラムにおける中核部分を確定することはできないが，ソースプログラムの情報からバイナリープログラムの中核部分の位置がある程度の精度でわかりさえすれば，後はバイナリープログラムのその近傍のみを解析することによりスクラッチングに適する箇所を見つけることができる．

ソースプログラムとバイナリープログラムの対応を知ることは，例えば付録に示した方法により可能である．この結果，次のような手順でスクラッチトプログラムを作成することができる．ここで，Sは機械語の

知識を有する人間であり，Pの作成したプログラムにスクラッチを施す作業を請け負う個人または団体である（PとSが同一であっても構わない）．

（1） Pは高級言語でプログラムを開発し，そのソースプログラムとソースプログラムにおける中核部分をSに通知する．

（2） Sは（例えば付録に示される方法を用いて）ソースプログラムの中核部分に関する情報からバイナリープログラムの中核部分を見つける．

（3） Sは，バイナリープログラムの中核部分近傍を解析することにより，スクラッチングに適する箇所を実際に決定する．

ここではソースプログラムの公開が「trap door」となる．Sにはソースプログラムが渡されるため，中核部分近傍の解析のみによりスクラッチング箇所を決定することができる．クラッカーCにはソースプログラムは公開されないため，スクラッチング箇所を特定するための解析が膨大となる．

なお，スクラッチを施すべき場所はCPU，OSにより異なり得るため，CPUやOSに特化した効率的なスクラッチトプログラム作成法が存在することは十分に考えられる．また，本論文はスクラッチング箇所の決定アルゴリズムを限定することを目的としていない．本節及び付録の方法はあくまでもその一例である．

3.4.2 各種アーキテクチャのCPUへの対応

3.1～**3.3**にて説明した本方式は，1ステップの命令がすべて1語で統一されている場合にはわかりやすいが，可変長命令のアーキテクチャやパイプライン，スーパスカラ形のアーキテクチャに対しても適用可能である．

これらのアーキテクチャに対する動的復元機構は個々のCPUの実装方式などとも深く関連するため，ここですべてを詳細に説明することは難しい．しかし，基本的に，

1. 各命令の命令フェッチフェーズにおいて，命令SRと修復箇所判別SRの内容の比較を行い，修復フラグを立てる．
2. 修復フラグが立っている場合は，次命令の実行フェーズにて修復器をenableにする．

という命令実行サイクルを構成することにより，これらのアーキテクチャにおいても動的復元を実現することができる．

なお，命令語長が可変の場合，n語からなる修復箇所判別SRにnステップの命令が収まるとは限らず，先行命令群の命令ステップ数としてはn以下となる．また，先行命令群内の最終命令の後半部がn語の修復箇所判別SRからあふれるケースも起こり得る．しかし，その最終命令の後半部分がフェッチされる直前においては，命令SR側も同様に最後の後半部分はまだ読み込まれておらず，一致検出するに当たっては支障はない．

3.4.3 マルチタスクへの対応

複数のプログラムが同時に実行される場合，CPUは現タスクに関する情報をすべてスタックに退避した上で，次のタスクの実行に移る．プログラムの動的復元機構にはいくつかのセキュアレジスタが存在するが，タスク切替え時にセキュアレジスタの内容がスタック（主記憶）に積まれてしまっては，クリーナ情報が漏洩する原因になりかねない．

したがって本方式をマルチタスクへ対応させるためには，同時に実行するタスクの数だけセキュアレジスタ群を用意する必要がある．更に，PSW（program status words）に数ビットの「セキュアレジスタ群指示フラグ」を用意する．

CPUは，現在何番目のタスクが実行されているかという情報を，PSWのセキュアレジスタ群指示フラグ部に残す．セキュアレジスタ群はPSWの情報を基に，自分がいつ動作すべきかを知ることができる．例えば，i番目のタスクが実行されているときには，PSWの指示フラグはi番を示し，それによりi番目のセキュアレジスタ群が活性化され，当該タスクに対するプログラムの動的復元を行う．

3.4.4 割込みへの対応

割込みが発生した場合，CPUは実行中のプログラムをいったん停止し，割込み処理の後，割込みルーチンの実行を開始する．この結果，命令SRには割込みルーチンの機械語命令が格納されてしまうことになる．したがって，先行命令群に対応する機械語命令が実行されている途中で割込みが発生した場合，命令SRと修復箇所判別SRの内容が一致しなくなってしまう．

3.4.2にて示したPSWの拡張がこの問題も解決する．割込みルーチン実行中はPSWのセキュアレジスタ群指示フラグは立たない．したがって，命令SRはdisableとなり，機械語命令の格納を停止する．割込みが終了した後には，再びPSWのセキュアレジスタ群指示フラグが当該番号となるので，当該命令SRがenableとなり機械語命令の格納を再開する．

4. 考　　察

4.1 汎　用　性

ネットワークを用いたコンテンツ配信の最大の利点は安価な配信コストにある．本方式によれば，スクラッチトプログラムはどのユーザに対しても同一のものを送ることができる．このため，プログラム全文をユーザごとに暗号化する方法と比べ，プロバイダの暗号化コストは減少する．

動的復元機構の実現はいかなるマイクロプロセッサに対しても上位互換的に可能である．また，先行命令群は傷が付けられている命令を機械的に特定するためだけのものであるので，異なる語長の命令が混在しているようなアーキテクチャに対しても本方式は対応可能である．マルチタスクに対応させるためには，セキュアレジスタ群をタスク数だけ用意しなければならない．よって，付加されるレジスタは相当の数になる．しかし，そのレジスタ数は現在のチップ微細化加工技術の前では特に問題とならないと思われる．

現在の方式では，クリーナ情報の復号に使われる配信用 RSA 秘密鍵は CPU のセキュア ROM に格納される．これではユーザの利用できる計算機が固定されてしまうという問題が残る．

4.2 安　全　性

本方式の解読耐性はプログラムのどこに傷を付けたかという情報を秘密にしている点に大きく依存している．ここには，当該プログラムはバイナリー形式でのみ流通し，逆アセンブルによる解読は経済的に引き合わないという前提が存在する．

本方式においては，オリジナルプログラムはディスク上はもちろんのこと，いかなる記憶装置上にも残らない．したがって，本方式に対する攻撃方法としては以下のものが考えられる．これらに対する耐性を検討する．

4.2.1 プログラムのバイナリー比較

クラッカー C が複数のユーザから当該プログラムを盗み出して比較することにより，プログラムのどこに傷が付けられているかを推定しようとしても，スクラッチトプログラム本体はすべてのユーザに対して同一であるため，その場所を特定することはできない．これは，悪意のあるユーザが複数結託して当該プログラムを比較する際も同様である．

クリーナ情報は（内容は同一であるが）各ユーザの配信用 RSA 公開鍵で暗号化されるので，ユーザごとに異なる．暗号化されたクリーナ情報の解読難易度は，RSA 暗号の強度による．

4.2.2 汚染プログラムの改変

C または U が，動的復元機構をもたない（従来の）計算機を利用して，スクラッチトプログラム中の任意の 1 命令を事前に改変して走行させてみるという攻撃が考えられる．動的復元機構をもたない計算機では，傷は修復されないので，プログラムはどこかで暴走することになる．しかし，総当りで何回も試行すれば，いつかはプログラム中の傷を正しく修復することに行き着く可能性がある．

1 回のデータ改変と 1 回の走行テストを合わせて約 8 秒の時間を要するものと仮定する．対象プログラムエリアが 4K $(= 2^{12})$ ステップであるとし，1 ステップ（命令 1 語）が 32 ビットの CPU を考える．命令中に存在するオペレーションコードとオペランドの比率はプログラムにより異なるが，ここでは大雑把にその比率を 50:50 と仮定すると，32 ビット中の 16 ビットがスクラッチングの対象となり得る．よって，総当りの試行に要する時間は

$$\begin{aligned}\text{全試行時間} &= 8[\text{s/回}] \times 2^{12}[\text{ステップ}] \\ &\quad \times 2^{16}[\text{回/ステップ}] \\ &= 2^{31}[\text{s}] \approx \text{約 } 70 \text{ 年}\end{aligned}$$

となり，平均的には全試行回数の半分で正解に行き当たるとしても 35 年を要する．

4.2.3 逆アセンブル

プログラム中の傷は機械語コードのオペランド部分に付けられている．したがって，C がプログラムを逆アセンブルしても，一見正常なアセンブラコードしか得られない．結局，どこに傷が付いているかを特定するためにはプログラムの全文解析をしなければならず，その労力は膨大となる．

4.2.4 プログラム解析ツールを用いた攻撃

C または U が，デバッガ，シミュレータ，エミュレータなどのツールを利用して，スクラッチトプログラムを解析するという攻撃が考えられる．プログラムは傷が付けられている場所から暴走を始めるので，C または U はスクラッチング箇所をある程度は推定することが可能である．

これに対しては，傷が暴走の直接的な原因になるのではなく，

1. ループ文中のループ回数を指定するオペランドに傷を付ける

2. 傷によるループ回数の変化
3. ループ回数の変化による変数値の変化
4. 変化した変数値によるプログラムの暴走

というように，2次的（更には3次的，4次的…）な被害によって暴走が引き起こされるようにスクラッチングの箇所を工夫するという対策が可能である．この結果，プログラム解析ツールによって傷が付けられている場所をある程度推定できたとしても，そこから実際にスクラッチング箇所を特定するまでには多大な時間を要することになる．例えば，老練のクラッカーがプログラム解析ツールを駆使することにより，スクラッチングが存在するであろう範囲を128ステップにまで絞り込めたとしても，**4.2.2**と同様の試算を行うと，128ステップの中からスクラッチング箇所を特定するためには平均して1年が費やされることがわかる．

ただし，クラッカーが数百台の計算機を準備して並列に試行を繰り返した場合，1年という計算複雑度は十分とは言えないかもしれない．しかし，これに対しては，プログラムに複数の傷を付けることにより対処できると考える．一組の動的復元機構の中に修復箇所判別SR，鍵SR，比較器，修復フラグ，修復器を複数組用意することにより，複数のスクラッチを含むプログラムの動的復元が可能となる．この結果，プログラム解析ツールによってスクラッチング箇所をおおまかに推定することも，その中からスクラッチング箇所を特定することも更に難しくなる．

また，付録に示したスクラッチの例のように，プログラムは正常終了するが，その結果が誤るように傷を付けることも可能である．この場合，スクラッチトプログラムは暴走したり，無限ループに突入することはなく，傷付けられている場所を機械的に判別することは不可能となる．すなわち，クラッカーはプログラムの処理手順を解析し，どこにどのような傷が付けられた際には実行結果がどのように誤るかということを調べた上で，実際の実行結果からスクラッチング箇所を推測するという作業が必要になる．

5. むすび

データのスクラッチングとその動的復元による実行形式プログラムの不正コピー防止方式を提案した．スクラッチトプログラムは不完全な情報であるので，クラッカー若しくは悪意のある購入者にコピーを取られたとしてもまったく支障がない．プログラム中に付けられた傷は，計算機上での実行の際にCPUにて動的復元され，正しく実行される．すなわち，オリジナルプログラムはいかなる記憶装置上にも残らない．本方式によれば，ハードウェア的に封印すべき情報は十分少なく，かつ，復号におけるオーバヘッドもない．本論文では，プログラムのスクラッチング，配信方法，動的復元機構について説明した．そして，本方式の攻撃耐性に対する検討を行った．今後，本方式の攻撃耐性に対する詳細な検討及び評価を行う予定である．また，スクラッチング，配信，動的復元の各方式について更に細部にわたっての設計・検証を行っていく．特に，次の点は早急に解決すべき課題である．

- 今回は，クリーナ情報の復号に使われる配信用RSA秘密鍵はCPUのセキュアROMに格納されるという前提を置いた．これではユーザの利用できる計算機が固定されてしまう．クリーナ情報または配信用RSA秘密鍵はICカードなどに格納し，ユーザが使用する計算機を自由に選べるように拡張を行う．
- マルチタスクに対応する場合，どのプログラムが何番目のタスクに割り当てられているかを管理し，どのプログラムが何番目のセキュアレジスタ群を使用するのかを特定しなければならない．ここで，この機構をどのように実現するかが問題となる．一般的に考えると，この割当てはOSが行うべきだと思われるが，OSの改良は本方式の汎用性を低下させる可能性がある．また，OSがセキュアレジスタの管理の一部を行うことにより，本方式の攻撃耐性が落ちるかもしれない．
- マルチCPUをサポートするOS，マルチスレッドのプログラムに対する本方式の適用可能性を検討する．
- 各種アーキテクチャのCPUに対する動的復元機構の詳細設計を行う．
- 本方式を，プログラム以外のディジタルコンテンツにも対応可能なように拡張する．ただしその場合は，1個所のスクラッチングでは商品価値を十分に低下させられないこと，2次加工のニーズもあること等から，前提にさかのぼっての見直し，拡張が課題となる．
- 本方式では，公の機関がユーザごとに認証用RSA鍵のペアを設定することに加え，CPUごとに配信用RSA鍵のペアを用意する．また，CPUメーカは配信用RSA秘密鍵を各CPUのセキュアROMに格納し，かつ，配信用RSA公開鍵にディジタル署名を施した上でユーザに配布しなければならない．これらに対しては，必要に応じては法を整備するな

どの制度の確立が必要であろう．

謝辞 本研究は，一部，(財) テレコム先端技術研究支援センターの助成を受けた．ここに謝意を表する．

文　献

[1] 郵政省監修，21世紀の知的社会への改革，コンピュータエージ社，東京，1994.

[2] 森　亮一，"ソフトウェアサービスシステムについて，" 信学誌，vol.67, no.4, April 1984.

[3] J. Zhao and E. Koch, "Embedding robust labels into images for copyright protection," Proceedings of International Conference on Intellectual Property Rights for Information, knowledge and New Techniques, pp.242–251, Wien, Austria, Aug. 1995.

[4] A. Herzberg and G. Karmi, "On software protection," Proc. 4th Jerusalem Conference on Information Technology, pp.388–393, Jerusalem, Israel, April 1984.

[5] A. Herzberg and S.S. Pinter, "Public protection of software, Advances in Cryptology," CRYPTO '85, pp.158–179, California, USA, Aug. 1985.

[6] D.J. Albert and S.P. Morse, "Combatting software piracy by encryption and key management," Computer, vol.17, no.4, pp.68–73, April 1984.

[7] 坂本広幸，特許公報，昭53-2541.

[8] R.M. Best, Microprocessor for executing encrypted programs, US patent 4,168,396. Issued Sept. 1979.

[9] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," Proc. 19th Annual ACM Symposium on Theory of Computing, pp.182–194, New York, USA, May 1987.

[10] W. Diffie and M.E. Hellman, "New directions in cryptography," IEEE Trans. Inf. Theory, vol.IT-22, no.6, pp.644–654, Nov. 1976.

[11] R.L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Commun., ACM, vol.21, no.2, pp.120–126, Feb. 1978.

[12] National Bureau of Standards: Data encryption standard, Federal Information Processing Standard Publication 46, 1977.

[13] D.R. Stinson, Cryptography:Theory and practice, CRC Press, Inc., Florida, USA, 1995.

[14] B.W. Kernighan and D.M. Ritchie (石田晴久訳)，プログラム言語 C 第2版，pp.131–134，共立出版，東京，1989.

付　録

[スクラッチング箇所決定のための一手法]

3.4.1 にて，ソースプログラムとバイナリープログラムの対応を知ることにより，スクラッチを施す部分を効率的に決定する方法を示した．ここではその詳細について示す．

一般の CPU では print 命令の実行が，(i) 出力文字列の先頭番地をスタックに積み，(ii) print 命令用サブルーチンをコールし，(iii) サブルーチン終了時に使用したスタックを解放する，という手順により実現されることを考慮すると，ダミーとなる print 文をソースプログラムに挿入することにより，ソースプログラム上の位置をバイナリープログラムに伝達できることが予想される．以下，文献 [14] に示されるクイックソートの C 言語プログラムを例に採り，ソースプログラムにおける中核部分をバイナリープログラムに伝える方法を説明する．

クイックソートプログラム [14] を図 A·1 (a) に，そのコンパイル結果を図 A·2 (a) に示す．ただし，図 A·1 (a) においてはプログラムの中核である関数 qsort() のみが，図 A·2 (a) は更に qsort() 内の 6〜

```
 0: void qsort(char *v[],int left,int right)
 1: {
 2:    int i,last;
 3:    if(left >= right)
 4:       return;
 5:    swap(v,left,(left+right)/2);
 6:    last=left;
 7:    for(i=left+1;i<=right;i++)
 8:    {
 9:       if(strcmp(v[i],v[left])<0)
10:          swap(v,++last,i);
11:    }
12:    swap(v,left,last);
13:    qsort(v,left,last-1);
14:    qsort(v,last+1,right);
15: }
```

図 A·1 (a)　オリジナルプログラム
Fig. A·1 (a)　Original program.

```
 0: void qsort(char *v[],int left,int right)
 1: {
 2:    int i,last;
 3:    if(left >= right)
 4:       return;
 5:    swap(v,left,(left+right)/2);
 6:    last=left;
 7:    for(i=left+1;i<=right;i++)
 8:    {
 9:       printf("dummy");
10:       if(strcmp(v[i],v[left])<0)
11:          swap(v,++last,i);
12:    }
13:    swap(v,left,last);
14:    qsort(v,left,last-1);
15:    qsort(v,last+1,right);
16: }
```

図 A·1 (b)　ダミー文付加プログラム
Fig. A·1 (b)　Program with a dummy.

```
; EBP-08h = i, EBP-04h = last, EBP+08h = *v,
; EBP+10h = right, EBP+0Ch = left
; CODE
00401290 | 8B45 0C      MOV EAX,dword ptr[EBP+0Ch]
00401293 | 8945 FC      MOV dword ptr[EBP-04h],EAX
                       ; last4 ← left
00401296 | 8B4D 0C      MOV ECX,dword ptr[EBP+0Ch]
00401299 | 83C1 01      ADD ECX,+01h
0040129C | 894D F8      MOV dword ptr[EBP-08h],ECX
                       ; i ← left+1
0040129F | EB 09        JMP 004012AA ; jump to for-loop
004012A1 | 8B55 F8      MOV EDX,dword ptr[EBP-08h]
004012A4 | 83C2 01      ADD EDX,+01h
004012A7 | 8955 F8      MOV dword ptr[EBP-08h],EDX ; i++
004012AA | 8B45 F8      MOV EAX,dword ptr[EBP-08h]
004012AD | 3B45 10      CMP EAX,dword ptr[EBP+10h]
004012B0 | 7F 3F        JG 004012F1
                           ; if i>right then exit for-loop
004012B2 | 8B4D 0C      MOV ECX,dword ptr[EBP+0Ch]
004012B5 | 8B55 08      MOV EDX,dword ptr[EBP+08h]
004012B8 | 8B048A       MOV EAX,dword ptr[EDX+ECX*4]
004012BB | 50           PUSH EAX ; push up v[left]
004012BC | 8B4D F8      MOV ECX,dword ptr[EBP-08h]
004012BF | 8B55 08      MOV EDX,dword ptr[EBP+08h]
004012C2 | 8B048A       MOV EAX,dword ptr[EDX+ECX*4]
004012C5 | 50           PUSH EAX ; push up v[i]
004012C6 | E8 75030000  CALL 00401640 ; call strcmp()
004012CB | 83C4 08      ADD ESP,+08h
                  ; remove v[i] and v[left] from stack
004012CE | 85C0         TEST EAX,EAX
004012D0 | 7D 1D        JNL 004012EF
          ; if strcmp(v[i],v[left])≧0 then do nothing
004012D2 | 8B4D F8      MOV ECX,dword ptr[EBP-08h]
004012D5 | 51           PUSH ECX ; push up i
004012D6 | 8B55 FC      MOV EDX,dword ptr[EBP-04h]
004012D9 | 83C2 01      ADD EDX,+01h
004012DC | 8955 FC      MOV dword ptr[EBP-04h],EDX
004012DF | 8B45 FC      MOV EAX,dword ptr[EBP-04h]
004012E2 | 50           PUSH EAX ; ++last and push it up
004012E3 | 8B4D 08      MOV ECX,dword ptr[EBP+08h]
004012E6 | 51           PUSH ECX ; push up *v
004012E7 | E8 41FFFFFF  CALL 0040122D ; call swap()
004012EC | 83C4 0C      ADD ESP,+0Ch
                  ; remove *v, last and i from stack
004012EF | EB B0        JMP 004012A1 ; go back to for-loop
```

図 A·2 (a)　オリジナルコード
Fig. A·2 (a)　Original code.

```
; CODE
00401296 | 8B4D 0C      MOV ECX,dword ptr[EBP+0Ch]
00401299 | 83C1 01      ADD ECX,+01h
0040129C | 894D F8      MOV dword ptr[EBP-08h],ECX
0040129F | EB 09        JMP 004012AA
004012A1 | 8B55 F8      MOV EDX,dword ptr[EBP-08h]
004012A4 | 83C2 01      ADD EDX,+01h
004012A7 | 8955 F8      MOV dword ptr[EBP-08h],EDX
004012AA | 8B45 F8      MOV EAX,dword ptr[EBP-08h]
004012AD | 3B45 10      CMP EAX,dword ptr[EBP+10h]
004012B0 | 7F 4C        JG 004012FE
004012B2 | 68 58604000  PUSH 00406058h
                  ; push up dummy's string pointer
004012B7 | E8 94000000  CALL 00401350 ; call printf()
004012BC | 83C4 04      ADD ESP,+04h
      ; remove the string pointer from stack
004012BF | 8B4D 0C      MOV ECX,dword ptr[EBP+0Ch]
004012C2 | 8B55 08      MOV EDX,dword ptr[EBP+08h]
004012C5 | 8B048A       MOV EAX,dword ptr[EDX+ECX*4]
004012C8 | 50           PUSH EAX
004012C9 | 8B4D F8      MOV ECX,dword ptr[EBP-08h]
004012CC | 8B55 08      MOV EDX,dword ptr[EBP+08h]
004012CF | 8B048A       MOV EAX,dword ptr[EDX+ECX*4]
004012D2 | 50           PUSH EAX
004012D3 | E8 78030000  CALL 00401650
004012D8 | 83C4 08      ADD ESP,+08h
004012DB | 85C0         TEST EAX,EAX
004012DD | 7D 1D        JNL 004012FC
004012DF | 8B4D F8      MOV ECX,dword ptr[EBP-08h]
004012E2 | 51           PUSH ECX
004012E3 | 8B55 FC      MOV EDX,dword ptr[EBP-04h]
004012E6 | 83C2 01      ADD EDX,+01h
004012E9 | 8955 FC      MOV dword ptr[EBP-04h],EDX
004012EC | 8B45 FC      MOV EAX,dword ptr[EBP-04h]
004012EF | 50           PUSH EAX
004012F0 | 8B4D 08      MOV ECX,dword ptr[EBP+08h]
004012F3 | 51           PUSH ECX
004012F4 | E8 34FFFFFF  CALL 0040122D
004012F9 | 83C4 0C      ADD ESP,+0Ch
004012FC | EB A3        JMP 004012A1
; DATA
00406050 | 00 00 00 00 25 73 0A 00 ; ....%s..
00406058 | 64 75 6D 6D 79 00 00 00 ; dummy...
```

図 A·2 (b)　ダミー文付加コード
Fig. A·2 (b)　Code with a dummy.

11 行目に対するアセンブリコードのみが記されている．ここで，使用計算機は CPU: Intel Celeron 300A, OS: WindowsNT であり，コンパイルは Microsoft Visual C++ ver.5.0 のコンパイルオプション：Release で実行した．

今，qsort() における 7〜11 行目の for ループのどこかに傷を付けようと考えたとする．プログラマは qsort() の 8 行目と 9 行目の間に，ダミーの printf("dummy") 文を加え，これを再びコンパイルする．ダミー文を加えたソースプログラムとアセンブリコードはそれぞれ図 A·1 (b)，図 A·2 (b) となる．

図 A·2 (b) のアセンブリコードにおいて，文字列 dummy を検索することにより，該当文字列が格納されている先頭番地（図 7 (b) の例では 00406058 番地）がわかる．したがって，次に 00406058 をキーワードにアセンブリコードを検索することにより，この文字列の先頭番地をスタックに積む命令（004012B2 番地）を発見することができ，printf("dummy") に対応する 004012B2 番地〜004012BE 番地のコードを発見す

ることが可能である．

引続き，図A·2(a)のオリジナルコードの中から，図A·2(b)の004012B1番地以前及び004012BF番地以降のコードと等しい部分を検索する．通常はスタック操作命令とCALL命令のみで実現されるprintf文の挿入がコンパイラの最適化を大きく変更することはないので，(アドレスのシフトに気を付ければ）基本的にはこの検索は成功し，オリジナルコードの004012B2番地の直前がダミーのprintf文の挿入位置であったということがわかる．後は，ソースプログラムを参考にしながら，オリジナルコードの004012B2番地近傍を解析することにより，図A·2(a)中のコメント文で記されたプログラムの詳細を知ることができる．

スクラッチの例としては，例えば，004012BE番地の「F8」を「0C」に変更することにより，004012BC番地～004012C5番地の命令を「push up v[i]」から「push up v[left]」へと変更することが考えられる．この場合のクリーナ情報は，排他的論理和をとるための定数Yが「F4」，先行命令が004012BD番地以前のコードである「45 10 7F 3F 8B 4D 0C 8B 55 08 8B 04 8A 50 8B 4D」となる．ただし，説明を簡単にするために，CPUはプログラムを8ビット単位で読み込み，先行命令数nは16であると仮定している．なお，先行命令中に条件付きジャンプ命令である「7F」が含まれるが，本プログラムにおいてはこの条件が偽のときにのみスクラッチト命令が実行されるため，「先行命令は一意固定的に順次実行されるn語である」という**3.1**の条件を満たしている．

もしスクラッチトプログラムをそのまま実行すると，プログラムは正常に終了するが，誤ったソート結果が得られる．クラッカーが誤ったソート結果からスクラッチング箇所を特定することは一般的には難しい．ここで，スクラッチが施されるのはオリジナルのバイナリープログラムであることに注意されたい．ダミーのprintf文が挿入されたバイナリープログラムに傷を付けた場合には，逆にダミー命令がスクラッチング箇所特定の手掛かりとして利用されてしまう．

（平成11年8月3日受付，12年3月14日再受付）

西垣　正勝　（正員）

平2静大·工·光電機械卒．平4同大大学院修士課程了．平7同大大学院博士課程了．日本学術振興会特別研究員（PD）を経て，平8静大情報学部助手．平11静大情報学部講師，現在に至る．博士（工学）．回路シミュレーション，ニューラルネットワーク，通信セキュリティなどに関する研究に従事．IEEE会員．

曽我　正和　（正員）

昭33京大·工·電子卒．昭35京大·修·電子了．昭35～平8三菱電機計算機製作所，情報電子研究所，本社開発本部．平8静岡大学情報学部教授．平11岩手県立大学ソフトウェア情報学部教授，現在に至る．博士（工学）．汎用計算機，制御用計算機，制御用システムの開発に従事．フォールトトレラントシステム，セキュリティシステムに関する研究に従事．

井熊　徹

平11静大·工·知能情報卒．同年同大大学院理工学研究科入学，現在在学中．セキュリティシステムに関する研究に従事．

田窪　昭夫

昭41早大·理工·電気卒．昭43同大大学院理工学研究科修士課程了．同年三菱電機入社．平10静大大学院博士後期課程了．博士（工学）．モバイルコンピューティング，ネットワーク，セキュリティなどに興味をもつ．電気学会，情報処理学会，IEEE，ACM各会員．