抗クラック剤の開発

冨永 和人

東京工業大学 工学部 情報工学科

tominaga@cs.titech.ac.jp

概要

近年インターネットワーク上において 「暴力的なパスワード破り」の方法を用いてシステムに侵入する事件が起きている [クリ 91]。それに対し、この手段によるパスワード破りに対して弱いパスワードを検出することでシステムの安全性を向上させるためのツールに Crack がある。しかし、Crack によるパスワードの検査は暗号化の操作を含むため、時間がかかるのが難点である。

そこで、暗号化の操作を行わずにパスワードの安全性を検査するツール AntiCrack を開発した。AntiCrack は Crack で用いることのできる辞書とルールファイル、および暗号化前のパスワードを入力とし、同じ辞書とルールファイルを用いた Crack でのパスワード検査と同様の効果をより短時間で得ることができる。

1 暴力的なパスワード破り

Crack と AntiCrack は 「暴力的なパスワード破り」からシステムを守るためのツールである。Crack および AntiCrack について述べる前に、このパスワード破りについて簡単に説明する。

1.1 暴力的なパスワード破りとは何か

「暴力的なパスワード破り (brute-force password cracking)」とは、多くのパスワードの候補を用意し、目的のユーザのパスワードがその候補中にないかどうかを探すことによって、不正にそのユーザのパスワードを得ようとすることである。単に 「パスワード破り (password cracking)」とよぶこともある [GM84]。

この方法で発見されやすい (あるいは発見された) パスワードを 「破られやすい」 「弱い (vulnerable)」パスワードとよぶ。

1.2 UNIX のセキュリティ

UNIX では、ユーザの認証をログイン名とパスワードの対によって行う[1)]。つまり、UNIX のセキュリティはその多くをパスワードに頼っている。パスワードは、最大 8 文字の文字列である。

システムはユーザのパスワードを暗号化して、得られた文字列をパスワードファイル (`/etc/passwd`) に格納する[2)]。多くの UNIX システムでは、ユーザは誰でもパスワードファイルを読むことができるようになっている。

ユーザはログイン時に、ログイン名とパスワードの入力を求められる。システムは入力されたパスワードを暗号化し、それをパスワードファイル中のそのユーザの (暗号化された) パスワードと比較する。そして一致すればログインを許す。

UNIX システムで用いられているパスワードの暗号化手続きは公開されているが、効率のよい復号化手続きは現在のところ見つかっていない [FK89]。そのため、パスワードファイルが公開されていても、そこに格納されている暗号化されたパスワードから元の (暗号化されていない) パスワードを効率よく得ることはできない。これが UNIX システムのセキュリティを支えている。

1.3 暴力的なパスワード破りの方法

暴力的なパスワード破りの方法は次の通りである: パスワードの候補を一つ一つ暗号化し、目的のユーザの暗号化されたパスワードと比較する。一致すればパスワードが得られたことになる。

候補としては例えば辞書にある語や目的のユーザのログイン名などが用いられる。

あるいはある種の文字列群 (例えば英小文字だけを用いた 4 文字のパスワードなど) を自動的に生成して、それを

[1)] UNIX のパスワードシステムの詳細は [MT79] を参照のこと。

[2)] 正確には、定数 (64 ビットの 0) をユーザのパスワードを鍵として暗号化し、暗号文をパスワードファイルに格納するのであるが、本稿ではわかりやすさのために、この操作を 「パスワードを暗号化する」とよぶことにする。

候補として用いることもできる。

2 Crack

Crack は暴力的なパスワード破りを行う機能を提供するツールである[3]。

システム管理者などが自分の管理するシステムのユーザのパスワードを Crack によって検査することで、破られやすいパスワードを持っているユーザにパスワードを変更するよう指示できるなど、システムの安全性を高めるのに役立つ[4]。

2.1 Crack の動作の仕組み

Crack の動作の仕組みを簡単に説明する。詳しくは [Muf92] を参照のこと。

2.1.1 候補の生成

Crack のパスワード破りは、基本的には辞書を用いたものである。辞書はユーザが与える。

さらにユーザは、ルールの集合をルールファイル (`dicts.rules`) に与える。

Crack は実行時に、辞書とルールを用いて候補群を生成する。それぞれのルールについて、与えられた辞書のすべての語をそのルールで変形・選択して候補群を作り、その候補群でパスワード破りを試みる。従って、ルールを複数与えることで、一つの辞書からいくつかの候補群のバリエーションを得ることができる。

Crack はさらに、ログイン名および GCOS フィールドに現れる語を別のルールファイル (`gecos.rules`) によって変形・選択してその結果も候補群として試す。

2.1.2 辞書

Crack が用いる辞書は、1 行を 1 語とするテキストファイルである。語は空白を含んでもよい。辞書として複数のファイルを指定することもできる。Crack は実行時に、これらのファイルのコメント行および空行を取り除いて連結し、`sort` そして `uniq` して 1 つの辞書ファイルを作る。

2.1.3 ルールファイル (`dicts.rules`)

ルールファイル `dicts.rules` にはルールを記述する。1 行に 1 つのルールを書く。

表 1: 選択コマンド

コマンドと引数	コマンドの表す条件
<n	入力の長さが n 文字未満
>n	入力の長さが n 文字より大きい
/x	入力が文字 x を含んでいる
/?c	入力が文字クラス c の文字を含んでいる
!y	入力が文字 y を含んでいない
!?c	入力が文字クラス c の文字を含んでいない
=nx	入力の n 文字目が文字 x である[5]
=n?c	入力の n 文字目が文字クラス c の文字である

表 2: 変形コマンド

コマンドと引数	入力の変形
^x	先頭に文字 x を付加する
$$x$	最後に文字 y を付加する
l	大文字をすべて小文字に変える
u	小文字をすべて大文字に変える
c	先頭を大文字に、残りを小文字に変える
r	反転する (『Fred』→『derF』)
d	2 回繰り返す (『Fred』→『FredFred』)
f	『反射』する (『Fred』→『FredderF』)
p	(できる限り) 複数形にする
onx	n 文字目を文字 x に変える
inx	n 文字目に文字 x を挿入する
xnm	n 文字目から m 文字を取り出す
sxy	文字 x をすべて y に置き換える
s?cy	文字クラス c の文字をすべて y に置き換える
@x	文字 x をすべて取り除く
@?c	文字クラス x の文字をすべて取り除く
:	何の変形もしない

ルールはコマンドの並びである。コマンドは 1 文字で表され、コマンドによってはその後に引数 (1 文字から 3 文字) が続く。コマンドには選択コマンドと変形コマンドの 2 種類がある。

ある語に選択コマンドを適用すると、その語がコマンドの表す条件にあっていないときはその語は捨てられ (出力されず、次の語の処理に移る)、条件にあっているときはその語はそのまま出力される。選択コマンドの一覧を表 1 に示す。

変形コマンドは、入力の語に何らかの変形を施して出力する。変形コマンドの一覧を表 2 に示す。

なお、文字クラスは文字の集合であり、あらかじめ定義

[3] ここで述べる Crack のバージョンは v4.1f である。このバージョンは 1992 年 3 月に発表された。

[4] Crack のドキュメントには、Crack パッケージの目的は安全なコンピュータシステムの構築を支援する手段を提供することであると述べられている [Muf92]。

[5] 文字の位置は語の最初の文字を 0 番目とする。

表 3: 文字クラス

クラス	マッチする文字集合
?v	母音 aeiou
?c	子音 bcdfghjklmnpqrstvwxyz
?w	空白文字とタブ文字
?p	句読点文字 .,:;'"?!‘
?s	シンボル $%^&*()-_+=|\[]{}#@/~
?l	小文字
?u	大文字
?d	数字
?a	英字
?x	英数字
??	文字 ? のみを含む集合

なお、?に続く文字を大文字にすることによって、各クラスの補集合を表すことができる (例:クラス?d は数字を表し、?D は数字でないものを表す)

されている。文字クラスの一覧を表 3に示す。

語にルールを適用するということは、そのルール (=コマンドの並び) の最初の (もっとも左の) コマンドにその語を入力し、そのコマンドの出力を次のコマンドの入力とし、これを繰り返して最後の (もっとも右の) コマンドの出力を得ることである。

例　abc, alloy, zebra の 3 語からなる辞書に、ルール>3c$! を適用して得られる候補群は、Alloy!, Zebra! の 2 語となる。

辞書語	>3		c		$!	候補群
abc						
alloy	→	alloy	→	Alloy	→	Alloy!
zebra		zebra		Zebra		Zebra!

2.2 Crack の動作

辞書、ルールファイル、パスワードファイルを入力として Crack を実行すると、Crack はルールファイル中のルールそれぞれを用いて辞書から候補群を生成する。そしてその候補群中の候補をひとつひとつ暗号化し、パスワードファイルにある暗号化されたパスワードと比較する。一致するユーザがあったなら (その候補がそのユーザのパスワードである)Crack は逐一それを報告する (ファイルにその旨のメッセージを出力する)。

すべてのルールについてこの検査が終ると実行終了である。

2.3 問題点

Crack にはいくつかの問題点がある。

まず、実行に時間がかかる。手元の SparcStation10 で実際に実行したところ、実行速度はおよそ 1.1×10^3[語·ユーザ·ルール/秒] であった。これは通常の辞書 (英和辞典などの見出し語の数)10 万語を用いて 200 ユーザのアカウントを 100 個[6] のルールで検査するのにかかる時間がおよそ 1.8×10^6[秒] $\simeq$ 21[日] ということである。

そして、破られやすいパスワードがあったとしても、それが一度システムに格納されてから検出される [Kle]。これでは、そのパスワードが検出されるまでの間、システムをパスワード破りの危険にさらすことになる。

さらに、不正に他人のアカウントのパスワードを得るための (つまり本来の「暴力的なパスワード破り」の) 手段として Crack を利用することもできる。

これらの問題を解決するために開発したツールが AntiCrack である。

3 AntiCrack

AntiCrack は、破られやすいパスワードを (パスワードファイルに登録される前に) 検出することを目的とするツールである。実現には C 言語を用いた。検査にかかる時間は Crack のおよそ数 10 から数 100 分の 1 である。

AntiCrack の使い方は以下の通りである。ユーザは AntiCrack を起動し、自分が新たにパスワードとして用いようとする文字列を入力する。しばらく待つと、AntiCrack はその文字列がパスワードとして破られやすいかどうかを表示する。

ユーザが新しいパスワードを登録する前に AntiCrack でそのパスワードを検査することで、弱いパスワードがシステムに登録されることを防ぐことができ、登録された後で Crack を用いて弱いパスワードを検出するというコストを省ける。

3.1 原理

AntiCrack の動作は、原理的には Crack の動作と同じである。Crack と同じルールと辞書を用いて辞書からルールを用いて候補を生成し、それが与えられたパスワードと等しいかどうかを検査する。

AntiCrack が Crack と異なっている点は、与えられた 1 つのパスワードだけを検査すること (Crack はパスワードファイルを与えられてその全アカウントを検査する)、そして暗号化を用いないことである。

[6] (参考) Crack v4.1f には 240 個のルールが標準で付属している。

3.2 ルール除去

暗号化を用いないだけでも AntiCrack は Crack よりおよそ 2 桁速い[7]が、さらに高速に処理するために、与えられたパスワードに応じて、不要なルールを除去する処理を行う。

ルール除去の原理は「どんな語をそのルールで変形しても与えられたパスワードを生成できないとき、そのルールを除去する」である。例をあげよう: いま仮に alloy というパスワードが与えられたとし、ルール集合には ">2c" というルールがあるとする。このルールが生成する語は、先頭の 1 文字は (それが英字ならば) 大文字になっているはずである。しかし与えられたパスワードは先頭が小文字である。このルールはどんな入力を与えられても alloy を生成することはないので、このルールを除去してもよい。

AntiCrack では第 1 段階としてルール除去によってルールを減らしてから、第 2 段階として残った (受理した) ルールと辞書語を用いて検査を行うことで、不要なルールで辞書語を変形し比較するという無駄な処理を省いている。

参考までにどのくらいの割合でルールが除去されるかを少ない試行回数 (16 回) [8]で実験したところ、現在の実現において、397 個のルール中 9 から 73 ルール (平均 42 ルール) を受理した。

3.3 ルール除去の原理

第 1 段階で行うルール除去の原理を詳しく説明する。

3.3.1 変形関数

変形コマンドは入力文字列から出力文字列への関数とみなせる。何もしないコマンド「:」は恒等関数を表す。

選択コマンドも関数とみなすことにする。選択コマンドは (文字列の変形を行わないので) 恒等関数である。そしてその定義域と値域はいずれも、そのコマンドで捨てられない文字列の集合とする。

このように各コマンドを関数とみなすことにより、文字列にルールを適用して候補を得ることが以下のように定式化できる。

あるコマンド c の表す関数を f とし、f の定義域を D_f、値域を R_f とする (図 1)。この関数を以後「変形関数」とよぶことにする。すると、ルール $c_1c_2\cdots c_n$ にあるそれぞれのコマンドから変形関数 $f_1,\ f_2,\ \cdots,\ f_n$ が得られる。

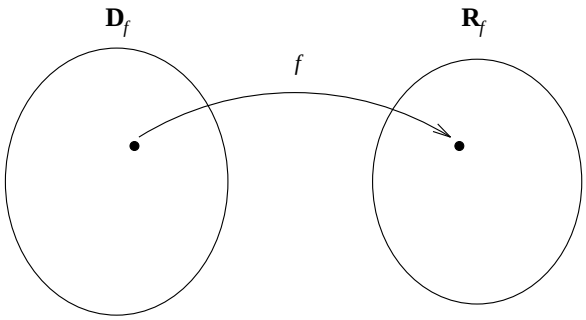

図 1: コマンド c を表す変形関数 f

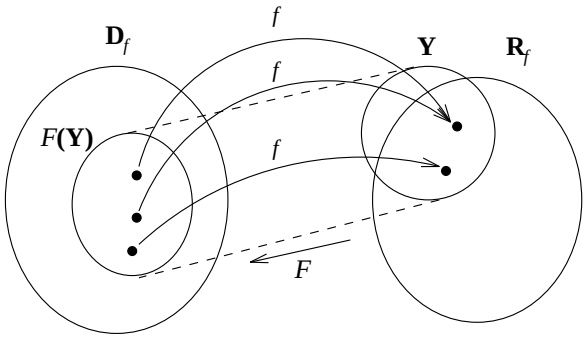

図 2: 逆変形関数 F

そして、辞書語 x に変形関数 $f_1,\ f_2,\ \cdots,\ f_n$ を適用することにより、x にルール $c_1c_2\cdots c_n$ を適用して得られる候補 y を

$$y = f_n(\cdots f_2(f_1(x))\cdots)$$

と表すことができる。ただし、文字列 $f_i(\cdots f_2(f_1(x))\cdots)$ が f_{i+1} の定義域にないときは、x は捨てられることとする。これは、選択コマンドによって入力が捨てられることを表す。

3.3.2 逆変形関数

次に、変形関数 f を用いて、関数 F を次のように定義する。

$$F(Y) = \{x | y = f(x), y \in Y \cap R_f\}$$

すなわち、コマンド c によって集合 Y 中の文字列を出力するような入力をすべて集めたものが $F(Y)$ である (図 2)。この関数 F を以後「逆変形関数」とよぶことにする。

すると、関数 $f_1,\ f_2,\ \cdots,\ f_n$ からそれぞれ逆変形関数 $F_1,\ F_2,\ \cdots,\ F_n$ が得られる。

これらの $F_1,\ F_2,\ \cdots,\ F_n$ を用いると、ルール $c_1c_2\cdots c_n$ を適用されてパスワードの集合 Y を生成する文字列 (辞書語) の集合 X は

$$X = F_1(F_2(\cdots F_n(Y)\cdots))$$

[7] 第 4節「Crack と AntiCrack の比較」の速度比較の表 (表 4) の「AntiCrack(実速度)」の欄参照。

[8] 筆者がいままで用いてきたパスワードのうち思い出せるものすべてを試した。さらに正確な結果を得るには、より多くのパスワードの候補が必要であるが、残念なことに (幸いにも!) パスワード候補の大きな辞書は存在しない。なお英単語の辞書は、含まれる多くの候補の性質が似ている (ほぼすべて英字、多くは英小文字からなる) ため、ルール除去割合の実験には適さない。

で表せる。

すると、ルール $c_1 c_2 \cdots c_n$ がどんな語を入力されても与えられたパスワードの集合 Y 中の文字列を生成できない (すなわちこのルールが不要である) ための必要十分条件は

$$F_1(F_2(\cdots F_n(Y)\cdots)) = \phi$$

となる。したがって、この条件が成立したときにルールを除去、成立しないときにルールを受理すればよい。

なお、AntiCrack では 1 つのパスワードを与えられてそれを検査するにもかかわらずここで Y をパスワードの集合とするのは、UNIX ではパスワードは 8 文字まで有効で、それより長い部分が無視されるためである。

例えばあるユーザが単語 「substitution」を自分のパスワードに選んだとしよう。この文字列の最初から 8 文字「substitu」がパスワードとしてシステムに登録される。すると、このユーザが選択した 「substitution」以外にも 「substitute」や 「substitutive」などがパスワードとして機能する。

したがってこのユーザのパスワードを用いて、あるルールを除去するためには、そのルールが 「substitu」に任意の文字列が続くような文字列を生成しないことを確認しなければならない。この文字列の集合をここで Y とするわけである。

AntiCrack の実現ではパターンを用いて集合を表現しており、この場合は 「*substitu**」という集合を生成しないかどうかでルールの除去の判断を行なっている (ここで 「*」は 0 文字以上の任意の文字列にマッチするメタ文字である)。これについては次の第 3.4節に述べる。

3.4 ルール除去の実現

以上の原理に従い、AntiCrack では次のようにルール除去を実現した。

まず、文字列の集合はパターンで表現した。パターンは以下の基本的なパターンで構成される。

基本パターン	マッチするもの
普通の 1 文字	その文字自身
メタ文字 「.」	任意の 1 文字
メタ文字 「*」	0 文字以上の任意の文字の並び

例えばパターン *substitu** には substitute や substitution などがマッチする。パターン *substitut...* には substitution や substitutive がマッチするが、substitute はマッチしない。

パターンの実現は文字列 (`unsigned char *`) で行なった。普通の 1 文字はその文字自身、メタ文字は第 8 ビットが 1 の文字に割り当てた。これは、UNIX のパスワードに用いられる文字は ASCII 文字 (7 ビット) だからである [MT79]。

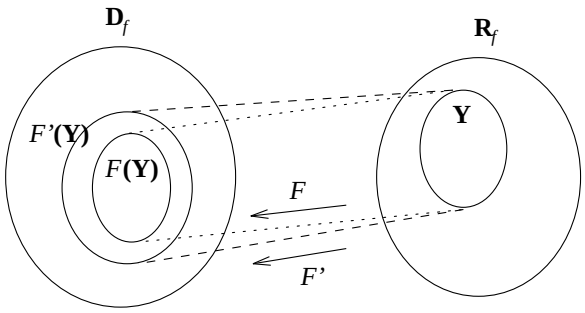

図 3: 近似逆変形関数 F'

パターンの表現能力に限界があるため、パターンで表されたある集合 Y に対して逆変形関数の値 $F(Y)$ がパターンで表現できないことがある。例えば、文字列 Mayday を出力するようなコマンド `sxy` の入力は、Maxdax, Maxday, Maydax, Mayday の 4 つのみである。これらの 4 つの文字列を含む集合をこのパターンで表現すると例えば $Ma.da.$ となる。ところがこのパターンにはこの 4 つの文字列以外の文字列も含まれる。

$$\text{Mayday} \xleftarrow{\texttt{sxy}} \left\{\begin{array}{l}\text{Maxdax}\\ \text{Maxday}\\ \text{Maydax}\\ \text{Mayday}\end{array}\right\} \subseteq Ma.da. = \left\{\begin{array}{c}\text{Maadaa}\\ \text{Maadab}\\ \text{Maadac}\\ \vdots\\ \text{Maxdax}\\ \text{Maxday}\\ \vdots\end{array}\right\}$$

そこで AntiCrack の実現では、逆変形関数 F の代わりとして、$F(Y) \subseteq F'(Y)$ となり $F'(Y)$ がパターンで表現できる関数 F' を考える (図 3)。この関数を以後 「近似逆変形関数」とよぶ。上の例では、例えばコマンド `sxy` に対応する近似逆変形関数 F' として、$F'(Mayday) = Ma.da.$ となるものを AntiCrack では実現するということである。

逆変形関数 $F_1, F_2, \cdots, F_n$ それぞれに対応する近似逆変形関数 $F'_1, F'_2, \cdots, F'_n$ を用いると、

$$F_1(F_2(\cdots F_n(Y)\cdots)) \subseteq F'_1(F'_2(\cdots F'_n(Y)\cdots))$$

となる。したがって

$$F'_1(F'_2(\cdots F'_n(Y)\cdots)) = \phi$$

が 「任意の入力をルール $c_1 c_2 \cdots c_n$ で変形した結果が与えられたパスワードの集合 Y 中に存在しない」ための (すなわち、そのルールが不要である) 十分条件となる。

そこで AntiCrack の実現では $F_1'(F_2'(\cdots F_n'(Y)\cdots))$ を計算し、その結果がϕならばそのルールを除去することにする。十分条件を利用するので、集合 Y 中の文字列を生成する可能性のある (つまり必要な) ルールを除去してしまうことはない。

これに基づき、Crack の各コマンドについて、対応する近似逆変形関数を次節のように実現した。

3.4.1 近似逆変形関数の実現

近似逆変形関数を手続きとして実現する方法を述べる。各々の手続きは 2 つのステップで計算を行うように設計した。以下では、コマンドを c、それに対応する変形関数を f、逆変形関数を F とし、求める近似逆変形関数を F' とする:

1. 与えられたパターンと、f の値域との交わりを計算する。

   逆変形関数 $F(Y) = \{x | y = f(x), y \in Y \cap R_f\}$ の、$Y \cap R_f$ にあたる集合の計算である。ここでも、計算の結果 Y' が完全に $Y \cap R_f$ に一致する必要はなく、$Y \cap R_f \subseteq Y'$ であれば、条件 $F(Y) \subseteq F'(Y)$ は満たされるのでよい。

   ここで $Y' = \phi$ になったなら、以後の計算をしなくても $F_1'(F_2'(\cdots F_n'(Y)\cdots)) = \phi$ となることは明らかなので、処理中のルールを除去する。

2. Y' に F を適用した結果を表すようなパターンを計算する。

なお、何もしないコマンド 「:」は定義域値域ともに文字列全体で、変形関数は恒等関数なので、その近似逆変形関数の実現ではステップ 1, 2 とも何もする必要がなく、与えられたパターンをそのまま返せばよい。

また、選択コマンドに対応する変形関数はすべて恒等関数であり、その近似逆変形関数も恒等関数となるので、ステップ 2 では何もする必要がない。したがって、ステップ 1 の結果をそのまま返せばよい。

実現の例　コマンド sxy の近似逆変形関数の実現の例をあげる。コマンド sxy は、入力にあるすべての文字 x を文字 y に変えるコマンドである。

ステップ 1 では、与えられたパターン中にパターン x があるかどうか検査する。パターン x があれば、それはコマンド sxy の出力を含まない (つまりこのパターンを Y とすると、$Y \cap R_f = \phi$ である) ので、処理中のルールを除去する。与えられたパターン中にメタ文字 (「」か 「*」のいずれか) があれば、このパターンの表す集合には文字 x を含む文字列があるが、$Y \cap R_f \subseteq Y'$ ならよいので、元のパターンそのものをこのステップの結果とする。

次にステップ 2 では、前ステップの結果のパターン中に y があればそれを 「」に書きかえる。なぜなら、コマンドの出力中の y は、入力では x か y のいずれかであったはずであるからであり、パターンを用いてこれを表現するには 「」を用いるしかないからである。

第 3.4 節にあげたパターンの例 $Mayday$ にこの近似逆変形関数を適用すると、$Ma.da.$ が得られることがわかる。

3.4.2 ルール除去の手順

実現した近似逆変形関数を用いてルールの除去を行う。手順は以下の通りである。

1. 入力されたパスワードをパターンに変換する。

   パスワードの長さが 8 文字未満ならパスワードそのものをパターンとする。8 文字以上ならば先頭から 8 文字を取り出し、その後にメタ文字 「*」をつけたものをパターンとする。下に例をあげる。

パスワード		パターン
abcdef	→	$abcdef$
anonymous	→	$anonymou*$

2. ルールにあるコマンドの逆順に、それぞれのコマンドに対応する近似逆変形関数をそのパターンに適用することにより、$F_1'(F_2'(\cdots F_n'(Y)\cdots))$ を求める。
3. その結果として得られたパターンが空文字列 (空集合を表す) ならそのルールを除去し、空文字列でなければそのルールを受理する。

受理されたルールを用いて、第 2 段階で辞書語の変形および与えられたパスワードとの比較を行う。

3.5 辞書語の変形 ・パスワードとの比較

第 1 段階で受理されたルールを用いて、Crack と同様に辞書語を変形して候補を作りだし、パスワードと比較する。辞書語の変形は Crack パッケージに付属している、C 言語で記述された関数群を利用した。候補とパスワードの比較は、大文字小文字を区別し先頭から 8 文字までで行なった。

比較してパスワードとある候補が一致した場合には、そのパスワードを破られやすいとみなし、ユーザにその旨を知らせるメッセージを出力する。オプション指定により、

どの辞書語がどのルールで変形された結果パスワードと一致したかを表示することも可能である。

付録 A に AntiCrack の実行例を示す。

4 Crack と AntiCrack の比較

Crack と AntiCrack との速度の比較を表 4に示す。参考に、各マシンの暗号化の速度も併記する。表において、AntiCrack に「実速度」と「見かけの速度」の値があるが、それらの意味は次の通りである。

実速度は、受理したルールの数を用いて速度を計算したときの値である。つまり、与えたルールを一つも除去できないときの速度を表していると考えてよい。

見かけの速度は、試行において与えたルールの数を用いて速度を計算したときの値である。与えたルールの数から計算にかかる時間を予想するときに参考にすることができる。ただし、与えたルールおよび入力したパスワードによって除去できるルールの数は大きく変わるので、試行時にもこの速度には大きなばらつきがあった。下段かっこ内に示した値が試行時の速度の最大値と最小値である。

比較の結果、AntiCrack は (実速度で)Crack の数 10 倍、あるいは (見かけの速度で) 数 100 倍の速度で 1 つのパスワードを検査できることがわかった。この差は例えば SparcStation での実験結果の値 (表 4より) を用いると、

- ルール数 200
- 辞書の語数 100000 語

を用いて 1 つのパスワードを検査したときに、Crack では

$$\begin{aligned} 100000 \times 1 \times 200/(1.1 \times 10^3) &\simeq 18000[\text{秒}] \\ &= 5[\text{時間}] \end{aligned}$$

かかるのに対して、AntiCrack では最悪で

$$\begin{aligned} 100000 \times 200/(9.0 \times 10^4) &\simeq 222[\text{秒}] \\ &= 3[\text{分}]42[\text{秒}] \end{aligned}$$

しかかからず、今回の試行で得られた平均値を用いると平均

$$100000 \times 200/(6.3 \times 10^5) \simeq 32[\text{秒}]$$

で検査が終るということである。

5 議論

近似逆変形関数の利用によるルール除去　AntiCrack では、逆変形関数ではなく近似逆変形関数を用いてルールの除去を行なっている。この影響は次の通りである。

いま、あるルール $c_1 c_2 \cdots c_n$があり、それぞれのコマンドに対応する逆変形関数および近似逆変形関数を $F_1, F_2, \cdots, F_n$および $F_1, F_2, \cdots, F_n$とし、与えられたパスワードを Y とすると、AntiCrack で用いたルール除去の条件は

$$F'_1(F'_2(\cdots F'_n(Y)\cdots)) = \phi$$

である。

この条件でルールの除去を行うと、

$$F'_1(F'_2(\cdots F'_n(Y)\cdots)) \neq \phi$$

かつ

$$F_1(F_2(\cdots F_n(Y)\cdots)) = \phi$$

となるときは、ルール $c_1 c_2 \cdots c_n$が実際には与えられたパスワード集合 Y 中のどの文字列も生成できないにもかかわらず、AntiCrack はその (無駄な) ルールを受理してしまう。これは実行速度の低下を招く。

しかし、必要なルールを除去することはない (第 3.4節参照) ので、近似逆変形関数の利用は AntiCrack の「正しさ」には影響を与えない。ここで「正しさ」とは、Crack との実行結果の等価性のことである。つまり、近似逆変形関数を用いても、同じ辞書とルールを与えたときに Crack で破られないパスワードは AntiCrack でも破られず、Crack で破られるパスワードは AntiCrack でも破られる。

無駄なルールをできるだけ受理しないようにするためには、よりよい近似逆変形関数を実現する必要がある。これについては次の項「近似逆変形関数の実現」で述べる。

近似逆変形関数の実現　AntiCrack では近似逆変形関数を用いたことにより、段階的な実現および改良が可能であった。

ある逆変形関数 F に対する近似逆変形関数 F' は、任意の文字列集合 Y について

$$F(Y) \subseteq F'(Y)$$

となるようなものならどんなものでもよい (第 3.4節参照)。極端には、どんな引数に対しても文字列全体の集合 I を返す関数 G を考えれば、G は任意の逆変形関数に対する近似逆変形関数である。

このように近似逆変形関数には選択の余地があるため、あるコマンドに対する近似逆変形関数として最初は「ゆるい」ものを実現し、徐々に「きつい (よい)」ものに改良していくことができる。以下にあげる例は、実際に AntiCrack の実現で行なった改良の過程である。

繰り返しのコマンド d に対する近似逆変形関数の実現の過程:

表 4: 速度の比較

	NEWS5000SB	HP9000/730	SparcStation10
Crack [語･ユーザ･ルール/秒]	1.8×10^3	1.8×10^3	1.1×10^3
AntiCrack (実速度) [語･ルール/秒]	1.3×10^5	1.1×10^5	9.0×10^4
AntiCrack (見かけの速度) [語･ルール/秒]	9.1×10^5 ($5.0 \times 10^5 \sim 1.6 \times 10^6$)	8.0×10^5 ($4.3 \times 10^5 \sim 1.3 \times 10^6$)	6.3×10^5 ($3.4 \times 10^5 \sim 1.0 \times 10^6$)
crypt (crypt(3)) [暗号化/秒]	184	75	85
crypt (ufc-crypt) [暗号化/秒]	1240 ∼ 1900	1920 ∼ 3750	900 ∼ 1340

測定条件: 測定はすべて CPU の負荷が軽い (ほかに CPU による計算を定常的に行うプロセスがない) ときに行なった。

- **Crack**

  ルール数は 10 から 400 程度にさまざまに変え、辞書語数も 2 万から 60 万までさまざまに変えて、それぞれ 25 回以上ずつ計測し、速度を計算してその平均をとった。

- **AntiCrack**

  ルール数が 11 で辞書語数は約 57 万語という組合せと、ルール数が 397 で辞書語数が 11 万語という組合せで、それぞれ 6 回ずつ各マシンで測定した速度の平均を記した。速度の計算に用いたルール数は、「実速度」では受理されたルールの数、「見かけの速度」では与えたルールの数である。

1. I を返す (G そのもの)

   プログラマ (筆者) は、ほかの (より簡単そうな) コマンドの近似逆変形関数を実現していた。

2. パターンの表す文字列が固定長なら、その長さが奇数のものを除去し、偶数なら I を返す

   文字列を 2 回繰り返したら長さは当然偶数である。これだけの簡単な実現でも、例えば 7 文字のパスワードは>2r というルールを除去することができる。

3. パターンの表す文字列が固定長なら、それが 2 回繰り返した形になっているかどうかを判断し、なっていたら元の集合を含むパターンを返す

   例えば $ma.m.y$ は、何かを 2 回繰り返した形になっている (このとき「.」は任意の 1 文字にマッチすることに注意)。そして、第 3.4.1節の手順のステップ 1により、交わりが $maymay$ であることが計算できる。そして結果として may を返す。

   不定長 (「*」を含む) なら I を返す。

4. パターンが「固定長の文字列*」の形になっていれば、元の集合を含むパターンを計算できる

   例えば $a..b*$は、$a..b$ にマッチする文字列に任意の文字列を追加したものを 2 回繰り返した文字列 (例えば axcbzaxcbz) にマッチする。

   さらに、ab を 2 回繰り返した abab も $a..b*$にマッチする。

   よって $a.*$を返す。

このように、徐々に結果の集合の小さい近似逆変形関数を実現することで、不要なルールを多く除去するように改良することができる。

重要なことは、必ず $F(Y) \subseteq F'(Y)$ となる近似逆変形関数 F' を実現することである。これを守らないと「正しさ」が保証されない。

パターンの表現能力 AntiCrack では実現の容易さのために、パターンの表現力を第 3.4節のように制限している。パターンの表現力を強化すると、各コマンドに対する近似逆変形関数もすべてそれに応じた実現をしなければならないからである[9]。

UNIX で用いられる正規表現ほどの表現能力が AntiCrack のルール除去に有効か (実現するコストに応じた見返りがあるか) はわからないが、Crack のルールでは大文字小文字を扱うことが多いので、*[Aa]* や *[Bb]* にあたる表現を導入することには意味があるかもしれない。

なお、場合によっては、ルールの記述を工夫するだけでルール除去を容易にすることができる。例えば、@?vl は

[9] 例えば正規表現の *[ ]* を導入したとすると、コマンド sxy の近似逆変形関数をパターン $Mayda[yz]$ に適用した結果を $Ma[xy]da[xyz]$ とするとか、コマンド d の近似逆変形関数をパターン $[ab]bca[ab]*$に適用した結果を $ab*$とすることになるだろう。

母音をすべて取り除き、すべて小文字にする」というルールであり、母音 a を含むパスワード happy はこのルールでは生成されない。しかしコマンド `l` の近似逆変形関数がパターン *happy* をパターン..... にしてしまうと、コマンド`@?v` の近似逆変形関数では..... に母音があるかどうかがわからず、`@?vl` を受理することになる。これは、ルールを `l@?v` にすることで避けることができる。この場合、パターン *happy* に母音があるので、`@?v` の近似逆変換関数には母音のある文字列はないから交わりが空集合になり、ルールを除去できる。

GCOS フィールド　第 2.1.1 節で説明したように Crack は GCOS フィールドからも情報を得てパスワード破りを試みる[10]が、AntiCrack は GCOS フィールドの情報を扱わない。AntiCrack はユーザのログイン名を得ないためである。AntiCrack で GCOS フィールドの情報を得ないようにした理由は次の通りである:

- AntiCrack は必ずしもそれを利用するユーザの UID で動作することを仮定していない。

  パスワードの安全性の検査では、どのような検査が行われるかをユーザがあらかじめ知らないほうがよい場合がある。検査の内容がわかると、ユーザによってはその検査をすり抜けるためだけのパスワードをつくり出すかもしれないからである。

  このような場合は、辞書やルールファイルを例えばユーザ crack の所有にしパーミッションを 0600 にしておいて、AntiCrack の実行ファイルを crack の所有で set uid(例えばパーミッション 4711) しておくことが役に立つ。

- トロイの木馬を仕掛けられるのを防ぐ。

  前項とも関係するが、万が一 AntiCrack の顔をしたトロイの木馬が仕掛けられたとする。パスワードの検査を行なったユーザは、その検査でパスワードが破られなかったなら、多分そのパスワードを使うだろう。もし AntiCrack がユーザ名の入力を要求したり、ユーザの UID で動作するものなら、トロイの木馬はユーザ名とパスワードの組を、それを仕掛けた人間に送るだろう。

  AntiCrack の利用者が誰であるかの情報を AntiCrack が得ないようになっていれば、ユーザ名がわからないので、この方法によるパスワードの窃盗をある程度防ぐことができる。

そこで、Crack の行う GCOS フィールドによる検査を AntiCrack でも行うには、GCOS フィールドから得た情報をファイルにして辞書とし、`gecos.rules` をルールファイルとしてパスワードを検査すればよい。その際、すべてのユーザの GCOS フィールドの情報を辞書にしておくと、より厳しい検査が可能になる。

6　AntiCrack 現在 未来

AntiCrack は C 言語で記述した。ソースファイルのサイズを以下に示す (コマンド `wc` による)。

行数	語数	バイト数	ファイル
295	1097	7552	anticrack.c
53	148	1195	anticrack.h
1280	4321	26557	rejrule.c
1628	5566	35304	合計

anticrack.c が AntiCrack の本体で、入力されたパスワードを辞書と、受理したルールで検査する。これは基本的には単純なループと文字列比較なので、ファイルのサイズも小さくなっている。

rejrule.c が入力されたパスワードを用いてルールを除去する部分である。

AntiCrack をコンパイルし実行するには、このほかに Crack のパッケージにあるファイル crack-lib.c, crack.h, conf.h, crack-glob.h が必要である。

現在は AntiCrack に対してテストと改良を行なっている。バグが見つからなくなり次第、ftp で公開する。

今後は除去できるルールをより多くするために、よりよい近似逆変形関数の実現やパターンの能力拡張を行う予定である。

謝辞

AntiCrack の開発に関して非常に貴重なコメントをしてくださった加藤 朗氏に感謝する。

参考文献

[FK89] David C. Feldmeier and Philip R. Karn. UNIX Password Security – Ten Years Later. In *Advances in Cryptology – CRYPTO '89 Proceedings*, pp. 44–63. Springer–Verlag, 1989.

[10] Crack がどのように GCOS フィールドの情報を利用しているかについては、[Muf92] を参照のこと。

[GM84] F. T. Grampp and R. H. Morris. UNIX Operating System Security. *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pp. 1649–1672, October 1984.

[Kle] Daniel V. Klein. "Foiling the Cracker": A Survey of, and Improvements to, Password Security. Ftp available from ftp.sei.cmu.edu: `~/pub/dvk/passwd.*`.

[MT79] Robert Morris and Ken Thompson. Password Security: A Case History. *Communications of the ACM*, Vol. 22, No. 11, pp. 594–597, November 1979.

[Muf92] Alec D. E. Muffett. *"Crack Version 4.1" A Sensible Password Checker for Unix*, April 1992. `Docs/readme.txt`.

[クリ 91] クリフォード・ストール. カッコウはコンピュータに卵を産む. 草思社, 1991.

A AntiCrack の実行例

AntiCrack 実行中の画面表示例を図 4, 5に示す (入力したパスワードはエコーバックされていないことに注意)。行末や字下げしてある日本語混じりのコメントは説明のためのものであり、画面には表示されない。

また、どのルールを用いてどういう辞書語を変形した結果、破られやすいと判定されたかを表示させることもできる (図 6参照)。

デバッグオプションを指定して、ルール除去の過程を表示させたものが図 7, 8, 9である。

```
= mike(~/crack/ac) ./AntiCrack
Welcome to AntiCrack.
Your password candidate:        「alloys」と入力
Please type it again:           もう一度入力
Cracked. Please try another candidate.
        破られやすいパスワードだった。
```

図 4: AntiCrack の実行例 (1)

```
= mike(~/crack/ac) ./AntiCrack
Welcome to AntiCrack.
Your password candidate:        「robotcat」と入力
Please type it again:           もう一度入力
Not Cracked.
        このルールと辞書では破られない。
```

図 5: AntiCrack の実行例 (2)

```
= mike(~/crack/ac) ./AntiCrack -ps
Welcome to AntiCrack.
Your password candidate:            「alloys」
Please type it again:
8/52 rules are accepted
        52 ルール中 8 ルールが受理された。
Cracked. "alloy" (>2!?Alp) -> "alloys"
Cracked. Please try another candidate.
        辞書語 alloy がルール >2!?Alp によって
        alloys に変形され、入力されたパスワードと
        一致してしまった。
```

図 6: 結果の表示

```
= mike(~/crack/ac) ./AntiCrack -ds
Welcome to AntiCrack.
Your password candidate:
Please type it again:              「alloys」
debug: candidate is "alloys"
```

（途中略）

```
debug: invert commands "!?Al" -> "l!?A"
```

ルールを反転する。

```
debug: password "alloys" -> pattern "alloys"
```

パスワードをパターンに変換。8 文字未満なので、そのままパターンとする。

```
debug: pattern "alloys", rules rest "l!?A"
```

パターン `alloys` に、大文字をすべて小文字にするコマンド `l` の近似逆変形関数を適用する。`alloys` にあるすべての英字は小文字なので、`alloys` はこの関数の定義域にある。

逆変形関数を `alloys` に適用した結果は `{ alloys Alloys aLloys ... aLLOYS ALLOYS }` という集合であるが、`AntiCrack` のパターンはこれを表現できないため、近似逆変形関数はこれを含む集合として `......` を返す。

```
debug: pattern "......", rules rest "!?A"
```

`!?A` は「英字でないものを含むなら除去する」という選択コマンドである。近似逆変形関数は入力と同じパターン `......` を出力する。`......` の表す集合には英字でないものを含む文字列もあるが、コマンド `!?A` が `......` に含まれる文字列を出力するための入力はすべて `......` に含まれるので、近似逆変形関数はこれをそのまま返す（ように実現した）。

```
debug: Accepting "!?Al"
```

`......` は空文字列ではないので `!?Al` を受理する。

図 7: ルール除去の様子 (1)

```
debug: invert commands ">2!?Alp" -> "pl!?A>2"
debug: password "alloys" -> pattern "alloys"
debug: pattern "alloys", rules rest "pl!?A>2"
```

コマンド `p` は複数形にする変形コマンド。パターンは `alloys` から `alloy` になる。

```
debug: pattern "alloy", rules rest "l!?A>2"
```

小文字にするコマンド `l` に対しては、前述したように英小文字の部分を `.` に変える。

```
debug: pattern ".....", rules rest "!?A>2"
debug: pattern ".....", rules rest ">2"
```

パターン `.....` にマッチする文字列の長さは 5 なので、`>2` の近似逆変形関数の結果は空集合にはならない。

```
debug: Accepting ">2!?Alp"
```

`>2!?Alp` を受理する。

図 8: ルール除去の様子 (2)

```
debug: invert commands "/?X" -> "/?X"
debug: password "alloys" -> pattern "alloys"
debug: pattern "alloys", rules rest "/?X"
```

コマンド `/?X` は、英数字でないものを含まないなら除去するという選択コマンドである。`alloys` には英数字でないものはないので、パターン `alloys` の表す集合と、`/?X` の変形関数の値域との間には交わりがない。したがってこの集合に近似逆変形関数を適用した結果は空集合となる。

```
debug: Rejecting "/?X"
```

結果が空集合なので、どんな辞書語を与えてもルール `/?X` が `alloys` を生成することはない。よって `/?X` を除去する。

（途中略）

```
Cracked. "alloy" (>2!?Alp) -> "alloys"
```

受理されたルール `>2!?Alp` で破られた。

```
Cracked. Please try another candidate.
```

図 9: ルール除去の様子 (3)