The Collberg-Thomborson Watermarking Algorithm

Authors

Christian Collberg (collberg@cs.arizona.edu)
Gregg Townsend (gmt@cs.arizona.edu)

Description and Examples

This algorithm is a dynamic software watermarking method that embeds the watermark in the topology of a graph structure built at runtime. Watermarking a Java jar-file using the CT algorithm and recognizing that watermark requires several phases. First, the source program has to be **annotated**. This means that calls to `sandmark.watermark.trace.Annotator.mark()` are added to the source program in locations where it is OK to insert watermarking code. Next, the source program is compiled and packaged into a jar-file. Then the program is **traced**, i.e. run with a special (secret) input sequence. This constructs a trace-file, a sequence of mark()-calls that were encountered during the tracing run. The next step is **embedding** which is where the watermark is actually added to the program, using the data from the tracing run.

Before distributing the resulting watermarked program it should be obfuscated, otherwise it will be susceptible to *collusive* attacks.

The watermark is recognized by again running it with the special input sequence.

These different steps are described further here:

- Annotation
- Tracing
- Embedding
- Recognition

Annotating Programs in the CT Algorithm

Annotations identify the points in a program where the watermarker can insert additional code.

> Annotations take the form of extra method calls added to the source code. At execution time, these methods write messages to a tracing log.
>
> One execution sequence, as defined by a specific sequence of input actions, is chosen by the program author to trigger recognition of the watermark. This is the "recognition sequence", and the corresponding path through the code is the "recognition path".
>
> The watermarking process generates special code that replaces the annotation calls. This code has no effect except to manipulate internal data structures that are otherwise unused.

Example

Below is an example program, TTT.java which has been annotated with calls to **sandmark.watermark.ct.trace.Annotator.sm$mark(*)**.

```
import java.awt.*;
import java.awt.event.*;

public class TTT extends Frame {
  Label lab = new Label("Tic-Tac-Toe", Label.CENTER );
  Panel pan = new Panel();
  Button [] sq = new Button [ 9 ];
  Panel south = new Panel();
  Button quit = new Button("Exit");
  Button reload = new Button("Reload");
  int ply = 2;

  public void start() {
    for ( int p = 0; p  9; p ++ ) {
      sq[ p ] = new Button( "." );
      sq[ p ].setActionCommand( Integer.toString( p ) );

      sq[ p ].addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent e ) {
          int b = new Integer( e.getActionCommand() ).intValue();
          move( b );
        }
      } );
      pan.add( sq[ p ] );
    }
  }
```

```
public void init() {
  sandmark.watermark.ct.trace.Annotator.sm$mark();
  setBackground( Color.green );
  setForeground( Color.yellow );
  setFont( new Font("SansSerif", Font.BOLD, 60 ) );
  setSize( 360, 360 );
  setLayout( new BorderLayout() );
  addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent e ){
      dispose();
      System.exit( 0 );
    }
  } );
  add ( "North",  lab );
  add ( "Center", pan );
  pan.setLayout( new GridLayout( 3, 3 ) );
  sandmark.watermark.ct.trace.Annotator.sm$mark();
  south.setLayout( new FlowLayout() );


  quit.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      dispose();
      System.exit( 0 );
    }
  } );
  reload.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      clear();
    }
  } );
  add( "South", south );
  south.add( quit );
  south.add( reload );
}

public static void main( String[] args ) {
  TTT ttt = new TTT();
  ttt.init();
  ttt.start();
  ttt.pack();
  ttt.show();
}

public void clear() {
  setForeground( Color.yellow );
  for ( int p = 0; p  9; p ++ )
    sq[ p ].setLabel( "." );
  sandmark.watermark.ct.trace.Annotator.sm$mark();
  lab.setText("Tic-Tac-Toe");
}

public void move( int b ) {
  if ( !hit() && free() ) {
    ply = ( ply == 2 )? 1 : 2;
    sandmark.watermark.ct.trace.Annotator.sm$mark(ply);
    mark( b );
```

```
      if ( hit() ) { setForeground( Color.blue );
      lab.setText ("Player " + ply + " won!");
      }
    }
    else {
      if ( free() ) setForeground( Color.magenta );
      else          setForeground( Color.red );
      lab.setText ("Reload game!");
    }
  }

  public boolean free() {
     boolean f = false;
     for( int b = 0; b  9; b++ )
        if  ( sq[ b ].getLabel() == "." ) f = true;
     return f;
  }

  public boolean hit() {
    boolean hit = singleHit(0,4,8) | singleHit(2,4,6);
    for(int p = 0; p3; p++) hit |= singleHit(p,p+3,p+6);
    for(int p = 0; p9; p+=3) hit |= singleHit(p,p+1,p+2);
    return hit;
  }

  public void mark( int b ) {
    if ( sq[ b ].getLabel() == "." ) {
      if ( ply == 1 )
        sq[ b ].setLabel( "X" );
      else
        sq[ b ].setLabel( "O" );
      sandmark.watermark.ct.trace.Annotator.sm$mark(b);
    }
  }

  private boolean singleHit( int b1, int b2, int b3 ) {
    return ( ( sq[ b1 ].getLabel() != "." ) &&
             ( sq[ b1 ].getLabel() == sq[ b2 ].getLabel() ) &&
             ( sq[ b1 ].getLabel() == sq[ b3 ].getLabel() ) );
  }
}
```

This program should be compiled and packaged into a jar-file, and then traced from SandMark's tracing pane:

Where to Annotate

A number of annotation points should be scattered along the recognition path. As the number of points increases, the amount of code added at each point decreases. (It's not at all clear what the best tradeoff is in terms of stealth.)

Annotation points should not be placed at performance-critical points, because the added code will slow things down.

Annotation points should be placed where only the input (and not the environment or other random factors) controls whether the points are reached. For example, no annotation point should be placed on a code path that is conditionalized on having (or lacking) a particular type of network connection.

Annotation points can pass a string or numeric value, in which case the value is referenced by the generated code. This further muddles analysis and makes watermark recognition dependent on the correct values.

These annotation calls can be made by the application:

```
sandmark.watermark.ct.trace.Annotator.sm$mark()
        marks an annotation point with no parameters

sandmark.watermark.ct.trace.Annotator.sm$mark(String s)
        marks an annotation point parameterized by a string value

sandmark.watermark.ct.trace.Annotator.sm$mark(long v)
        marks an annotation point parameterized by a char, byte,
        short, int, or long value
```

Tracing in SandMark

Before you can actually embed a watermark into your program you have to produce a *trace*. Simply put, you have to

1. choose an input sequence to your program,
2. keep this input sequence secret and secure, since only with this sequence can the watermark be retrieved, and
3. enter the input sequence into your program, as it is being run from SandMark's **TRACE** pane.

As a result, a *trace* file is produced. This trace file becomes the input to the watermark *embedder* which you access from SandMark's **EMBED** tab. When, at a later date, you want to extract the watermark from the program you go to SandMark's **RECOGNIZE** tab. There, you run the program again with the same command-line arguments and the same secret input sequence, and the watermark/fingerprint will reappear.

Embedding in SandMark

Once you have generated a trace file from SandMark's **TRACE** tab you can start embedding watermarks. You can either provide your own watermarks or generate a random numeric watermark.

The output of the embedder is a new watermarked jar-file **prog_*wm*.jar**. Several other files are also generated:

- `Watermark.dot` is the watermark graph (in the dot format) embedded into the application.
- `TraceForest*.dot` are the call-graphs generated during tracing.
- `Watermark.java` is the class file that contains the watermark-building methods.

It is prudent to save these files for any future legal proceedings.

Example

Here is a view of SandMark embedding the watermark `WILDCATS` in the TTT application shown

in here : Note that the graphs that SandMark generates can be viewed by clicking on the `Graphs` button.

Configuration

Several options steer the embedding of the mark. In most cases you can leave the defaults in place.

- `Storage Policy`

  Either 'root' or 'all'. 'root' means that only roots of subgraphs are stored globally (or passed around in formal parametrs. 'all' means that all graph nodes are stored.

- `Storage Method`

  A colon-separated list of 'vector', 'array', 'pointer', and 'hash'. These are the types of storage containers in which subgraph nodes are stored.

  1. 'vector' means 'java.util.Vector'
  2. 'array' means 'Watermark[]'.
  3. 'hash' means 'java.util.Hashtable.
  4. 'pointer' means 'Watermark n1,n2,....

NOTE: 'pointer' currently doesn't work!

- `Storage Location`

  Either 'global' or 'formal'.

  1. 'global' means that subgraph nodes are stored in global static variables.
  2. 'formal' means that subgraph nodes are passed around in method parameters.

- `Protection Method`

  Colon-separated list of 'if', 'safe', 'try'.

  1. 'if' means that we protect against null pointers using 'if(n!=null)...'.
  2. 'safe' means that we use 'n=(n!=null)?n:new Watermark'.
  3. 'try' means that we use 'try{...}catch(...){}'.

- `Graph Type`

  Select the type of graph to use to encode the watermark:

  1. '*' means that we choose an encoding randomly.
  2. `RadixGraph`s have a high data rate.
  3. `PlantedPlaneCubicTree`s have some error-correction properties.
  4. `PermuationGraph` are similar to `RadixGraph`s.
  5. `ReduciblePermutationGraph`s have a low data rate but high resilience to attack.

- `Subgraph Count`

  An integer describing the number of subgraphs the graph should be broken up into.

- `Dump Intermediate Code`

  Print out the intermediate code.

- `Numeric Watermark`

  Pure numeric watermarks are encoded more efficiently than watermarks that can be arbitrary strings. NOTE: If you check this you must also check the same box during recognition.

- `Use Cycle Graph`

  To protect against node-splitting attacks, transform the underlying graph such that every node becomes a 3-cycle. Any node split will just expand the length of the cycle. During recognition, the cycles are contracted to generate the original graph.

- `Inline Code`

  Inline the methods for creating the watermark.

- `Replace Watermark Class`

  Choose a class that would best represent a watermark graph node rather than creating a new class from scratch.

- `Dump Intermediate Code`

  Print out the intermediate code used to generate the watermark class.

  If you are able to obtain a copy of your program that you believe has been illegally copied, you run a watermark recognizer to extract the watermark/fingerprint from the program.

Example

Here is a view of SandMark recognizing the watermark `WILDCATS` in the TTT application shown in here :

Note that the graphs that SandMark generates can be viewed by clicking on the `Graphs` button:

Configuration

Several options steer the recognition of the mark. In most cases you can leave the defaults in place.

- `Graph Type`

  Select the type of watermark graph we look for.

1. '*' means that we will look for every type of graph.
2. Selecting one of `RadixGraph`s, `PlantedPlaneCubicTree`s, `PermuationGraph`, `ReduciblePermutationGraph`s, means we will only look for this type of graph.

- `Numeric Watermark`

  Pure numeric watermarks are encoded more efficiently than watermarks that can be arbitrary strings. NOTE: If you check this you must also have checked the same box during embedding.

- `Use Cycle Graph`

  To protect against node-splitting attacks, the underlying graph has been expanded such that every node becomes a 3-cycle. Any node split will just expand the length of the cycle. During recognition, the cycles are contracted to generate the original graph.