On detection methods and analysis of malware

Jean-Yves Marion
LORIA
On detection methods and analysis of malware

1. A quick tour of Malware detection methods
2. Behavioral analysis using model-checking
3. Cryptographic function identification
What is a malware?
What is a malware?

- A malware is a program which has malicious intentions
What is a malware?

- A malware is a program which has malicious intentions

- A malware is a virus, a worm, a botnet...
What is a malware?

- A malware is a program which has malicious intentions
- A malware is a virus, a worm, a botnet...
- Giving a mathematical definition is difficult
What is a malware?

- A malware is a program which has malicious intentions
- A malware is a virus, a worm, a botnet ...
- Giving a mathematical definition is difficult

- How to protect a system?
What is a malware?

- A malware is a program which has malicious intentions
- A malware is a virus, a worm, a botnet ...
- Giving a mathematical definition is difficult
  - How to protect a system?
  - How to detect a malware?
What is a malware?

- A malware is a program which has malicious intentions
- A malware is a virus, a worm, a botnet ...
- Giving a mathematical definition is difficult
  - How to protect a system?
  - How to detect a malware?
Code protection

Detection is hard because malware are protected

1. Obfuscation
2. Cryptography
3. Self-modification
4. Anti-analysis tricks

Win32.Swizzor Packer displayed by IDA
A common protection scheme for malware

layer 1
A common protection scheme for malware
A common protection scheme for malware
A common protection scheme for malware

layer 1

layer 2

Decryption process:

Slide à remplacer?
A common protection scheme for malware

layer 1

Decrypt

layer 2

Decrypt

Slide à remplacer ?
A common protection scheme for malware

Slide à remplacer?

layer 1

layer 2

Decrypt

Decrypt

Decrypt

jeudi 1 novembre 2012
A common protection scheme for malware
A common protection scheme for malware
Packer protections

Different code waves with their relations

Theyda packer

Hostname pack`e avec Themida

Exemple (4/5)
Malware detection by string scanning

• Signature is a regular expression denoting a sequence of bytes

Pros:
• Accuracy: low rate of false positive
  ➔ programs which are not malware are not detected
• Efficient: Fast string matching algorithm
  ➔ Karp & Rabin, Knuth, Morris & Pratt, Boyer & Moore
Malware detection by string scanning

• Signature is a regular expression denoting a sequence of bytes

Pros :
• Accuracy: low rate of false positive
  ➡ programs which are not malware are not detected
• Efficient: Fast string matching algorithm
  ➡ Karp & Rabin, Knuth, Morris & Pratt, Boyer & Moore

Cons :
• Signature are quasi-manually constructed
• Signatures are not robust to malware protections
  ➡ Mutations, Code obfuscations, ...
• Static analysis of binary is very difficult
Detection by integrity check

- Identify a file using a hash function

<table>
<thead>
<tr>
<th>Files</th>
<th>Hash function</th>
<th>Hash numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

Numerical fingerprints
Detection by integrity check

- Identify a file using a hash function

**Cons:**
- File systems are updated, so numerical fingerprints change
- Difficult to maintain in practice
- Files may change with the same numerical fingerprint (due to hash fct)
Behavioral detection

- Identification of a sequence of actions:
  - System calls or library calls, Network interactions, ...
Behavioral detection

- Identification of a sequence of actions:
  - System calls or library calls, Network interactions, ...
- Two approaches
Behavioral detection

- Identification of a sequence of actions:
  - System calls or library calls, Network interactions, ...

- Two approaches
  - Anomaly Detection from a set of normal behaviours
Behavioral detection

• Identification of a sequence of actions:
  • System calls or library calls, Network interactions, ...

• Two approaches
  • Anomaly Detection from a set of normal behaviours
  • Detection from a set of potential malicious behaviours
Behavioral detection

- Identification of a sequence of actions:
  - System calls or library calls, Network interactions, ...

- Two approaches
  - Anomaly Detection from a set of normal behaviours
  - Detection from a set of potential malicious behaviours

Cons:
- Difficult to have a set of normal or bad behaviours
- Difficult to maintain in practice
- Functional obfuscations:
Behavioral detection

• Identification of a sequence of actions:
  • System calls or library calls, Network interactions, ...

• Two approaches
  • Anomaly Detection from a set of normal behaviours
  • Detection from a set of potential malicious behaviours

Cons:
• Difficult to have a set of normal or bad behaviours
• Difficult to maintain in practice
• Functional obfuscations:

Two ways of writing into a file

```
h=fopen(C:\windows\sys.dll);fwrite("test",h)
```

```
h=createFile(C:\windows\sys.dll);writeFile(h,"test")
```

✓ Several possible implementations of a high level action
## Anti-virus tests against unknown threats

Source: A study of anti-virus response to unknown threats by C. Devine and N. Richaud (EICAR 2009)

<table>
<thead>
<tr>
<th>Product name</th>
<th>testA01</th>
<th>testA02</th>
<th>testA03</th>
<th>testA11</th>
<th>testA12</th>
<th>testA13</th>
</tr>
</thead>
<tbody>
<tr>
<td>avast!</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
</tr>
<tr>
<td>AVG</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
</tr>
<tr>
<td>Avira</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
</tr>
<tr>
<td>BitDefender</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
</tr>
<tr>
<td>ESET</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
<td>No alert; keys logged.</td>
</tr>
</tbody>
</table>

[U] testA01: The GetRawInputData() API was introduced in Windows XP to access input devices at a low level, mainly for DirectX-enabled games. This function was documented in 2008 on the Firewall Leak Tester [6] web site.

[U] testA02 installs a WH_KEYBOARD_LL windows hook to capture all keyboard events (contrary to the WH_KEYBOARD hook, it does not inject a DLL into other processes).

[U] testA03: The GetAsyncKeyState() API allows querying the state of the keyboard asynchronously.

[A] testA11 hooks the keyboard driver’s IRJ_MJ_READ function.

[A] testA12 hooks the keyboard driver’s Interrupt Service Request.

[A] testA13 installs a “chained” device driver which places itself between the keyboard driver and upper level input device drivers.
## Versions of anti-virus software

<table>
<thead>
<tr>
<th>Product name</th>
<th>Version tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>avast! professional edition</td>
<td>4.8.1296</td>
</tr>
<tr>
<td>AVG Internet Security</td>
<td>8.0.200</td>
</tr>
<tr>
<td>Avira Premium Security Suite</td>
<td>8.2.0.252</td>
</tr>
<tr>
<td>BitDefender Total Security 2009</td>
<td>12.0.11.2</td>
</tr>
<tr>
<td>ESET Smart Security (NOD32)</td>
<td>3.0.672.0</td>
</tr>
<tr>
<td>F-Secure Internet Security 2009</td>
<td>9.00 build 149</td>
</tr>
<tr>
<td>Kaspersky Anti-Virus For Windows Workstations</td>
<td>6.0.3.837</td>
</tr>
<tr>
<td>McAfee Total Protection 2009</td>
<td>13.0.218</td>
</tr>
<tr>
<td>Norton 360 Version 2.0</td>
<td>2.5.0.5</td>
</tr>
<tr>
<td>Panda Internet Security 2009</td>
<td>14.00.00</td>
</tr>
<tr>
<td>Sophos Anti-Virus &amp; Client Firewall</td>
<td>7.6.2</td>
</tr>
<tr>
<td>Trend Micro Internet Security Pro</td>
<td>17.0.1305</td>
</tr>
</tbody>
</table>
AV industry in 1998

AV industry in 2008

Image Copyright: IKARUS Security Software GmbH
On detection methods and analysis of malware

1. A quick tour of Malware detection methods
2. Behavioral analysis using model-checking
3. Cryptographic function identification

Joint work with Philippe Beaucamps and Isabelle Gnaedig
Esorics 2012
```c
void scan_dir(const char* dir) {
    HANDLE hFind;
    char szFilename[2048];
    WIN32_FIND_DATA findData;

    sprintf(szFilename, "%s\\%s", dir, ".*.");
    hFind = FindFirstFile(szFilename, &findData);
    if (hFind == INVALID_HANDLE_VALUE) return;

    do {
        sprintf(szFilename, "%s\\%s", dir, findData.cFileName);
        if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
            scan_dir(szFilename);
    } while (FindNextFile(hFind, &findData));
    FindClose(hFind);
}

void main(int argc, char** argv) {
    HANDLE hIcmp;
    const char* icmpData = "Babcdef..."
    char reply[128];

    hIcmp = IcmpCreateFile();
    for (int i = 0; i < 2; ++i)
        IcmpSendEcho(hIcmp, ipaddr, icmpData, 10, NULL, reply, 128, 1000);
    IcmpCloseHandle(hIcmp);

    SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in sin = {AF_INET, ipaddr, htons(139)/
      Netbios
    };
    if (connect(s, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR) {
        ...
    }

    char buffer[1024];
    GetLogicalDriveStrings(sizeof(buffer), buffer);
    const char* szDrive = buffer;
    while (*szDrive) {
        if (GetDriveType(szDrive) == DRIVE_FIXED)
            scan_dir(szDrive);
        szDrive += strlen(szDrive) + 1;
    }
}
```
Low level Traces

```c
void scan_dir(const char* dir) {
    HANDLE hFind;
    char szFilename[2048];
    WIN32_FIND_DATA findData;

    sprintf(szFilename, "%s\%s", dir, "*.*");
    hFind = FindFirstFile(szFilename, &findData);
    if (hFind == INVALID_HANDLE_VALUE) return;
    do {
        sprintf(szFilename, "%s\%s", dir, findData.cFileName);
        if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
            scan_dir(szFilename);
        else {
            ...}
    } while (FindNextFile(hFind, &findData));
    FindClose(hFind);
}

void main(int argc, char** argv) {
    HANDLE hIcmp;
    const char* icmpData = "Babcdef ...";
    char reply[128];

    hIcmp = IcmpCreateFile();
    for (int i = 0; i < 2; ++i)
        IcmpSendEcho(hIcmp, ipaddr, icmpData, 10, NULL, reply, 128, 1000);
    IcmpCloseHandle(hIcmp);

    SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in sin = {AF_INET, ipaddr, htons(139)/* Netbios */};
    if (connect(s, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR) {
        ...
    }

    char buffer[1024];
    GetLogicalDriveStrings(sizeof(buffer), buffer);
    const char* szDrive = buffer;
    while (*szDrive) {
        if (GetDriveType(szDrive) == DRIVE_FIXED)
            scan_dir(szDrive);
        szDrive += strlen(szDrive) + 1;
    }
}
```

/* Behavior pattern: ping of a remote host */
hlcmp = IcmpCreateFile();
for(int i = 0; i < 2; ++i)
    IcmpSendEcho(hlcmp, ipaddr, icmpData, 10, NULL, reply, 128, 1000);
IcmpCloseHandle(hlcmp);

/* Behavior pattern: Netbios connection */
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin = {AF_INET, ipaddr, htons(139)/* Netbios */};
if (connect(s, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR) {
    ...
}

/* Behavior pattern: scanning of local drives */
char buffer[1024];
GetLogicalDriveStrings(sizeof(buffer), buffer);
const char* szDrive = buffer;
while (*szDrive) {
    if (GetDriveType(szDrive) == DRIVE_FIXED)
        scan_dir(szDrive);
    szDrive += strlen(szDrive) + 1;
}
```

Trace are finite terms:
FindfirstFile(x,y).FindNextFile(z,x). FindNextFile(z,x).FindClose(z).
IcmpSendEcho(u,...).IcmpSendEcho(u,...).IcmpCloseHandle(u)....
Program behaviour

• The program behaviour is given by sequences of system calls
  - represented by a set $L$ of terms

• How to collect traces?
Program behaviour

• The program behaviour is given by sequences of system calls
  - represented by a set $L$ of terms

• How to collect traces?

  **Static analysis**
  • A good approximation of a set of execution traces
  • Good detection coverage
  • But static analysis is difficult to perform
Program behaviour

- The program behaviour is given by sequences of system calls
  - represented by a set $L$ of terms
- How to collect traces?

  **Static analysis**
  - A good approximation of a set of execution traces
  - Good detection coverage
  - But static analysis is difficult to perform

  **Dynamic analysis**
  - Collect an execution trace (with use PIN)
  - Monitor program interactions (sys calls, network calls, ...)
  - What is the detection coverage? partial behaviours ...
Trace abstraction
Trace abstraction

- Several ways to send a ping:
Trace abstraction

- Several ways to send a ping:

  1. Call the socket function with the parameter IPPROTO_ICMP and then call the sendto function with ICMP_ECHOREQ
Trace abstraction

- Several ways to send a ping:

1. Call the socket function with the parameter IPPROTO_ICMP and then call the sendto function with ICMP_ECHOREQ

2. Call the IcmpSendEcho function
Trace abstraction

- Several ways to send a ping:
  1. Call the socket function with the parameter IPPROTO_ICMP and then call the sendto function with ICMP_ECHOREQ
  2. Call the IcmpSendEcho function
- Abstract the ping behaviour by a predicate PING(x) to represent a ping on socket x
Trace abstraction

- Several ways to send a ping:

  1. Call the socket function with the parameter IPPROTO_ICMP and then call the sendto function with ICMP_ECHOREQ

  2. Call the IcmpSendEcho function

- Abstract the ping behaviour by a predicate PING(x) to represent a ping on socket x

- Define an abstraction relation R as a term rewrite system

\[
\begin{align*}
\text{socket}(x,u).\text{sendto}(x,v,y) & \rightarrow \text{socket}(x,u).\text{sendto}(x,v,y).\text{PING}(x) \\
\text{IcmpSendEcho}(x) & \rightarrow \text{IcmpSendEcho}(x).\text{PING}(x)
\end{align*}
\]

- We abstract/rewrite a pattern on a trace only once
Trace abstraction

- Several ways to send a ping:
  1. Call the socket function with the parameter IPPROTO_ICMP and then call the sendto function with ICMP_ECHOREQ
  2. Call the IcmpSendEcho function

- Abstract the ping behaviour by a predicate PING(x) to represent a ping on socket x

- Define an abstraction relation R as a term rewrite system

\[
\begin{align*}
\text{socket}(x,u).\text{sendto}(x,v,y) & \rightarrow \text{socket}(x,u).\text{sendto}(x,v,y).\text{PING}(x) \\
\text{IcmpSendEcho}(x) & \rightarrow \text{IcmpSendEcho}(x).\text{PING}(x)
\end{align*}
\]

- We abstract/rewrite a pattern on a trace only once

- As a result, we have a terminating and rational abstraction system

\[
\begin{align*}
\text{IcmpSendEcho}(u,...).\text{IcmpSendEcho}(u,...).\text{IcmpCloseHandle}(u) & \rightarrow \text{IcmpSendEcho}(u,...).\text{PING}(u).\text{IcmpSendEcho}(u,...).\text{IcmpCloseHandle}(u) \\
& \rightarrow \text{IcmpSendEcho}(u,...).\text{PING}(u).\text{IcmpSendEcho}(u,...).\text{PING}(u).\text{IcmpCloseHandle}(u)
\end{align*}
\]

- We keep the LHS to deal with complex patterns
Computation of Abstract Trace language

Abstract a trace language \( L \) by reducing it w.r.t. an abstraction relation \( R \)

\[
L \rightarrow \ldots \rightarrow L^\downarrow
\]

**Theorem:**
Let \( R \) be a rational abstraction relation and \( L \) be a trace language. If \( L \) is regular then so is \( L^\downarrow \)

- Based on tree automata methods

**Related work**
- Martignoni et al. 2008: multi-layered abstraction on a single trace
Behaviour patterns

- A behavior pattern is a First-order LTL (Linear temporal logic) formula

\[
\varphi_1 = \exists x, y. \text{socket} (x, \alpha) \land (\neg \text{closesocket} (x) \cup \text{sendto} (x, \beta, y))
\]

\[
\varphi_2 = \exists x. \text{IcmpSendEcho} (x)
\]

\[
\varphi_{\text{ping}} = \varphi_1 \lor \varphi_2
\]

Quantification domain is the finite set of parameter names

Let \( L \) be the behaviour of the program \( P \). If a trace \( t \) of \( L \) satisfies a behaviour pattern \( \varphi \), then \( P \) has the behaviour described by \( \varphi \)
Behaviour patterns

• A behavior pattern is a First-order LTL (Linear temporal logic) formula

\[ \varphi_1 = \exists x, y. \, \text{socket}(x, \alpha) \land (\neg \text{closesocket}(x) \cup \text{sendto}(x, \beta, y)) \]
\[ \varphi_2 = \exists x. \, \text{IcmpSendEcho}(x) \]
\[ \varphi_{\text{ping}} = \varphi_1 \lor \varphi_2 \]

Quantification domain is the finite set of parameter names

• Traces satisfying a FO-LTL formula are:

\[ B = \{ t \in T_{\text{Trace}}(\mathcal{F}_\Sigma) \mid t \models \varphi \} \]

Let L be the behaviour of the program P. If a trace t of L satisfies a behaviour pattern \( \varphi \), then P has the behaviour described by \( \varphi \)
Malicious behavior detection

**Theorem**: Let $L$ be a finite set of finite traces. Let $L^\downarrow$ be a trace correctly abstracted from a rational abstraction relation $R$. Let $\varphi$ be a FOLTL formula. Deciding whether deciding $L^\downarrow$ is infected by $\varphi$ is linear-time computable.

It works also when $L$ is regular (and infinite), see the paper for details

**Related work**

- Jacob et al., 2009: low-level functionalities, exponential-time detection
A C Keylogger or a sms message leaking app

CHAPITRE 2. ANALYSE COMPORTEMENTALE STATIQUE

Ainsi, dans cette section, nous prenons l'exemple d'un programme capturant les caractères entrés au clavier (afin d'en extraire notamment mots de passe, informations bancaires et autres données sensibles). L'analyse sous IDA de sa forme compilée permet de déterminer la structure de la pile et de l'espace des variables la composant. Par simplicité, nous travaillons sur son code source :

```c
LRESULT WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    RAWINPUTDEVICE rid;
    RAWINPUT *buffer;
    UINT dwSize;
    USHORT uKey;

    switch(msg) {
    case WM_CREATE: /* Creation de la fenetre principale */
        /* Initialisation de la capture du clavier */
        rid.usUsagePage = 0x01;
        rid.usUsage = 0x06;
        rid.dwFlags = RIDEV_INPUTSINK;
        rid.hwndTarget = hwnd;
        RegisterRawInputDevices(&rid, 1, sizeof(RAWINPUTDEVICE));
        break;

    case WM_INPUT: /* Evenement clavier, souris, etc. */
        /* Quelle taille pour buffer ? */
        GetRawInputData( (HRAWINPUT) lParam, RID_INPUT, NULL, &dwSize, sizeof(RAWINPUTHEADER) );
        buffer = (RAWINPUT*) malloc(dwSize);
        /* Recuperer dans buffer les donnees capturees */
        if(!GetRawInputData( (HRAWINPUT) lParam, RID_INPUT, buffer, &dwSize, sizeof(RAWINPUTHEADER) ))
            break;

        if(buffer->header.dwType == RIM_TYPEKEYBOARD &
            buffer->data.keyboard.Message == WM_KEYDOWN) {
            printf("%c\n", buffer->data.keyboard.VKey);
        }
        free(buffer);
        break;
    }
    return 0;
}
```

Ce code contient sept variables : hwnd, msg, wParam, lParam, rid, buffer et dwSize. Le types HWND et HRAWINPUT représentent des entiers. Les types WPARAM et LPARAM sont interprétés différemment selon le contexte : ici, seule la variable lParam est utilisée et représente un entier dans le contexte de son utilisation. Les types RAWINPUTDEVICE et RAWINPUT sont des structures décrites ci-après. Seuls les champs nous intéressant ont été conservés.

```c
struct RAWINPUTDEVICE {
    int  usUsagePage;
    int  usUsage;
    ...
};
```

CHAPITRE 3. AUTOMATES DE TRACES

```java
public void onReceive(Context context, Intent intent) {
    Bundle bundle;
    Object pdus[];

    String from = null;
    String msg = "";
    String str = "";

    bundle = intent.getExtras();
    pdus = (Object[])bundle.get("pdus");

    // Pour chaque message envoye
    int pdus_len = pdus.length;
    for(int i = 0; i < pdus_len; i++)
    {
        Object pdu = pdus[i];
        SmsMessage smssmessage = SmsMessage.createFromPdu((byte[])pdu);
        // from = "From:" + smssmessage.getDisplayOriginatingAddress() +
        String sb1 = new StringBuilder("From:");
        String s1 = smssmessage.getDisplayOriginatingAddress();
        sb1.append(s1);
        sb1.append(":");
        from = sb1.toString();
    }
}
```

jeudi 1 novembre 2012
An information leaking behaviour pattern

\[ M := \exists x. \lambda_{\text{steal}}(x) \land \neg \lambda_{\text{inval}}(x) \cup \lambda_{\text{leak}}(x) \]

Keystroke or IMEI capture

invalidated

Network send functionality

\[ \lambda_{\text{steal}}(x) := \text{GetAsyncKeyState}(x) \lor \\
(\text{RegisterDev}(\text{KBD, SINK}) \odot \text{GetInputData}(x, \text{INPUT})) \lor \\
(\exists y. \text{SetWindowsHookEx}(y, \text{WH KEYBOARD_LL}) \land \\
\neg \text{UnhookWindowsHookEx}(y) \cup \text{HookCalled}(y, x)) \lor \\
\exists y. \text{TelephonyManager_getDeviceId}(x, y). \]

\[ \lambda_{\text{inval}}(x) := \text{free}(x) \lor \exists y. \text{sprintf}(x, y) \lor \\
\text{GetInputData}(x, \text{INPUT}) \lor \ldots \]

\[ \lambda_{\text{leak}}(x) := \exists y, z. \text{sendto}(z, x, y) \lor \\
\exists y, z. (\text{connect}(z, y) \land \neg \text{close}(z) \cup \text{send}(z, x)) \lor \\
\exists c, s. \text{HttpURLConnection_getOutputStream}(s, c) \land \\
\neg \text{OutputStream_close}(s) \cup \text{OutputStream_write}(s, x) \]
Information Leak Behaviors

Abstraction can be applied to detection of generic threats, and in particular to detection of sensitive information leak. Such a leak can be decomposed into two steps: capturing sensitive information and sending this information to an exogenous location. The captured data can be keystrokes, passwords or data read from a sensitive network location, while the exogenous location can be the network, a removable device, etc. Thus, we define a behavior pattern $\lambda_{\text{steal}}(x)$, representing the capture of some sensitive data $x$, and a behavior pattern $\lambda_{\text{leak}}(x)$, representing the transmission of $x$ to an exogenous location. Moreover, since the captured data must not be invalidated before being leaked, we define a behavior pattern $\lambda_{\text{inval}}(x)$, which represents such an invalidation.

Finally, the captured data is usually not leaked in its raw form, so we take into account transformations of this data via the behavior pattern $\lambda_{\text{depends}}(x, y)$ which denotes a dependency of $x$ on $y$. For instance, $x$ may be a string representation of $y$, or $x$ may be an encryption or an encoding of $y$.

Then, in order to account for one such transformation of the stolen data, we define the information leak abstract behavior:

$$M := \exists x, y. \lambda_{\text{steal}}(x) \land \neg \lambda_{\text{inval}}(x) \mathop{U} \lambda_{\text{depends}}(y, x) \land \mathop{U} \lambda_{\text{leak}}(y).$$

Related work
Kinder & al, Detecting malicious code by model checking
Abstraction based analysis of malware behaviours

Our works

- Expressing set of traces by regular term languages
- Compute an higher level semantics of traces by term rewriting systems
- Keeping track of parameters
- Expressing Behavior patterns by FOLTL formulas
- Testing whether abstract traces satisfy a FOLTL-behavior pattern
- Efficient analysis (quasi-linear time wrt several restrictions)
A first conclusion

- Detection of malicious behaviors:
  - Our approach is difficult and time-consuming to implement in practice.
  - We made only a few experiments Allaple, Rbot, Acore, Mimail and a keylogger for Android
  - Detection of malware is a difficult subject and a reason is

A problem is the absence of high level abstraction to structure and understand obfuscated codes.

Related works
- Chrisdorescu, Song & al 2007: Semantics-Aware Malware detection
On detection methods and analysis of malware

1. A quick tour of Malware detection methods
2. Behavioral analysis using model-checking

3. Cryptographic function identification

Joint work with Joan Calvet, José M. Fernandez
CCS 2012
Cryptographic function identification in obfuscated binary programs

Joint work with Joan Calvet, José M. Fernandez

CCS 2012
Identification of cryptographic functions

Example:
Win32.Sality.AA

```
push esi
pop esi
and ecx, 766F1C80h
test ebx, eax
sub eax, 68FBh
xadd ecx, ecx
mov ecx, 3E676485h
bts ecx, 60h
sub eax, 0CEEEDh
push edx
pop edx
shl ecx, 1
sub eax, 5351h
test ebx, eax
bsf ecx, eax
sub eax, 5C586h
push esp
pop esp
inc ecx
shld ecx, eax, cl
push esi
pop esi
sub eax, 2E79h
push edi
mov ecx, 0C6FEEC90h
pop ecx
and ecx, edi
sub eax, 540h
bsf ecx, eax
bts ecx, eax
mov ecx, 0BE572485h
push ecx
pop ecx
sub eax, 100DF39Ah
```

```
mmov eax, eax
cmp al, 77h
mmov esp, esp
sub edi, 0BF59686h
xchg ebx, ebx
bts ecx, 54h
inc ecx
push ebx
nop
pop ebx
and ecx, edi
push eax
pop eax
push esi
pop ecx
jmp loc_40FF6E
```

```
xor [edi], al
```

Decryption?

No API Calls and function names

Not far from the program entry point, in the first code layer...
Is the previous code by any chance an implementation of a known cryptographic algorithm?
Is the previous code by any chance an implementation of a known cryptographic algorithm?

Answering this question affirmatively would provide to the analyst a high-level description of this code, without studying it line-by-line!
Is the previous code by any chance an implementation of a known cryptographic algorithm?

Answering this question affirmatively would provide to the analyst a high-level description of this code, without studying it line-by-line!

The general questions are

• How to determine the meaning of a piece of code?
• How to determine the meaning of an execution trace?
  What is computed?
The problem

Set of known cryptographic functions $S$

Does $P$ contain any known functions of $S$?

jeudi 1 novembre 2012
The problem

Set of known cryptographic functions $S$

Does $P$ contain any known functions of $S$?

Just from an execution trace
The problem

Does \( P \) contain any known functions of \( S \)?

Just from an execution trace

Proving a *general semantic equivalence* between a function of \( P \) and one of the \( S \) functions seems difficult.
Existing approaches

• A common way to locate cryptographic code is to calculate the ratio of arithmetic machine instructions (ADD, SUB, XOR...).

• When this ratio is superior to a certain threshold, it indicates cryptographic code.
In Our Sality Sample...

... every basic block looks like those.
Our approach
Our approach

1. To observe an execution of P
Our approach

1. To observe an execution of P

2. To collect input-output values used during this execution, that is a set of $(x,y)$ such that $[P(x)]=y$
Our approach

1. To observe an execution of P

2. To collect input-output values used during this execution, that is a set of (x,y) such that $P(x) = y$

3. To check if one $F$, or more function(s), of $S$ satisfies $F(x) = y$
Our approach

1. To observe an execution of $P$

2. To collect input-output values used during this execution, that is a set of $(x,y)$ such that $P(x) = y$

3. To check if one $F$, or more function(s), of $S$ satifies $F(x) = y$

If yes, we conclude that $P$ behaves as an implementation of $F$ (in the values $(x,y)$).
Our approach

1. To observe an execution of \( P \)

2. To collect input-output values used during this execution, that is a set of \((x,y)\) such that \( [P(x)] = y \)

3. To check if one \( F \), or more function(s), of \( S \) satisfies \( F(x) = y \)

If yes, we conclude that \( P \) behaves as an implementation of \( F \) (in the values \((x,y))\).

But, roughly (...), in the cryptographic case:

There is a unique (with high probability) cryptographic function \( K \) such \( K(x) = y \) where \( x \) is a ciphered text, \( y \) is the deciphered.

One point should be enough to interpolate a cryptographic function
Obfuscation

Implementing this simple reasoning in obfuscated binary programs is non trivial...

... and this is our focus in this project!
Obfuscation

Implementing this simple reasoning in *obfuscated* binary programs is non trivial...

... and this is our focus in this project!

- Where are I/O parameters?
Obfuscation

Implementing this simple reasoning in obfuscated binary programs is non trivial...

... and this is our focus in this project!

- Where are I/O parameters?
- Where are functions?
Obfuscation

Implementing this simple reasoning in **obfuscated** binary programs is non trivial...

... and this is our focus in this project!

- Where are I/O parameters?
- Where are functions?

... there are no such things as function calls.

Never returns!

There is no high level definition
Implementation
1. Information gathering:
Implementation

1. **Information gathering:**
   - We collect an execution trace of P:
Implementation

1. Information gathering:

   We collect an execution trace of $P$:

   For each run instruction, we gather
Implementation

1. **Information gathering:**

   - We collect an execution trace of P:
     
     For each run instruction, we gather
     
     a) its memory address
Implementation

1. **Information gathering:**
   - We collect an execution trace of $P$:
     - For each run instruction, we gather
       a) its memory address
       b) its machine instruction
Implementation

1. **Information gathering:**

   - We collect an execution trace of P:
     
     For each run instruction, we gather

     a) its memory address

     b) its machine instruction

     c) its access to memory, registers and the values
Implementation

1. **Information gathering:**
   
   - We collect an execution trace of $P$:
     
     For each run instruction, we gather
     
     a) its memory address
     
     b) its machine instruction
     
     c) its access to memory, registers and the values

2. **Extraction:**
Implementation

1. **Information gathering:**
   - We collect an execution trace of $P$:
     - For each run instruction, we gather
       a) its memory address
       b) its machine instruction
       c) its access to memory, registers and the values

2. **Extraction:**
   - Delimit possible cryptographic code in the execution trace.
Implementation

1. **Information gathering:**
   - We collect an execution trace of $P$:
     - For each run instruction, we gather
       - a) its memory address
       - b) its machine instruction
       - c) its access to memory, registers and the values

2. **Extraction:**
   - Delimit possible cryptographic code in the execution trace.

3. **Identification:**

jeudi 1 novembre 2012
Implementation

1. **Information gathering:**
   - We collect an execution trace of P:
     - For each run instruction, we gather
       - a) its memory address
       - b) its machine instruction
       - c) its access to memory, registers and the values

2. **Extraction:**
   - Delimit possible cryptographic code in the execution trace.

3. **Identification:**
   - Check if the extracted code maintained during the previous execution the input-output relationship of a known cryptographic function.
Loop extraction

• Cryptographic algorithms usually apply a *same treatment* on their input-output parameters.

• It makes **loops** a cryptographic code feature and a possible criterion to extract it from execution traces.

• But there are loops everywhere, not only in crypto algorithms...

  What kind of loops are we looking for?
Loop extraction

• Cryptographic algorithms usually apply a same treatment on their input-output parameters.

• It makes loops a cryptographic code feature and a possible criterion to extract it from execution traces.

• But there are loops everywhere, not only in crypto algorithms...

What kind of loops are we looking for?

Win32.Mebroot

Unrolling optimization

- Unrolling optimization
  - 0x00 inc eax
  - 0x01 inc ebx
  - 0x02 mov [ebx], eax
  - 0x03 inc eax
  - 0x04 inc ebx
  - 0x05 mov [ebx], eax
  - 0x06 inc eax
  - 0x07 inc ebx
  - 0x08 mov [ebx], eax
A loop definition

- We look for the same operations applied repeatedly on a set of data.

**Our definition:** “A loop is the repetition of a same sequence of machine instructions at least two times.”

### Execution Trace

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>add</td>
<td>ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub</td>
<td>edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec</td>
<td>dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz</td>
<td>0x401325</td>
</tr>
<tr>
<td>401325</td>
<td>add</td>
<td>ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub</td>
<td>edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec</td>
<td>dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz</td>
<td>0x401325</td>
</tr>
</tbody>
</table>

...
A loop definition

- We look for the **same operations applied repeatedly** on a set of data.

**Our definition:** “A loop is the repetition of a same sequence of machine instructions at least two times.”

### Execution Trace

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
</tbody>
</table>

Iteration 1

Iteration 2
A loop definition

- We look for **the same operations applied repeatedly** on a set of data.

**Our definition:** “A loop is the repetition of a same sequence of machine instructions at least two times.”

**Execution Trace**

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
</tbody>
</table>

Iteration 1

Iteration 2

Loop
A loop definition

- We look for **the same operations applied repeatedly** on a set of data.

**Our definition:** “A loop is the repetition of a same sequence of machine instructions at least two times.”

**Execution Trace**

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
</table>
| ...
401325      | add ebx, edi
401327      | sub edx, ebx
401329      | dec dword ptr [ebp+0xc]
40132c      | jnz 0x401325
401325      | add ebx, edi
401327      | sub edx, ebx
401329      | dec dword ptr [ebp+0xc]
40132c      | jnz 0x401325
...         | ...

*It corresponds to the language $L=\{ww\}$, which is non-context free...*
What About Nested Loops?

Simplified CFG

A

B

C

Simplified CFG
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace

Loop B
3 iterations

Loop B
2 iterations
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Different at each iteration!

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace

Trace Rewriting

Ok!
What About Nested Loops?

Simplified CFG

Execution trace

Trace Rewriting

Ok!
Loop Detection Algorithm

1. Detects two repetitions of a loop body in the execution trace. 
   (non trivial, non-context free language)
2. Replaces in the trace the detected loop by a symbol representing their body.
3. Goes back to step 1 if new loops have been detected.
We extracted possible cryptographic code from execution traces thanks to a particular loop definition.
We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

But our identification method needs the input-output values of this crypto code.
We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

But our identification method needs the input-output values of this crypto code.
We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

But our identification method needs the input-output values of this crypto code.

How can we define such input-output parameters from the bytes read and written in execution traces?
Distinction between input and output bytes in the execution trace:

- **Input** bytes have been read *without having been previously written*.
- **Output** bytes have been *written*.

A reasonable hypothesis
Identification (3)

Loop parameters

Grouping of several bytes into the same parameter:

1. If they are adjacent in memory *(too large!)*
2. And if they are manipulated by the same instruction in the loop body.

Iteration 1

```
add [ecx], edi
mov eax, [ebx]
...
```

Iteration 2

```
add [ecx], edi
mov eax, [ebx]
...
```
Loop Data Flow

• A crypto implementation can contain several loops.
Loop Data Flow

• A crypto implementation can contain several loops.

• We consider that two loops $L_1$ and $L_2$ are in the same crypto implementation:
  – If $L_1$ started before $L_2$ in the trace.
  – If $L_2$ uses as an input parameter an output parameter of $L_1$. 
Loop Data Flow Graph (oriented, acyclic)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

*(in order to deal with algorithm combinations)*
Method Recap

1. We collect an execution trace.

2. We extract possible cryptographic algorithms with their parameter values.

3. We compare the input-output relationship with known cryptographic algorithms.

We can demonstrate that a program behaves like a known crypto algorithm during one particular execution.
Let’s illustrate this process on our Sality sample...
Step 1 : Gather Execution Trace
Step 1 : Gather Execution Trace
Step 1 : Gather Execution Trace
Step 1: Gather Execution Trace
Step 2: Recognize Loops on the Trace
Step 2 : Recognize Loops on the Trace

L1

L2

L3
Step 3: Define Loop I/O Parameters
Step 4 : Connect Loops With Data-Flow
Unknown Algorithm Extracted
We still have the last mile to do...
Comparison Algorithm

1. Build the set $I$ of possible input values with all possible orderings of $A$ input parameters.
2. Build the set $O$ of possible output values in the same manner with $A$ output parameters.
3. Evaluate each $S$ function on all values in $I$ and check if the result produced is in $O$.

If yes, this is a success!
**Input 1:** unknown algorithm $\mathcal{A}$ with its parameter values
**Input 1:** unknown algorithm $A$ with its parameter values

**Input 2:**

Finite set of known cryptographic functions $S$

Custom Cipher $X$

Custom Cipher $Y$

AES

RC4

MD5

TEA
Question
Question

• Some difficulties, for example:
  – **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  – **Parameter number**: we collect more than the cryptographic parameters.
In practice, the complexity of this algorithm can be greatly reduced with some simple rules:

– Do not consider memory addresses as valid parameters.
– Do not consider common constant values (0,FF) as valid parameters.
No particular ciphers selected, test with all of them: ['tea', 'xtea', 'russian_tea', 'rc4', 'aes_128_core', 'md5_core']

Testing LDF 1 ...
  Heuristic: Remove constants from parameters
  Comparison with TEA... Fail!
  Comparison with XTEA... Fail!
  Comparison with RussianTEA... Fail!
  Comparison with RC4...

!! Identification successful: RC4 encryption with:

Plain text (57066 bytes):
0xd2276d92e4cb5446342196edd2a011b9ae1ebbea51ceb7bc834d7626ffaa344f1f315d5b0ce29deb26db7cs63fb23d23ee034148acf63495ea0a117abc4d89834b51b5a4d062207d9589319917536618999a58f0c5e4d2ac49215809e5fa0c900eadf95d9e2b636e6f5d9eb9c94c1f5a2a877f77edef1f9f182d5123093909df3ec84656e30eda6d7cc88625d4c9476075c70e959cc912efc4126a9b5959c7e6de2099a96e3c136f63317ffddf7ebc3f4a88f9f33121ff7f850accf5b2e7278cc961372c5eff2712646ec51d6ec18bee4feb70c771ea577f7ec5bc73f1a769fd89f84c540ea1ec9fa563222d8919dd46e14b74ff56253f994709d0e...

Key (8 bytes): 0xb8a2b9a789850cea

Encrypted text (57066 bytes):
0xe800000000005d81ed05104000582daad20000895ae11400008bd73724000007519c7859d13400022222222c7858c1340003333333333e9820000003d3b64678b1e300085db780e8b5bcb8b5b1cb8b885b08b8eb0a8b5bb48d57b7c8b5cfc866813b4d5a7405e9420000008bf303763c813e5045000070450e93010000899d791340008d859013400050ffbf579134000ee49010000e81e0100008959d1340008d857d13400050ff5b579134000e82c010000e8010100008958c1340008d858f15400050ff959d1340008e90000008985791340008d122200000680280000ff95f71340008d771540005768008000006a06ea0...
More Results

<table>
<thead>
<tr>
<th></th>
<th>$B_1$</th>
<th>$B_2$</th>
<th>Storm*</th>
<th>$SBank^*$</th>
<th>$B_3$</th>
<th>$Sal^*$</th>
<th>$B_4$</th>
<th>$B_5^\dagger$</th>
<th>$B_6$</th>
<th>$B_7^\dagger$</th>
<th>$Wal^*$</th>
<th>$B_8^\dagger$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligot</td>
<td>TEA</td>
<td>TEA</td>
<td>RTEA</td>
<td>RTEA</td>
<td>RC4</td>
<td>RC4</td>
<td>AES</td>
<td>AES</td>
<td>MD5</td>
<td>MD5</td>
<td>AES, MD5</td>
<td>MOD MUL</td>
</tr>
<tr>
<td>CryptoSearcher</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AES</td>
<td>X</td>
<td>MD5</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Draca</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>MD5</td>
<td>SHA-1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Findcrypt2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AES</td>
<td>X</td>
<td>MD4</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>H&amp;C Detector</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>MD5</td>
<td>SHA-1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PEiD KANAL</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>AES</td>
<td>X</td>
<td>MD5</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Signsrch</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>TEA</td>
<td>X</td>
<td>X</td>
<td>AES</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SnDCryptoS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AES</td>
<td>X</td>
<td>MD5</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

(cf. paper)
Limitations

• As we analyze execution traces, we have to know how to exhibit interesting execution paths.

• The tool is as good as the reference implementation database.

• It is easy to bypass, like any program analysis technique.
Future Work

• Extract reference implementations directly from binary programs.

• Implement other extraction criteria than our loop model.

Code is available at

https://code.google.com/p/aligot/
High Security Lab @ Nancy.fr

Other subjects
- Morphological analysis
- Botnet counter-attacks

Telescope & honeypots
In vitro experiment clusters

Thanks!
BONUS SLIDES
Morphological analysis in a nutshell
Morphological analysis in a nutshell

Signatures are abstract flow graph
Morphological analysis in a nutshell

Signatures are abstract flow graph

Detection of subgraph in program flow graph abstraction
Automatic construction of signatures
Reduction of signatures by graph rewriting

<table>
<thead>
<tr>
<th>Original</th>
<th>One-one substitution</th>
<th>Substitution</th>
<th>Permutation</th>
<th>Jcc obfuscation</th>
<th>All in one</th>
<th>Normalised CFG</th>
</tr>
</thead>
</table>
Morphological detection : Results

- False negative
  - No experiment on unknown malware
  - Signatures with < 18 nodes are potential false negative
  - Restricted signatures of 20 nodes are efficient
- Less than 3 sec. for signatures of 500 nodes
Conclusion about morphological detection

- Benchmarks are good

- Pro
  - More robust on local mutation and obfuscation
  - Detect easily variants of the same malware family
  - Try to take into account program semantics
  - Quasi-automatic generation of signatures

- Cons
  - Difficult to determine flow graph statically of self-modifying programs
  - Use of combination of static and dynamic analysis
Reference


• Recon 2012 and Malware 2012
## Performances

<table>
<thead>
<tr>
<th></th>
<th>Sality 1</th>
<th>Sality 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Size (instructions)</td>
<td>~1M</td>
<td>~4M</td>
</tr>
<tr>
<td>Time To Trace</td>
<td>5mn</td>
<td>10mn</td>
</tr>
<tr>
<td>Time To Extract Crypto Algorit</td>
<td>4h</td>
<td>10h</td>
</tr>
<tr>
<td>Time To Identify</td>
<td>3mn</td>
<td>4mn</td>
</tr>
</tbody>
</table>

- The tool is just a PoC, no optimization at all.
- When the analysts knows where the algorithm is, it will reduce the trace size.
# Existing Tools For Crypto Identification

<table>
<thead>
<tr>
<th>Tools</th>
<th>Answers on Sality sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto Searcher</td>
<td>Ø</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>Ø</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>Ø</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>Ø</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>Ø</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
</tbody>
</table>