BinJuice Automata

Similarity analysis in binary (malware) code

FACE
Formal Avenue for Chasing malware
The problem
Anti-anti malware

* How can we escape signature checking?
  * ...by dynamically modifying malware structure!

* Polymorphic malware contain decryption routines which decrypt encrypted constant parts of their body.

* Metamorphic malware typically do not use encryption, but mutates (obfuscate) forms in subsequent generations.
Anti-anti malware

*From Chernobyl CIH 1.4*

```
Loop:
  pop  ecx
  jecxz SFModMark
  mov  esi, ecx
  mov  eax, 0d601h
  pop  edx
  pop  ecx
  call edi
  jmp  Loop
```

```
Loop:
  pop  ecx
  nop
  jecxz SFModMark
  xor  ebx, ebx
  beqz N1
N1:
  mov  esi, ecx
  nop
  mov  eax, 0d601h
  pop  edx
  pop  ecx
  xor  ebx, ebx
  beqz N2
N2:
  jmp  Loop
```
Anti-anti malware

From Chernobyl CIH 1.4

Loop:

pop  ecx
jecxz SFModMark
mov  esi, ecx
mov  eax, 0d601h
pop  edx
pop  ecx
pop  edx
call edi
jmp  Loop

N2:

nop
xor  ebx, ebx
beqz N2
mov  eax, 0d601h
pop  edx
pop  ecx
nop
jecxz SFModMark
xor  ebx, ebx
beqz N1

N1:

mov  esi, ecx
Anti anti malware

Loop:

pop ecx
jecxz SFModMark
mov esi, ecx
mov eax, 0d601h
pop edx
pop ecx
jmp Loop

From Chernobyl CIH 1.4

Loop:

pop ecx
jecxz SFModMark
mov esi, ecx
mov eax, 0d601h
pop edx
pop ecx
nop
jmp L1
L3: call edi
xor ebx, ebx
beqz N2
N2: jmp Loop

L1: jecxz SFModMark
xor ebx, ebx
beqz N1
N1: mov esi, ecx
jmp L2

L4:
Anti-anti malware

Malware evolution

* How can we model and compute signatures for metamorphism?
Solutions

- **Idea:** Behavior Monitors

- Run suspect program in an emulator and extract a DB of relevant signatures *(huge DB)*

- Look for changes in file structure: Some viruses modify files in a consistent way *(inaccurate)*

- Disassemble and look for virus-like instructions: reverse engineering malware *(expensive)*
Solutions

- The code may appear in many variants
- The signature has to describe the semantics
- The signature is a language \( L \) of possible signatures extracted from the various versions of the malware
  \[
  \sigma \in L \implies \sigma \text{ is a possible signature}
  \]
- Malware may include its own metamorphic engine (Jean-Yves)!
How to build signatures?
By Abstract Syntax Trees

Look for clones in this program:

\[(5 + (a + b)) \times (7 + (c + 9))\]

Parse and build an AST S:

Generate all tree patterns
Look for clones in this program:

\[(5 + (a + b)) \times (7 + (c + 9))\]

Parse and build an AST:

```
int + int
var var
5 +
int
7 +
var var
int
9 +
```

Look for clones in this program:

\[(5 + (a + b)) \times (7 + (c + 9))\]

Parse and build an AST:

```
+ int + var + int + var + int
5 a b c 7 9
```

Which patterns would make good clones?

This pattern seems like it might make a good choice:

```
+ int + var + var + int + var + int + int
5 a b c 7 9
```

Slow, syntactic and it does not scale!
Graph-based similarity

PDG: Program dependency graph

\[
\begin{align*}
P_1 & \quad [1. x := 0; \\
& \quad 2. i := 1; \quad 3. \text{while } i > 0 \text{ do } i := i + 1; \\
& \quad 4. y := x; \\
\end{align*}
\]

\[
\begin{align*}
P_2 & \quad [1. x := 0; \\
& \quad 4. y := x; \\
\end{align*}
\]
Graph-based similarity

\[ S_0: \text{int } k = 0; \]
\[ S_1: \text{int } s = 1; \]
\[ S_2: \text{while } (k < w) \{ \]
\[ S_3: \quad \text{if } (x[k] == 1) \]
\[ S_4: \quad R = (s*y) \% n; \]
\[ \quad \text{else} \]
\[ S_5: \quad R = s; \]
\[ S_6: \quad s = R*R \% n; \]
\[ S_7: \quad L = R; \]
\[ S_8: \quad k = k+1; \]
\}
Graph-based similarity

Definition
Common subgraphs Let $G$, $G_1$, and $G_2$ be graphs. $G$ is a *common subgraph* of $G_1$ and $G_2$ if there exists subgraph isomorphisms from $G$ to $G_1$ and from $G$ to $G_2$.

$G$ is the *maximal common subgraph* of two graphs $G_1$ and $G_2$ ($G = mcs(G_1, G_2)$) if $G$ is a common subgraph of $G_1$ and $G_2$ and there exists no other common subgraph $G'$ of $G_1$ and $G_2$ that has more nodes than $G$.

Still syntactic and it does not scale!
Graph-based similarity

Definition

Graph similarity and containment Let $|G|$ be the number of nodes in $G$. The $\text{similarity}(G_1, G_2)$ of $G_1$ and $G_2$ is defined as

$$ \text{similarity}(G_1, G_2) = \frac{|\text{mcs}(G_1, G_2)|}{\max(|G_1|, |G_2|)} $$

The $\text{containment}(G_1, G_2)$ of $G_1$ within $G_2$ is defined as

$$ \text{containment}(G_1, G_2) = \frac{|\text{mcs}(G_1, G_2)|}{|G_1|} $$

We say that $G_1$ is $\gamma$-isomorphic to $G_2$ if

$$ \text{containment}(G_1, G_2) \geq \gamma, \gamma \in (0, 1]. $$

$G_1$ and $G_2$ are graphs, and $\text{mcs}(G_1, G_2)$ is the maximum common subgraph of $G_1$ and $G_2$.

Example

$\text{similarity}(G_1, G_2) = \frac{4}{7}$ and $\text{containment}(G_1, G_2) = \frac{4}{6}$. 
Plagiarism detection

Make pair-wise comparisons between all the programs handed in by the students:

\[ \langle P1, P2 \rangle = 70\% \]
\[ \langle P1, P3 \rangle = 20\% \]
\[ \langle P2, P3 \rangle = 30\% \]

Malware!

Similarity analysis
Plagiarism detection

Make pair-wise comparisons between all the programs handed in by the students:

\[
\langle P_1, P_2 \rangle = 70\% \\
\langle P_1, P_3 \rangle = 20\% \\
\langle P_2, P_3 \rangle = 10\%
\]

Malware!

Similarity analysis

False negative
[Dalla Preda et al., POPL 2007]

In order to abstract away from syntactic variants of similar malware, we need to extract semantic models!

Develop a semantics-based similarity analysis!
Undecidable!

Too complicated, complex, undecidable
Abstraction

False positives!
Completeness

Abstraction $\rightarrow$ (sound) loss of precision

Incompleteness
Completeness

Abstraction $\Rightarrow$ loss of precision

Incompleteness
Completeness

Abstraction ➞ loss of precision
Incompleteness

refining

Error

Making ABSTRACTIONS COMPLETE: Refining input domains

\[ \text{Giacobazzi et al. JACM'00} \]
...the ingredients!
CFG
A language of instructions
CFG

Idea: abstract basic blocks
Why Abstract Interpretation?

Abstract Interpretation (1977) is the a general model for the (static or dynamic) approximation of semantics of discrete dynamic systems.

Including: Static program analysis, dynamic analysis, profiling, debugging, tracing, compilation, de-compilation, type checking and type inference, model checking and predicate abstraction, trajectory evaluation, testing, proof systems, etc.
**Abstract Interpretation**

Design approximate semantics of programs [Cousot & Cousot ’77, ’79].

Galois Connection: \( \langle C, \alpha, \gamma, A \rangle \), \( A \) and \( C \) are complete lattices.

Closures: \( \langle uco(C), \sqsubseteq \rangle \) set of all possible abstract domains,

\( A_1 \sqsubseteq A_2 \) if \( A_1 \) is more concrete than \( A_2 \)
Symbolic Automata

Effective boolean algebra
\[ \langle \mathcal{D}_A, \Psi_A, [\cdot], \bot, \top, \land, \lor, \neg \rangle \]

Semantics
\[ [\cdot] : \Psi_A \rightarrow \mathcal{P}(\mathcal{D}_A) \]

SFA
\[ \langle \mathcal{A}, Q, q_0, F, \Delta \rangle \]

Moves
\[ p \xrightarrow{a}^M q \quad a \in [\Phi] \]

\[ \mathcal{L}(M) = \{ a_1, \ldots, a_n \in \mathcal{D}_A \mid \forall 1 \leq i \leq n. \ p_{i-1} \xrightarrow{a_i}^M p_i \} \]
combining ingredients!
Abstract Symbolic Automata

\[ \langle \mathcal{D}_A, \Psi_A, [\cdot]^\rho, \rho(\bot), \top, \land, \lor^\rho, \lnot^\rho \rangle \]

Abstraction

\[ [\cdot]^\rho : \Psi_A \rightarrow \rho(\mathcal{D}_A) \text{ such that } [\varphi]^\rho = \rho([\varphi]) \]

Abstract Interpretation

Building symbolic formulae
Abstract Symbolic Automata

$I \in \mathbb{I} ::= \text{mov } e_1, e_2 \mid \text{pop } e \mid \text{push } e \mid \text{OP } e_1, e_2 \mid \text{undef } e$

Effective Boolean Algebra: \( A_P = \langle \Sigma \times \Sigma, \mathbb{I}^*, [\cdot], \bot, \top, \lor, \land, \neg \rangle \)

SFA of P: \( \langle A_P, Q_P, \text{start}, F, \Delta_P \rangle \)

Formulae are sequence of instructions in Basic Blocks: I/O predicates

\( \varphi \in \mathbb{I}^* : [\varphi] = \{ \ i, o \in \Sigma \land \langle i, o \rangle \mid o = \text{exec}(\varphi, i) \ \} \)

\[ [P](i) = \left\{ \langle i_0, o_1 \rangle, \ldots, \langle i_{n-1}, o_n \rangle \mid \begin{array}{c} B_0, \ldots, B_n \in \text{Trace}(G_P) \\ i_0 = i \land B_0 = \text{start} \\ \forall 1 \leq i \leq n. \ o_i = \text{exec}(B_i, i_{n-1}) \end{array} \right\} \]

\[ [P](i) \in \mathcal{L}(M(P)) \]
Abstract Symbolic Automata

Abstraction
Symbolic Abstraction

(a) Code

(b) Semantics

(c) Juice

(a) Semantics 1

(b) Semantics 2

(c) Juice

(d) Abstracted Juice
**BinJuice as Abstractions**

<table>
<thead>
<tr>
<th>(a) Variant 1</th>
<th>(b) Variant 2</th>
<th>(c) Semantics</th>
<th>(d) Juice</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(ebp)</td>
<td>push(ebp)</td>
<td>eax = -20 + def(esp)</td>
<td>A = -N1 + def(B)</td>
</tr>
<tr>
<td>mov(ebp, esp)</td>
<td>mov(ebp, esp)</td>
<td>ebp = -4 + def(esp)</td>
<td>C = -N2 + def(B)</td>
</tr>
<tr>
<td>sub(esp, 16)</td>
<td>sub(esp, 16)</td>
<td>esp = -24 + def(esp)</td>
<td>B = -N3 + def(B)</td>
</tr>
<tr>
<td>mov(eax, dptr(ebp))</td>
<td>mov(eax, dptr(ebp))</td>
<td>memdw(-24 + def(esp)) = -20 + def(esp)</td>
<td>memdw(-N3 + def(B)) = -N1 + def(B)</td>
</tr>
<tr>
<td>push(edi)</td>
<td>mov(ebp, esp)</td>
<td>memdw(-20 + def(esp)) = 1634038339</td>
<td>memdw(-N1 + def(B)) = N4</td>
</tr>
<tr>
<td>push(ebx)</td>
<td>sub(esp, 16)</td>
<td>memdw(-16 + def(esp)) = 1766221172</td>
<td>memdw(-N5 + def(B)) = N6</td>
</tr>
<tr>
<td>mov(ebx, ebp)</td>
<td>mov(ebix, ebp)</td>
<td>memdw(-12 + def(esp)) = 4285804</td>
<td>memdw(-N7 + def(B)) = N8</td>
</tr>
<tr>
<td>mov(edi, ebx)</td>
<td>mov(ebx, ebx)</td>
<td>memdw(-8 + def(esp)) = def(ebp)</td>
<td>memdw(-N9 + def(B)) = def(C)</td>
</tr>
<tr>
<td>pop(ebx)</td>
<td>mov(ebx, ebx)</td>
<td>memdw(-4 + def(esp)) = def(ebp)</td>
<td>memdw(-N2 + def(B)) = def(C)</td>
</tr>
<tr>
<td></td>
<td>pop(ebx)</td>
<td></td>
<td>where</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N6 = N10 + N11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-N12 = -N2 - N13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-N1 = -N2 - N5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-N5 = -N1 + N2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-N7 = -N1 + N9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-N9 = -N12 + N14</td>
</tr>
</tbody>
</table>

Symbolic semantics

Algebraic simplification
The example is indicative of what we find for all the variants of structurally equal.

Some goat files were reinfected up to six times, creating up to six 'generations' of infection.

The proposed method of extracting a binary's juice consists of the following steps:

1. Compute the semantics of a block.
2. Compute the juice of a block.
3. Compute the juice of a procedure.
4. Compute the juice of a function.
5. Limitations

Any method for finding semantically similar code fragments is fundamentally limited by Rice's Theorem. So it is a given that no compiler can precisely identify equivalent (or similar) code segments in columns (a) and (b) are for corresponding blocks of code for variants even though the code for the blocks may be cally equivalent programs that differ in how computation is spread.

The distributive property is used to refactor an expression so that, while preserving the semantics of the overall program, do not significantly improve upon the lineage computed using n-perms. Though these results are not exciting, they are also not conclusive. Since we used the same version of the compiler with the method presented.

The most significant limitation of the proposed method is that it did not significantly improve upon the lineage computed using n-perms. Yet, it is instructive to show the tautologies used in algebraic simplification for Variant 1.

Our experience with Win32.Evol showed that the semantics and structure to the CFG and whose nodes represent the semantics (or interpretation. The operation encoded by an assembly instruction, such as push(ebx), pop(esi), sub(edi,ebx), push(ebx), push(edi), mem(def(ebx)) = def(eax), mov(ebp,esp), sub(esp,16), ebx = 10, etc. Similarly, 'A = -N1 + def(B)
C = -N2 + def(B)
B = -N3 + def(B)
memdw(-N3 + def(B)) = -N1 + def(B)
memdw(-N1 + def(B)) = N4
memdw(-N5 + def(B)) = N6
memdw(-N7 + def(B)) = N8
memdw(-N9 + def(B)) = def(C)
memdw(-N2 + def(B)) = def(C)
memdw(-8 + def(esp)) = def(ebp)
memdw(-4 + def(esp)) = def(ebp)

R1 = N1
R2 = N2
mem(def(R1)) = def(R2)
mem(def(R2)) = def(R1)

R = N
mem(def(R)) = def(R)

The crux of the algorithm lies in the computation of semantics (juice) of the corresponding blocks always yields the correct semantics (juice), though the two semantics (juice) are not necessarily an approximation. The disassembly produced by the metamorphic engine of Win32.Evol. The features were then used in the MAAGI system to construct BinJuice Automata.
Challenges

• **Applications**

  • **Minimization**: Minimal BinJuice Automata as *signature* for a variety of similar malware

  • **BinJuice Simulation/bisimulation**: when one malware simulates another or two are bisimilar (*whole code similarity analysis*)

  • **Widening** of BinJuice Automata to merge signatures

• **Theory**

  • How symbolic automata transformations behaves when abstraction to formulae is applied?

  **Completeness in ASFA transformations**

  ![Palantir](image1.png) ➔ **Big Data of Code**
Challenges

Widening on SFA

- Let $M_1$ and $M_2$ be two SFA (BinJuice Automata)

- $R_n \subseteq Q_1 \times Q_2$ is a state relation

  $(q_1, q_2) \in R_n$ if $q_1$ and $q_2$ recognize the same language of strings of length $n$

  $q \equiv_R q'$ iff $\exists r \in Q_1 : (r, q) \in R_n$ and $(r, q') \in R_n$

  $M_1 \triangledown M_2 = M_2/\equiv_R$

- It is a widening if on finite / ACC formulae set: approximate instruction terms!
Challenges

Widening on SFA

MEM[f]:=100;input=>MEM[a];MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...
Challenges
Completeness on SFA
Thanks!