
Forensic Memory Analysis: From Stack and Code to Execution History.

Computer Security Laboratory
Concordia Institute for Information
System Engineering (CIISE)

Summary

- State of the art.
 - Our approach.
 - Program execution modeling.
 - Control Flow Graphs.
 - Local automata model.
 - Push Down System model.
 - ADM logic.
 - Stack modeling using ADM.
 - Implementation.
 - Future research directions.
-

State of the Art

- The Memory Analysis Challenge (DFRWS 2005 Challenge) resulted in 2 new tools:
 - Memparser by Chris Betz
 - Enumerates processes (PsActiveProcessList)
 - Dumps process memory to disk
 - Dumps process strings to disk
 - Displays Process Environment Information
 - Displays all DLLs loaded by process
 - Kntlist by George M. Garner Jr. and Robert Jan Mora
 - Copies, compresses, creates checksums & sends a physical memory to a remote location.
 - Enumerates processes (PsActiveProcessList).
 - Enumerates handle table.
 - Enumerates driver objects (PsLoadedModuleList).
 - Enumerates network information such as interface list, arp list, address object and TCB table.
 - References are examined to find hidden data.
 - Object table, its members and objects inside object directory point to processes and threads.
 - Enumerates contents of IDT, GDT and SST to identify loaded modules.
-

Related work

- M. Burdach presents an approach to retrieve process and file information from the memory of Unix operating system by following the unbroken links between data structures in the memory.
 - FATKit (an extensible frame work) which provides the analyst with the ability of automatically deriving digital object definition from c source codes and extracting the objects from the memory.
 - A. Schuster proposed an approach to define signatures for executive object structures in the memory and recover the hidden and lost structures by scanning the memory looking for the predefined signatures.
 - Defining a signature that uniquely identify most of the data structures are not achievable except for a small set of kernel structures.
 - There are chances that this small set of kernel structures are overwritten by the kernel after a process has finished its execution while there is still some useful information about the process in other structures for which defining a signature which uniquely identifies the structure is impractical.
-

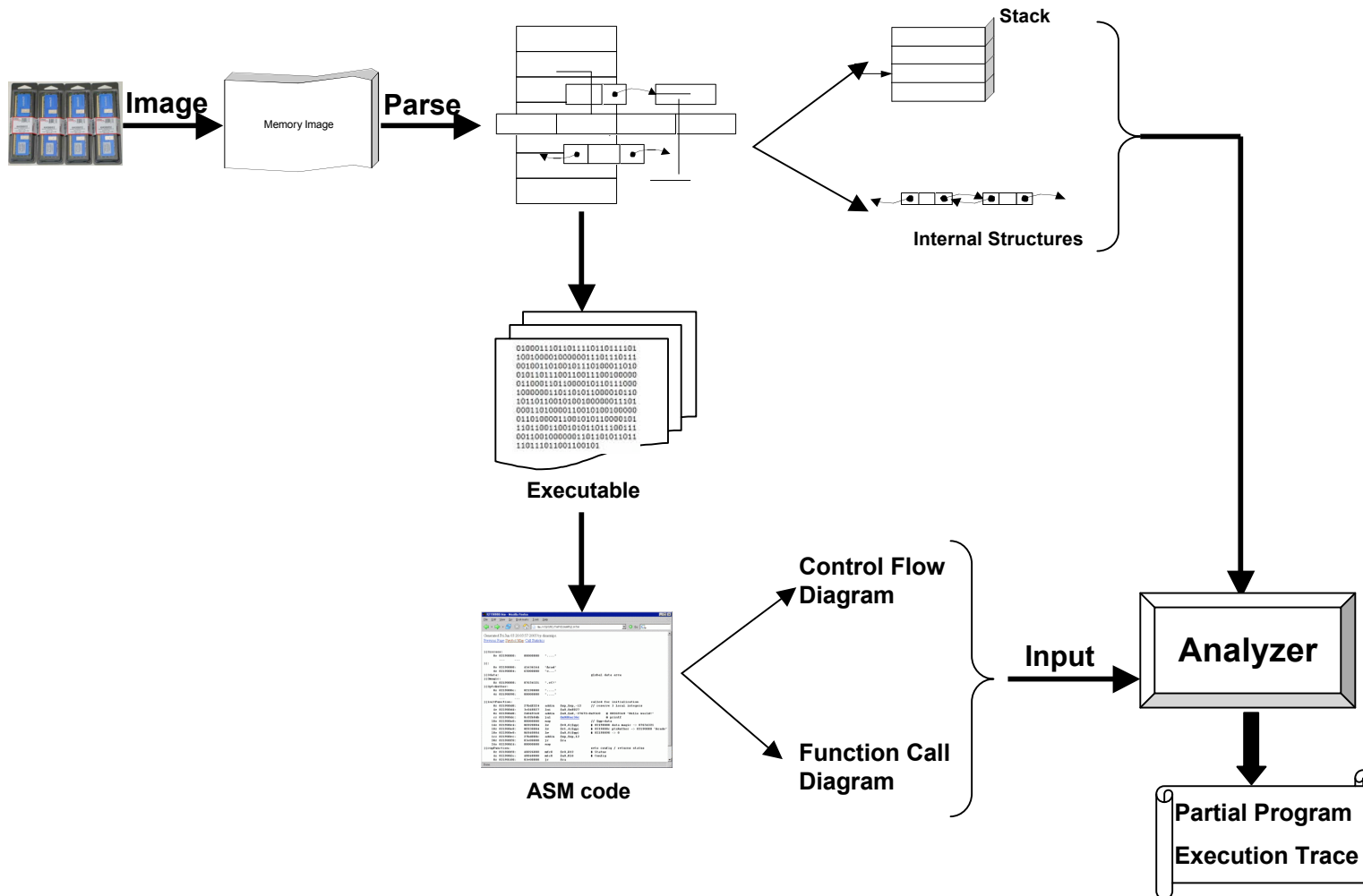
Our Approach

- Most of the previous work on forensics memory analysis has been on extraction and presentation of forensically relevant structures.
 - Our objective is to create a timeline of what has exactly been done during the incident in the form of an execution trace.
-

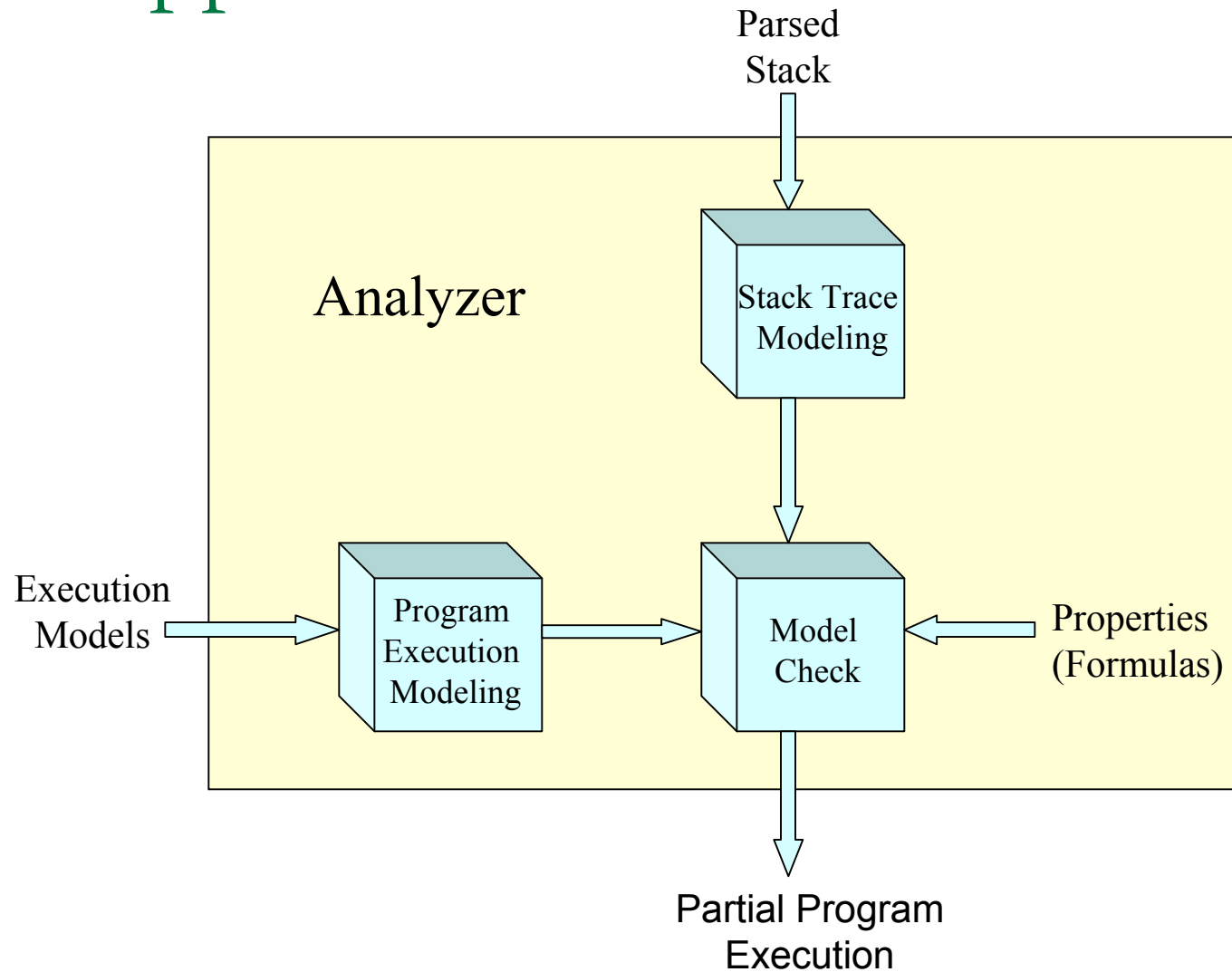
Our Approach

- For each function call made by a process, a stack frame is created and stored on the stack.
 - The stack frame contains the parameters passed to that function, the address of the caller and the local variables.
 - These function call traces entail the history of what a process has done.
 - After a function returns, the stack pointer will be moved down to the previous frame.
 - However, the stack frame still resides in the memory at the location where the stack pointer is pointing after the function returns. (overwritten)
 - We also have access to the binary code. We can extract a partial execution path of the process at the time of the incident that includes the data which has been processed.
 - Data flow analysis and control flow analysis techniques are utilized to recover the partial execution history of a program.
-

Our Approach



Our Approach



Advantage

- The analysis is performed on the extracted assembly code of the process from the memory and there is no need for the external provision of the source code or executable.
 - The technique integrates the formal analytical power of process logic and program models to retrieve the execution history of the process.
 - The result of the analysis could reveal important fact about what a process has done rather than what is currently existing in the memory.
-

Modeling program execution

- The program execution is modeled in three step.
 - First, we create the Control Flow Graph(CFG) of each function in the process.
 - Second, we transform these CFGs into local automata models.
 - Third, the local automata models of each function are combined to form a Push Down System (PDS).
-

Modeling program execution

- A control flow graph (CFG) is a structure that characterizes possible execution paths in a program.
 - Vertices of the graph contain one or more instructions of the program that execute sequentially.
 - Edges in the graph show how control flow transfers between blocks.
 - The first step of our approach is the generation of a control flow graph of each function called in the program.
-

Example

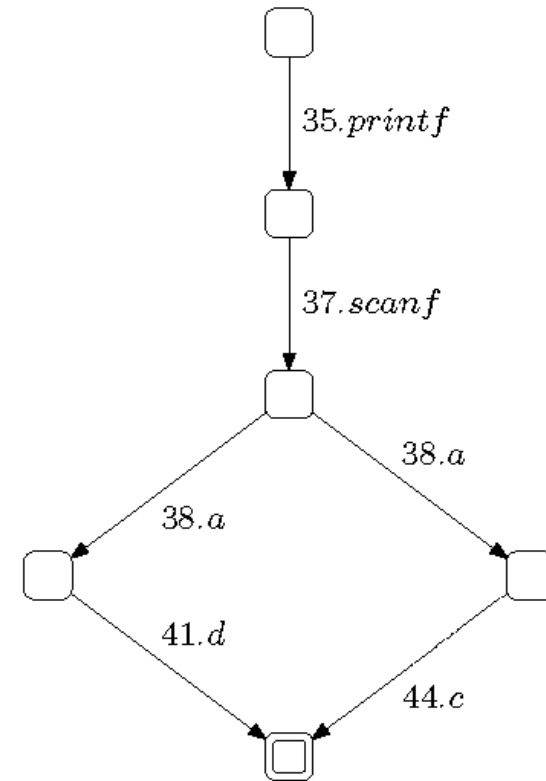
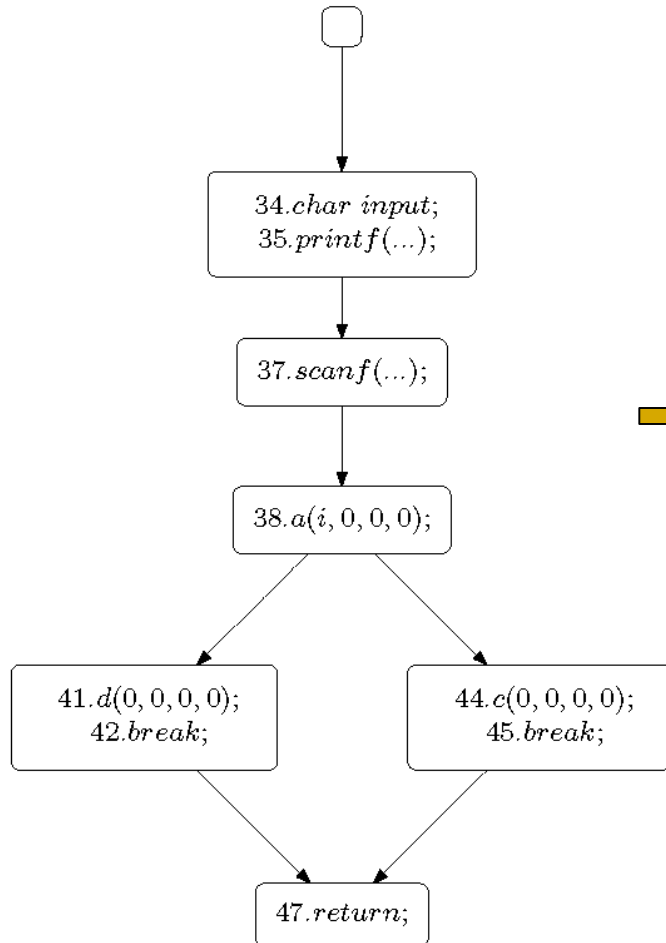
```
1. #include <iostream>
2. void op(int i);
3. void h(int i, int j) {
4.     return;
5. }
6. void g(int i, int j) {
7.     return;
8. }
9. void b(int i, int j, int k, int l) {
10.    return;
11. }
12. void e(int i, int j, int k, int l) {
13.    op(2);
14. }
15. void a(int i, int j, int k, int l) {
16.    if (i == 49) {
17.        g(i,j);
18.        e(i,j,k,l);
19.        return;
20.    }else{
21.        h(i,j);
22.        return;
23.    }
24. }
25. void c(int i, int j, int k, int l) {
26.    b(i,j,k,l);
27.    return;
28. }
29. void d(int i, int j, int k, int l) {
30.    h(i,j,k,l);
```

```
31.    return;
32. }
33. void op(int i) {
34.    char input;
35.    printf("Input a value
           between 1, 2:\n");
36.
37.    scanf("%c", &input);
38.    a(i,0,0,0);
39.    switch (input) {
40.        case '1':
41.            d(0,0,0,0);
42.            break;
43.        case '2':
44.            c(0,0,0,0);
45.            break;
46.    }
47.    return;
48. }
49. void inc(int i) {
50.    if (i < 10) {
51.        inc(i+1);
52.    } else {
53.        op(1);
54.        return;
55.    }
56. }
57. void main() {
58.    inc(0);
59. }
```

```
33. void op(int i) {
34.     char input;
35.     printf("Input a value
           between 1, 2:\n");
36.
37.     scanf("%c", &input);
38.     a(i,0,0,0);
39.     switch (input) {
40.         case '1':
41.             d(0,0,0,0);
42.             break;
43.         case '2':
44.             c(0,0,0,0);
45.             break;
46.     }
47.     return;
48. }
```

CFG of function *op*

Local Automaton

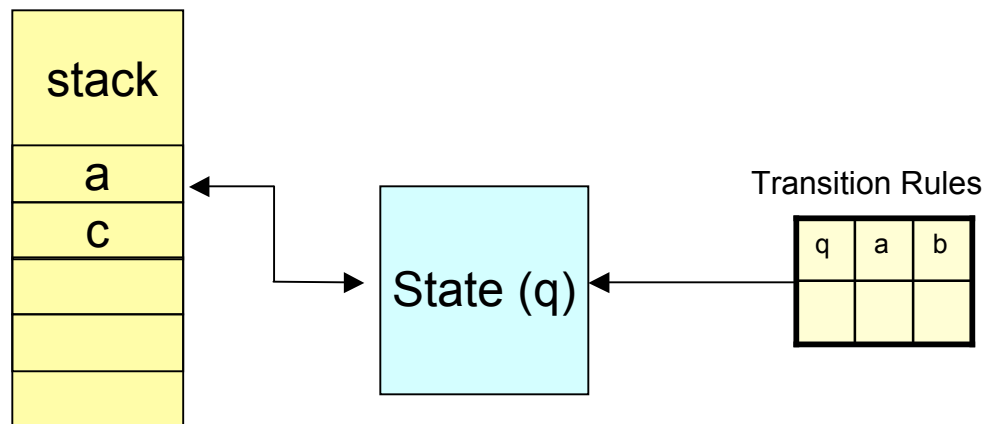


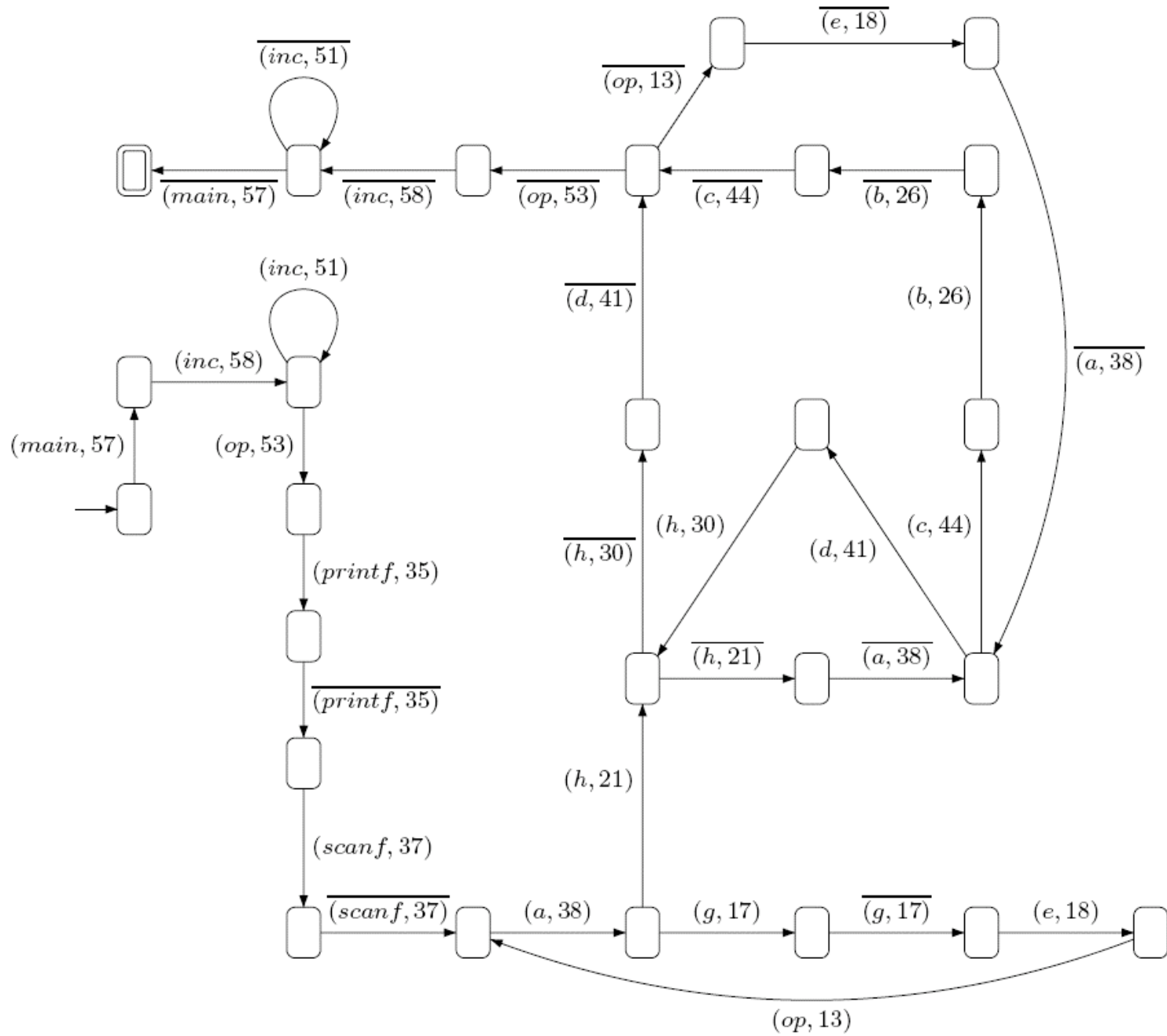
Combining the models

- Until now, we have modeled the execution of the program as a set of local state machines.
 - Next step is to combine these local models into a model that represents the whole program.
 - The local models can be combined to form a Push Down System (PDS).
 - As the result of this combination, we can accurately model the execution of the program in terms of function call and returns.
-

Push Down Systems

- A PDS is a triple $P = (Q, \Gamma, \sigma)$ where Q is the final set of control locations, Γ is the finite set of stack alphabets and $\sigma \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of transition rules.





Modeling the stack

- The process execution is modeled.
 - A set of rules could be derived from the function call implementation using stacking mechanism.
 - If stack frame *b* is on top of stack frame *c*, then either *c* has called *b* or *b* has been called before *c*.
 - The function call history should generate exactly the same stack trace and should not overwrite any of the currently existing stack frames.
-

ADM Logic

- ADM is a temporal, dynamic, linear and modal logic.
 - It has been designed initially to capture the specification of security properties in the context of cryptographic protocols.
 - Such a logic is also proved to be very relevant and useful in the context of cyber forensics analysis.
-

ADM Syntax

- The syntax of the logic is based on patterns that are sequences of actions and pattern variables.
- A pattern is defined by the following grammar:

$$p := a.p \mid x.p \mid \epsilon$$

Where:

ϵ - stands for the empty pattern

a - is an action

p - is a pattern variable

The set of action variables is denoted by \mathcal{V}_a

The set of pattern variables is defined by \mathcal{V}_p

ADM Logic Syntax

- Let X be a formula variable, then the set of logic formulas is obtained by the grammar given below:

$$\Phi ::= X \mid \neg\Phi \mid [p_1 \varrho p_2]\Phi \mid \Phi_1 \wedge \Phi_2 \mid \nu X.\Phi$$

Where:

\neg and \wedge represent negation and conjunction respectively.

$p_1 \varrho p_2$ is a model operator indexed by the two patterns p_1 and p_2 .

$\nu X.\Phi$ is a recursive formula, the greatest fixed point operator ν binds all free occurrences of X in Φ .

ADM Logic Syntax (cont.)

- For further convenience we use the following standard abbreviations:

$$\begin{aligned} tt &\equiv \nu X.X \\ ff &\equiv \mu X.X \\ \langle p_1 \leftrightarrow p_2 \rangle \Phi &\equiv \neg[p_1 \leftrightarrow p_2] \neg \Phi \\ \mu X.\Phi &\equiv \neg \nu X. \neg \Phi[\neg X/X] \\ \Phi_1 \vee \Phi_2 &\equiv \neg(\neg \Phi_1 \wedge \neg \Phi_2) \\ \Phi_1 \rightarrow \Phi_2 &\equiv \neg \Phi_1 \vee \Phi_2 \\ \Phi_1 \leftrightarrow \Phi_2 &\equiv \Phi_1 \rightarrow \Phi_2 \wedge \Phi_2 \rightarrow \Phi_1 \end{aligned}$$

Where $\Phi[\Gamma/X]$ represents the simultaneous replacement of all free occurrences of X in Φ by Γ .

ADM Denotational Semantics

- Suppose that Sub denotes the set of all possible substitutions σ such that:

$$\sigma \in [\mathcal{V}_p \rightarrow \mathcal{T}] \circ [\mathcal{V}_a \rightarrow \mathcal{A}]$$

Where:

\mathcal{V}_p is the set of pattern variables.

\mathcal{V}_a is the set of action variables.

\mathcal{T} is the set of all valid traces.

Env is the set of all possible environments in $[\mathcal{V} \rightarrow 2^{\mathcal{T}}]$

Furthermore, we use $e[X \mapsto U]$ to denote the environment e' defined as follows:

$$\begin{aligned} e'(Y) &= e(Y) && \text{if } Y \neq X \\ e'(X) &= U \end{aligned}$$

Semantics

- The semantics of formulas is given by the function:

$$\llbracket _ \rrbracket_{-}^{\prime} : \mathcal{L} \times \mathcal{T} \times Sub \times Env \rightarrow 2^{\mathcal{T}}$$

Defined inductively on the structures of formulas as shown in the next slide, where t_{\downarrow} is the set of traces inductively defined as follows:

$$\begin{aligned} (i) \quad & t \in t_{\downarrow} \\ (ii) \quad & t_1.a.t_2 \in t_{\downarrow} \Rightarrow t_1.t_2 \in t_{\downarrow} \end{aligned}$$

- Informally t_{\downarrow} contains all subtraces that could be extracted from t by eliminating some actions from the beginning, from the middle and/or from the end of t .
- Given a trace t , the semantics of a formula will be all the traces in t_{\downarrow} respecting the conditions specified by this formula.

Semantics (cont.)

$$\llbracket X \rrbracket_e^{t,\sigma} = e(X)$$

$$\llbracket \neg \Phi \rrbracket_e^{t,\sigma} = t_\downarrow - \llbracket \Phi \rrbracket_e^{t,\sigma}$$

$$\llbracket \Phi_1 \wedge \Phi_2 \rrbracket_e^{t,\sigma} = \llbracket \Phi_1 \rrbracket_e^{t,\sigma} \cap \llbracket \Phi_2 \rrbracket_e^{t,\sigma}$$

$$\llbracket [p_1 \multimap p_2] \Phi \rrbracket_e^{t,\sigma} = \{u \in t_\downarrow \mid \forall \sigma' : p_1 \sigma \sigma' = u \Rightarrow p_2 \sigma \sigma' \in \llbracket \Phi \rrbracket_e^{p_2 \sigma \sigma', \sigma' \circ \sigma}\}$$

$$\llbracket \nu X. \Phi \rrbracket_e^{t,\sigma} = \nu f \text{ where } \begin{cases} f : 2^T & \longrightarrow & 2^T \\ U & \longmapsto & \llbracket \Phi \rrbracket_{e[X \mapsto U]}^{t,\sigma} \end{cases}$$

Environment

- Environments are used to give a semantics to the formula X and to deal with recursive formulae.
- Substitutions are internal parameters used to give a semantics to the formula $[p_1 \rightsquigarrow p_2]\Phi$.

- Given an environment e and a substitution σ , we say that a trace t satisfies if:

$$t \in [[\Phi]]_e^{t,\sigma}$$

- Intuitively, the trace t satisfies the formula $[p_1 \rightsquigarrow p_2]\Phi$ if for all substitutions σ such that $p_1\sigma = t$, the new trace $p_2\sigma$ (the modified version of the trace t) satisfies the remaining part of the formula Φ .
-

Example

- Suppose that we want to verify whether trace t satisfies the formula Φ such that:

$$\begin{cases} t = b.a.c.b.d.a \\ \Phi = \langle x_1.a.x_2.b.x_3 \ \heartsuit \ x_1.x_2.x_3 \rangle \langle x_4.b.x_5.d.x_6 \ \heartsuit \ x_4.x_5.x_6 \rangle t \end{cases}$$

More precisely we want to verify if $t \in \llbracket \Phi \rrbracket_{\emptyset}^{t, \emptyset}$.

Verification

- Step1: Verify if there exists at least one substitution such that the trace t is equal to $(x_1.a.x_2.b.x_3)\sigma_1$

- This part is satisfied, since the following substitution fills the required condition.

$$\sigma_1 = \{x_1 \mapsto b, x_2 \mapsto c, x_3 \mapsto d.a\}$$

- The second version of the trace will be:

$$t_1 = (x_1.x_2.x_3)\sigma = b.c.d.a$$

- Step2: Verify if there exist at least one substitution such that the resulting trace is equal to $(x_4.b.x_5.d.x_6)\sigma_2$

- Again This part is satisfied with the following substitution:

$$\{x_4 \mapsto \epsilon, x_5 \mapsto c, x_6 \mapsto a\}$$

Modeling the stack

- A stack frame contains the address from which the program execution should continue after the function is returned.
 - Based on this address, both the **callee** and the **caller** and the exact address of the **call site** in the code are identifiable.
 - Each stack frame in our trace represents a unique call site.
 - The PDS model also captures program flows based on the call site instead of the function name.
 - Therefore, each stack frame can be associated with a PDS transition.
 - Each **stack frame** in our trace is modeled as a **triple (a,b,c)** which represents function a being called by function b at call site c.
 - The **transitions of the PDS model** are annotated by the call site as **(a,c)** showing the call to function a at call site c.
-

Modeling the stack

- The left over on the stack can be thought of as a trace of function calls.
- Therefore, the properties of the stack can be modeled using ADM logic.

- b was called and returned before c:

$$\langle x_1.b.x_2.b'.x_3.c.x_4 \varphi \rightarrow \epsilon \rangle tt.$$

- b has called c:

$$\nu X \langle x_1.b.c.x_2 \varphi \rightarrow \epsilon \rangle tt \vee$$

$$\langle x_3.b.x_4.y_1.x_5.y_1'.x_6.c.x_7 \varphi \rightarrow x_3.b.x_4.x_5.x_6.c.x_7 \rangle X.$$

Modeling the stack

- Remember the two rules:
 - If stack frame b is on top of stack frame c , then either c has called b or b has been called and returned before c .
 - The function call history should generate exactly the same stack trace and should not overwrite any of the currently existing stack frames.
 - The first property is modeled before.
 - With ADM logic we can only express properties on one single trace.
 - To model the second property we need to combine the program execution trace with the stack properties.
-

Combined Execution Trace

- **Definition.** If S is the stack trace and E represent an execution path that is accepted by the PDS model of the program, the *combined execution trace* is defined as follows:
 $comb(S,E) = S|.E$
 - Essentially, the combined execution trace is the concatenation of both traces while separating the traces using the | symbol.
-

Modeling the stack properties

- **The function call history should generate exactly the same stack trace and should not overwrite any of the currently existing stack frames.**

- This property can be modeled as below:

$$\nu X \langle (z_1, -, c).x_1. | .x_2.(z_1, c).x_3 \varphi \rightarrow x_3 \rangle (\nu Y \langle z_2.x_4.allow \varphi \rightarrow x_4 \rangle Y \vee \langle z_3.x_5 \varphi \rightarrow x_5.allow \rangle Y.) \wedge [(z_1, -, c).x_1 \varphi \rightarrow x_6] X.$$

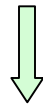
- The trace variables starting with x are subtraces and the variables starting with z are single events.
 - For each function call made after the return of the function representing the stack frame (z'_1) , the formula removes one *allow* from the end and for each function return it adds an *allow* at the end.
 - This way, the formula does not allow the paths that overwrite the stack frame being analyzed.
-

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \varphi \rightarrow x_3 \rangle (\nu Y \langle z_2.x_4.allow \varphi \rightarrow x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \varphi \rightarrow x_5.allow \rangle Y.) \wedge [(z_1, -, c).x_1 \varphi \rightarrow x_6] X.$$

$$(a, -, l_1). (b, a, l_2). (c, b, l_3). | . (a, l_1). (b, l_2). (d, l_3). (d', l_3). (c, l_4). (c', l_4). (b', l_2). (a', l_1)$$

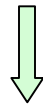
(c, l ₃)
(b, l ₂)
(a, l ₁)

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \varphi \rightarrow x_3 \rangle \quad \begin{array}{l} z_1 \rightarrow a \\ c \rightarrow l_1 \end{array}$$



x_3 is empty \rightarrow the first recursion is satisfied.

$$[(z_1, -, c).x_1 \varphi \rightarrow x_6]$$



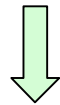
$$(b, a, l_2). (c, b, l_3). | . (a, l_1). (b, l_2). (d, l_3). (d', l_3). (c, l_4). (c', l_4). (b', l_2). (a', l_1)$$

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \wp \rightarrow x_3 \rangle (\nu Y \langle z_2.x_4.allow \wp \rightarrow x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \wp \rightarrow x_5.allow \rangle Y.) \wedge [(z_1, -, c).x_1 \wp \rightarrow x_6] X.$$

$(b, a, l_2). \underbrace{(c, b, l_3). | .(a, l_1).}_{\leftarrow} \underbrace{(b, l_2). (d, l_3). (d', l_3). (c, l_4). (c', l_4).}_{\leftarrow} \underbrace{(b', l_2). (a', l_1)}_{\leftarrow}$

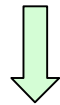
(c, l ₃)
(b, l ₂)
(a, l ₁)

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \wp \rightarrow x_3 \rangle \quad \begin{matrix} z_1 \rightarrow b \\ c \rightarrow l_2 \end{matrix}$$



x_3 is (a', l_1)

$$(\nu Y \langle z_2.x_4.allow \wp \rightarrow x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \wp \rightarrow x_5.allow \rangle Y.) \quad \begin{matrix} z_3 \rightarrow a \\ c \rightarrow l_1 \end{matrix}$$

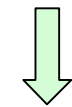


Satisfied

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \looparrowright x_3 \rangle (\nu Y \langle z_2.x_4.allow \looparrowright x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \looparrowright x_5.allow \rangle Y.) \wedge [(z_1, -, c).x_1 \looparrowright x_6] X.$$

$$(c, b, l_3). | .(a, l_1). (b, l_2). (d, l_3). (d', l_3). (c, l_4). (c', l_4). (b', l_2). (a', l_1)$$

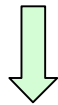
$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \looparrowright x_3 \rangle \quad \begin{array}{l} z_1 \rightarrow c \\ c \rightarrow l_3 \end{array}$$



x_3 is $(b', l_2). (a', l_1)$

$$(\nu Y \langle z_2.x_4.allow \looparrowright x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \looparrowright x_5.allow \rangle Y.) \quad \left[\begin{array}{l} z_3 \rightarrow b \\ c \rightarrow l_2 \end{array} \right.$$

First iteration



Satisfied

$$\left[\begin{array}{l} z_3 \rightarrow a \\ c \rightarrow l_1 \end{array} \right.$$

Second iteration

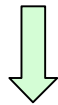
(c, l ₃)
(b, l ₂)
(a, l ₁)

$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \wp \rightarrow x_3 \rangle (\nu Y \langle z_2.x_4.allow \wp \rightarrow x_4 \rangle Y \vee \langle \bar{z}_3.x_5 \wp \rightarrow x_5.allow \rangle Y.) \wedge [(z_1, -, c).x_1 \wp \rightarrow x_6] X.$$

(c, l ₃)
(b, l ₂)
(a, l ₁)

$$(c, b, l_3). | .(a, l_1). (b, l_2). (d, l_3). (d', l_3). (c, l_4). (c', l_4). (e, l_5). (e', l_5) (b', l_2). (a', l_1)$$

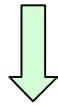
$$\nu X \langle (z_1, -, c).x_1. | .x_2. (\bar{z}_1, c).x_3 \wp \rightarrow x_3 \rangle \left[\begin{array}{l} z_1 \rightarrow c \\ c \rightarrow l_3 \end{array} \right]$$



$$x_3 \text{ is } (e, l_5). (e', l_5) (b', l_2). (a', l_1)$$

$$(\nu Y \langle \underline{z_2.x_4.allow \wp \rightarrow x_4} \rangle Y \vee \langle \bar{z}_3.x_5 \wp \rightarrow x_5.allow \rangle Y.) \left[\begin{array}{l} z_2 \rightarrow e \\ c \rightarrow l_5 \end{array} \right]$$

First iteration



X

Modeling the whole stack

- In order to model the whole stack we add all the properties specified above in one formula in five steps:
 - Find the call and return states corresponding to the first frame:

$$\langle (z_1, -, c). - .x_1. | .x_2. (z_1, c).x_3. (\bar{z}_1, c).x_4 \varphi \rightarrow x_3 \rangle \\ (\nu Z \langle (z_3, l).x_5. (z_3, l).x_6 \varphi \rightarrow x_5.x_6 \rangle Z \vee \text{empty.tt})$$

- Make sure that the stack frame is not overwritten by the following function calls:

$$\nu X \langle (z_1, -, c). - .x_1. | .x_7. (\bar{z}_1, c).x_3 \varphi \rightarrow x_3 \rangle (\nu Y \langle z_2.x_8.allow \varphi \rightarrow x_8 \rangle Y \\ \vee \langle \bar{z}_3.x_9 \varphi \rightarrow x_9.allow \rangle Y.)$$

Modeling the whole stack

- Specify interrelationships between two consecutive stack frames:

$$\begin{aligned} z_2 \rightarrow z_1 &\equiv \\ \nu U \langle (z_1, -, c). (z_2, z_1, d). x_1. |. x_2. (z_1, c). (z_2, d). x_{10} \\ &\quad \wp \rightarrow \epsilon \rangle tt \vee \\ &\langle (z_1, -, c). (z_2, z_1, d). x_1. |. x_2. (z_1, c). x_{11}. (z_3, e). \\ &\quad x_{12}. (z_3, e). x_{13}. (z_2, d). x_{10} \wp \rightarrow \\ &\quad (z_1, -, c). (z_2, z_1, d). x_1. |. x_2. (z_1, c). x_{11}. x_{12}. x_{13}. \\ &\quad (z_2, d). x_{10} \rangle U. \end{aligned}$$

$$\begin{aligned} z_2 \langle z_1 &\equiv \\ \langle (z_1, -, c). (z_2, -, d). x_1. |. x_{13}. (z_2, d). x_{14}. (z_2, d). x_{15} \\ &\quad . (z_1, c). x_3. (z_1, c). x_4 \wp \rightarrow \epsilon \rangle tt. \end{aligned}$$

Modeling the whole stack

- To preserve the place of the function call that correspond to the next frame and was found in the previous formula, we mark the trace at the specified location.

$$\begin{aligned} & \textit{if } z_1 \rightarrow z_2 : \\ & \langle (z_1, -, -). (z_2, z_1, -). x_1. |. x_{16}. (z_2, -). x_{10} \uparrow \rightarrow \\ & \quad (z_2, -, -). x_1. |. x_{16}. \uparrow . (z_2, -). \downarrow . x_9 \rangle \end{aligned}$$

$$\begin{aligned} & \textit{if } z_2 \langle z_1 : \\ & \langle (z_1, -, c). (z_2, -, d). x_1. |. x_{13}. (z_2, d). x_{17} \uparrow \rightarrow \\ & \quad (z_2, -, -). x_1. |. x_{13}. \uparrow . (z_2, -). \downarrow . x_{17} \rangle \end{aligned}$$

- These markers should be removed in the next iteration by the following formula.

$$\begin{aligned} & \langle (z_1, -, -). (z_2, -, -). x_1. |. x_2. \downarrow . (z_1, -). \uparrow . x_{18} \uparrow \rightarrow \\ & \quad (z_1, -, -). (z_2, -, -). x_1. |. x_2. (z_1, -). x_{18} \rangle \end{aligned}$$

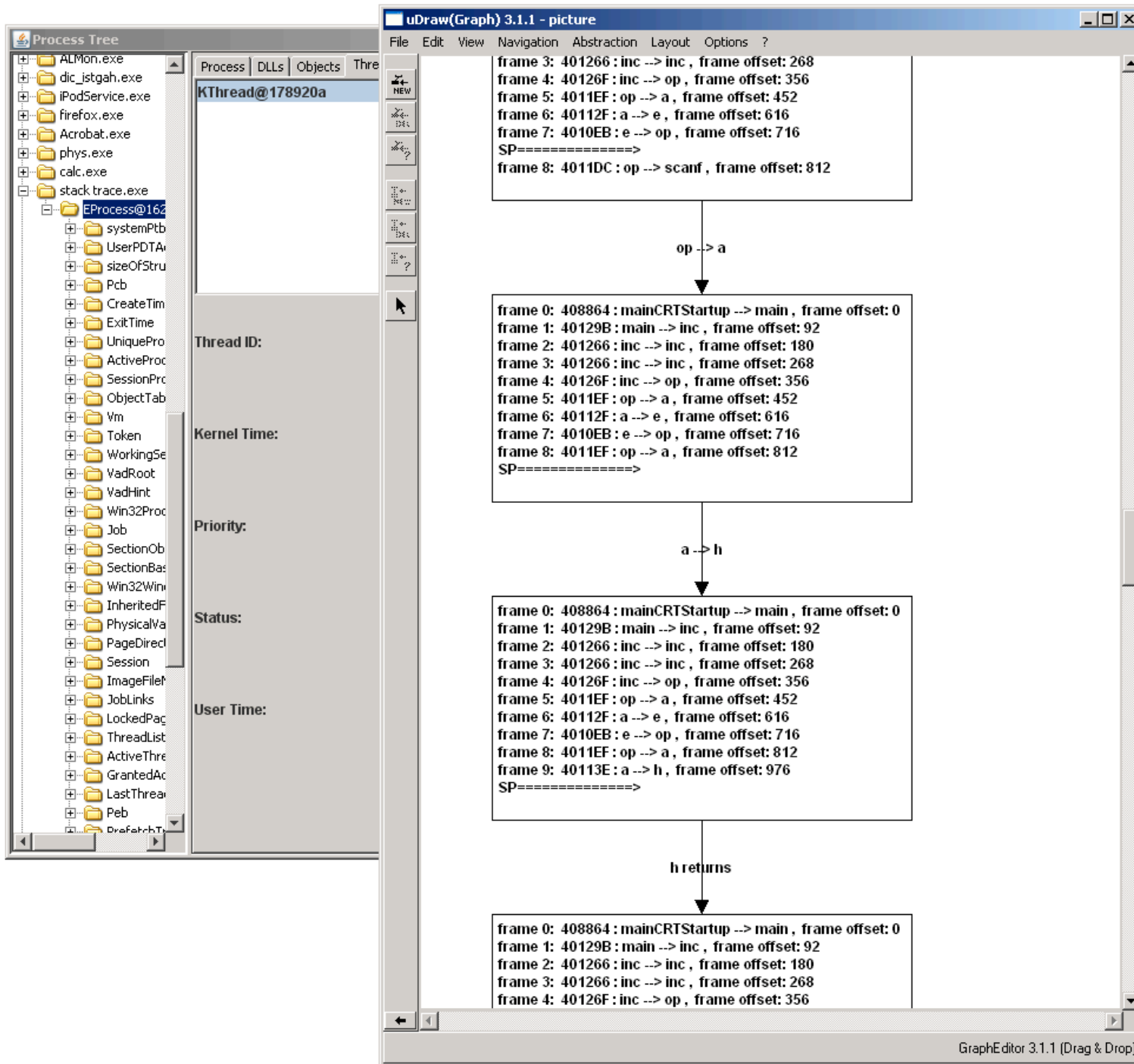
Modeling the whole stack

- To model the whole stack, we combine all the formulae using proper operators (\forall, \wedge).
- The complete query could be simplified by defining some macros as below:

$$\begin{aligned} & FindCallCite(a, d, e) \wedge \\ & \nu X. Unmark(a, d) (NotOverWrite(a, d, e) \wedge \\ & \quad ((Call(a, b, d, f) \wedge MarkIfCall(a, b, f) X.) \vee \\ & \quad (Before(a, b, g) \wedge MarkIfBefore(a, b, g) X.))) \end{aligned}$$

Some implementation details

- The verification of the logic is implemented based on the tableau based proof system of the logic.
 - Windows memory forensic analyzer is developed as part of our integrated forensic analysis framework.
 - The stack of the process is extracted from memory by parsing the structures of the process manager (EPROCESS, ...).
 - In order to parse the stack we use two techniques:
 - The OLD_EBP field on the stack holds the address of the previous frame OLD_EBP and therefore stack frames are chained together and the stack parser can follow this chain to correctly identify each stack frame.
 - Some compilers tend to use the EBP pointer within the function as a general purpose register and therefore in this case we can not trace back stack frames using this technique.
 - Look for return addresses that point to right after a call instruction. The stack will be traversed word by word testing which address is pointing to an instruction after a call instruction.
-



Future research direction

- Improve the model checking algorithm.
 - The same technique could be very useful in debugging of the crash dumps.
 - There is a wealth of information in memory dumps that can be used to more specifically detect the process execution history such as thread heap and kernel structures.
 - The stack trace and stack residue retrieved by our approach could be considered as a log from system activities and could be correlated with other sources such as network logs, operating system logs, etc.
-