Automated fault identification

Hashem WALY
Supervisor: Dr. Béchir KTARI

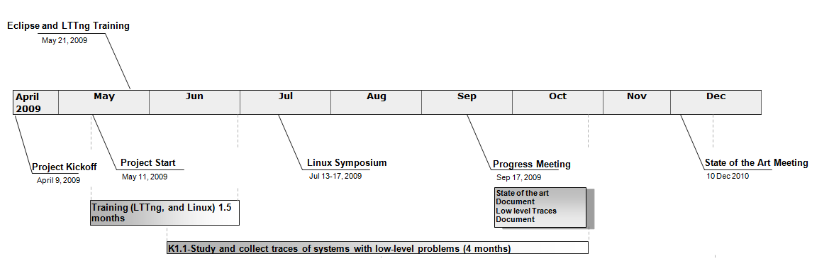
FACULTÉ DES SCIENCES ET DE GÉNIE
Université Laval, Quebec, Canada.

September 17, 2009
Montréal, Canada

Agenda

Plan

Updated Project Schedule



Milestones

K1.1

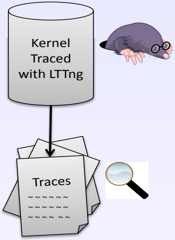
Build a list of low level problems and collect a database of good traces and of traces illustrating these problems (excessive swapping, saturated disk subsystem...).

K1.2

Study the various languages that may be suitable to describe different fault patterns. Compare their expressiveness, potential for performance, and applicability to detect a wide range of problems.

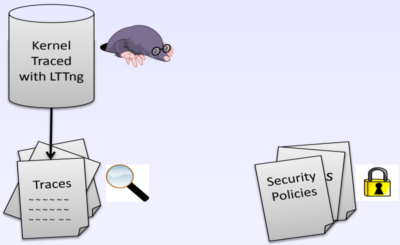
Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



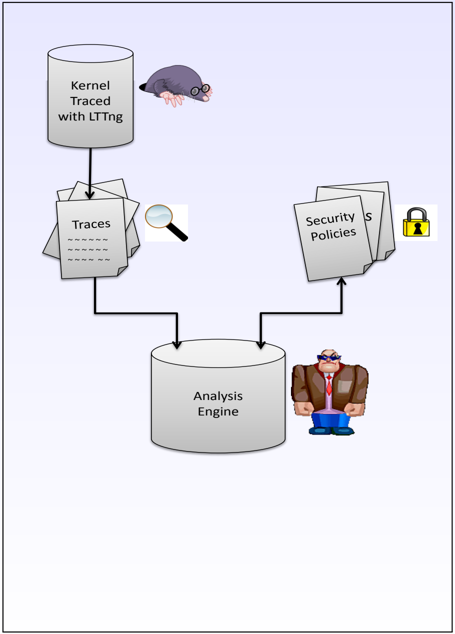
Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



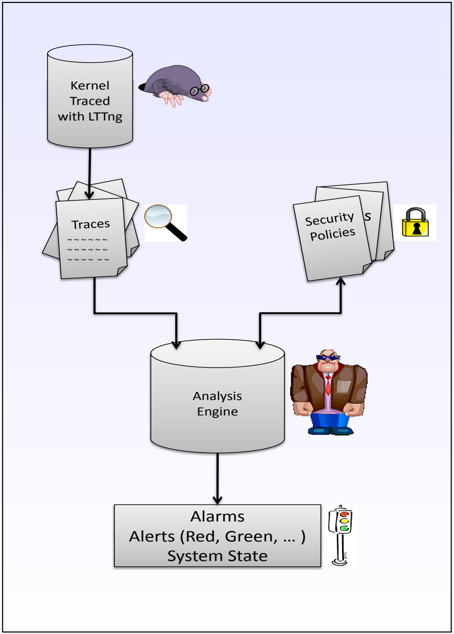
Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



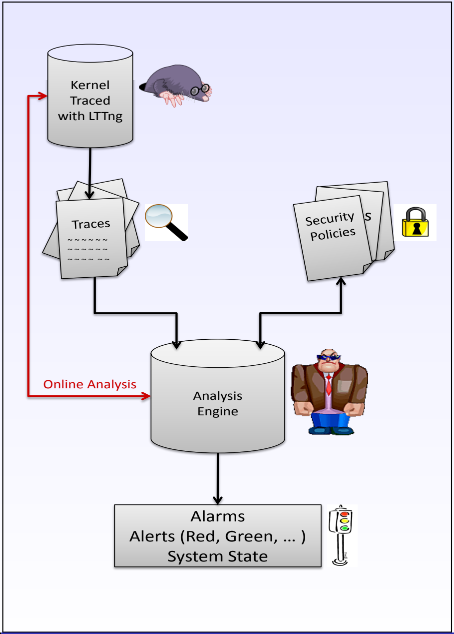
Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



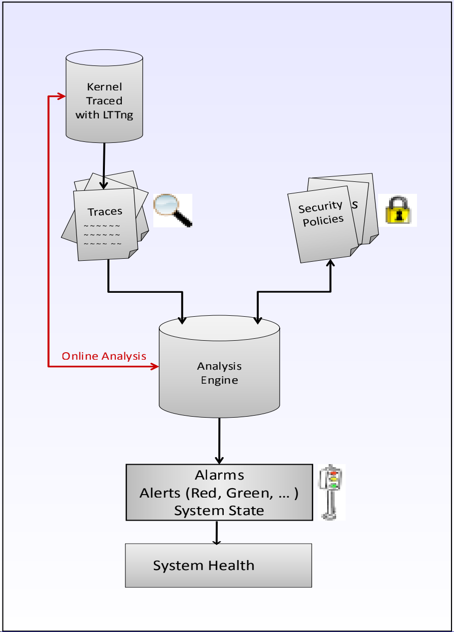
Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



Objectives

- **Automating** the detection of malicious behaviors, performance degradation, and software bugs.
- In the context of multi-core CPUs, and high level of interconnectivity between networked systems.



Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.

pgflastimage

Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.

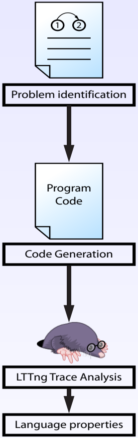
pgflastimage

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.

pgflastimage

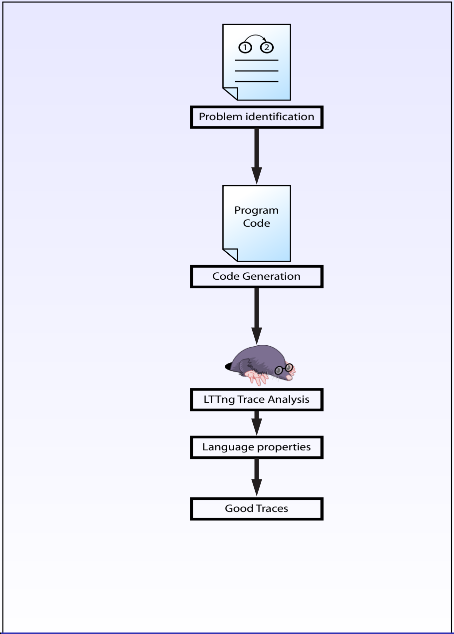
Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.



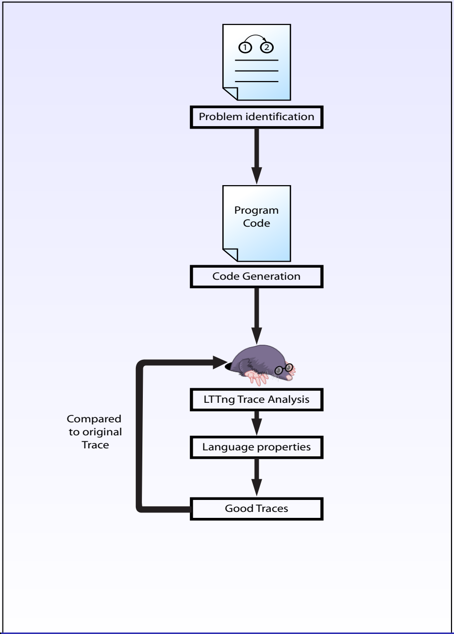
Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.



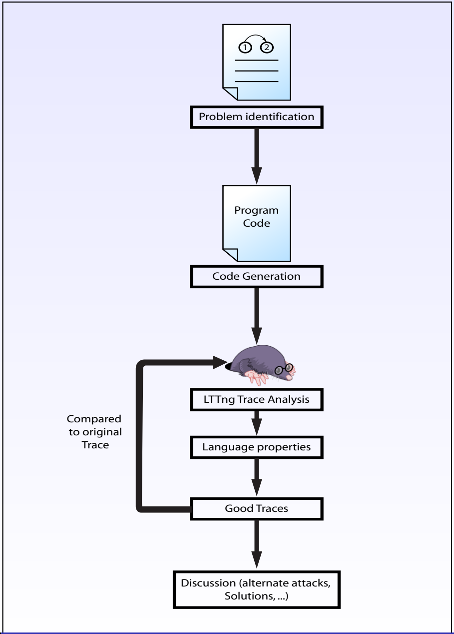
Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.



Scientific Model

- Problem Identification (the category of the attack, severity, FSM, ...).
- Code snapshot (Create or re-use code).
- LTTng Trace analysis (refine the trace and study relevant events).
- Language properties.
- Good Traces.
- Analysis.



Plan

Malicious Traces

- Security
  - File permissions and attributes.
    - Escaping a chroot Jail.
    - Race conditions on files.
  - Privilege Escalation.
    - Abusing `setuid` function.
  - Buffer Overflow.
  - Networks.
    - SYN Flood attack.
  - Viruses.
    - Linux RST.b virus.
- Testing Programs
  - Using File Descriptors
- System Performance
  - Inefficient I/O

Malicious Traces

- Security
  - File permissions and attributes.
    - Escaping a chroot Jail.
    - Race conditions on files.
  - Privilege Escalation.
    - Abusing `setuid` function.
  - Buffer Overflow.
  - Networks.
    - SYN Flood attack.
  - Viruses.
    - Linux RST.b virus.
- Testing Programs
  - Using File Descriptors
- System Performance
  - Inefficient I/O

Plan

File Permissions and attributes

Why securing file permissions is important?

- In Linux, everything is a file!
- First line of defense against attacks.

Linux file attributes: Users fall into:

1. Owner of the file.
2. Same group.
3. Others.

Each has the *read, write* and *execute* capabilities.

```
user@sigma:ls -l
-rw-r--r-- 1 user group 653 2009-07-23 12:11 file.txt
```

File Permissions and attributes attacks

Escaping a chroot jail

attacker could escape from a chroot jail, and corrupt real file systems.

Race conditions on File Systems

a privileged process could be altered to access and damage file systems.

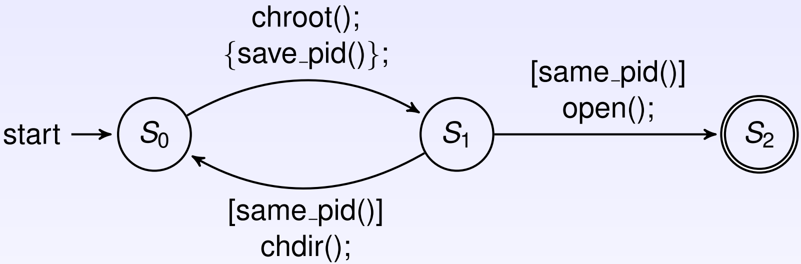
Escaping a chroot jail

Chrooting:

- It's a combination of two words: change and root.
- Changes the root directory of logged-on users or applications.

Problem:

- After call to `chroot`, `chdir("/")` should be called.
- Any open-like system calls, immediately after `chroot` could open real system files.



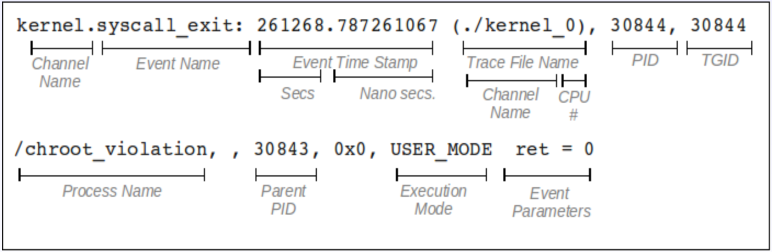

```
chroot("/home/hamow1/myjail");
open("../../../etc/passwd",O_RDONLY);
```

```
user@sigma:sudo chroot /home/hamow1/myjail
```

LTTng Trace Details

To convert the trace into text format, use `textDump` module:

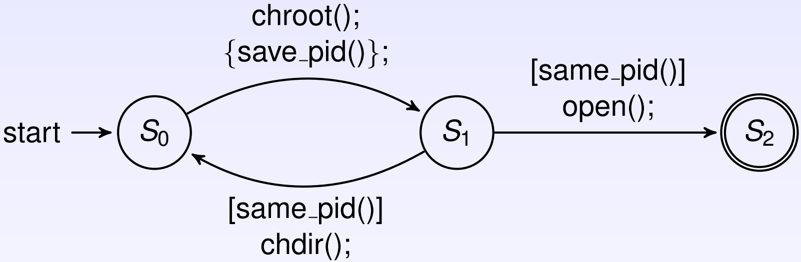
```
user@sigma:lttv -m textDump -o ascii_file.txt -t trace_directory
```



LTTng Trace Details

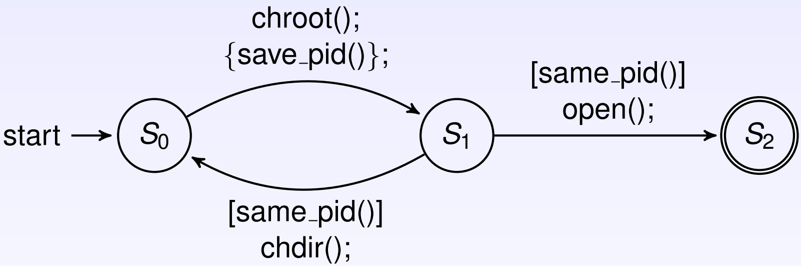
1. kernel.syscall_entry: 261268.787261067 (./kernel_0), 30844, 30844, ./chroot_violation, , 30843, 0x0, SYSCALL ip = 0xb800e430, syscall_id = 61 [sys_chroot+0x0/0xa0]
2. kernel.syscall_exit: 261268.787275718 (./kernel_0), 30844, 30844, ./chroot_violation, , 30843, 0x0, USER_MODE ret = 0
3. kernel.syscall_entry: 261268.787742811 (./kernel_0), 30844, 30844, ./chroot_violation, , 30843, 0x0, SYSCALL ip =0xb800e430, syscall_id = 5 [sys_open+0x0/0x40]
4. fs.open: 261268.787755723 (./fs_0), 30844, 30844, ./chroot_violation, , 30843, 0x0, SYSCALL fd = 3, filename = "../../../../etc/passwd"
5. kernel.syscall_exit: 261268.787757648 (./kernel_0), 30844, 30844, ./chroot_violation, , 30843, 0x0, USER_MODE ret = 3

Language properties



1. Scenario based on multiple events.
2. Conditional transition.
3. Variables.
4. Grouping

Discussion

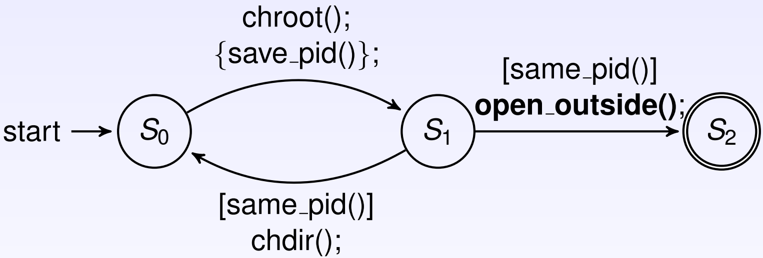


Discussion

False Alarms

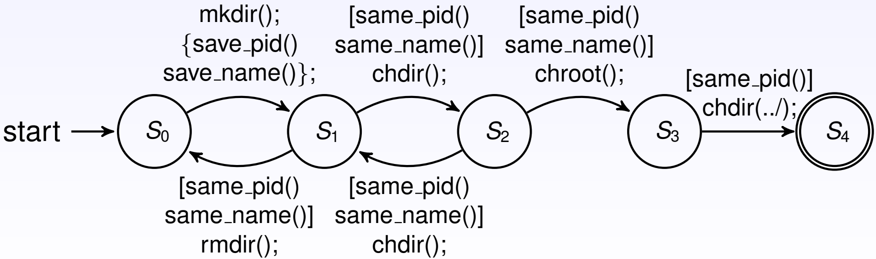
If the user opened a normal file, or a file inside the jail.

Discussion


chroot();
{save_pid()};
[same_pid()]
open_outside();
start
S_0
S_1
S_2
[same_pid()]
chdir();

Alternate Attack

- Attacker needs to have a root permission in the chrooted environment.
- Create a new folder within the chrooted environment.
- Change directory into that folder, and sets the folder as the new chroot directory.
- Perform `chdir(../)` to escape from a chroot jail, and attacker is now able to navigate the true file system and even has a root access.



Good Traces

- The behavior of a normal user doing the same functionality.
- Not always an easy task.
- Normally it's not a single instance.

```
user@sigma:sudo chroot /home/hamow1/myjail
```

Good Traces

1. kernel.syscall_entry: 14881.772973238 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 61 [sys_chroot+0x0/0xa0]
2. kernel.syscall_exit: 14881.773144700 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0
3. kernel.syscall_entry: 14881.773175093 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 12 [sys_chdir+0x0/0x80]
4. kernel.syscall_exit: 14881.773178827 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0
5. kernel.syscall_entry: 14881.7731785057 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 213 [sys_setuid+0x0/0xe0]
6. kernel.syscall_exit: 14881.14690258819 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0

Good Traces

1. kernel.syscall_entry: 14881.772973238 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 61 [sys_chroot+0x0/0xa0]
2. kernel.syscall_exit: 14881.773144700 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0
3. kernel.syscall_entry: 14881.773175093 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 12 [sys_chdir+0x0/0x80]
4. kernel.syscall_exit: 14881.773178827 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0
5. kernel.syscall_entry: 14881.7731785057 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, SYSCALL ip = 0xb7f99430, syscall_id = 213 [sys_setuid+0x0/0xe0]
6. kernel.syscall_exit: 14881.14690258819 (./kernel_0), 10409, 10409, /usr/sbin/chroot, , 8345, 0x0, USER_MODE ret = 0

Plan

1. Introduction
2. Malicious Traces
   - Security Patterns
   - Testing Programs
   - System Performance
   - Discussion
3. Scenario Description Languages
   - Domain Specific Languages
     - Declarative DSL
     - Imperative DSL
   - Automata-Based Languages
   - Temporal Logic
   - Expert Systems
   - Discussion

Plan

Using File Descriptors

- Detection of software bugs, inefficient code.
- Cause performance degradation.
- Very difficult to detect in multi-core, and distributed systems.

Everything in a Linux is a file

A set of common errors:

- Accessing a file descriptor that has been closed.
- Accessing a file descriptor that has not been opened.
- Not closing a file at the end of operation.
- Opening a file and not using it in any read/write operations.

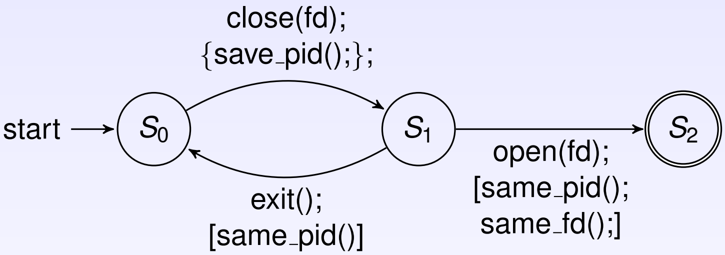
Using File Descriptors

- Detection of software bugs, inefficient code.
- Cause performance degradation.
- Very difficult to detect in multi-core, and distributed systems.

Everything in a Linux is a file
A set of common errors:

- Accessing a file descriptor that has been closed.
- Accessing a file descriptor that has not been opened.
- Not closing a file at the end of operation.
- Opening a file and not using it in any read/write operations.

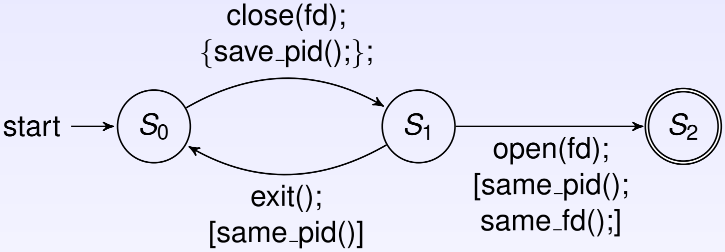
FSM & Code




```
fd = open("/home/hashem/test2.txt", O_RDONLY);
close(fd);
read(fd,buff,length);
```

LTTng Trace Details

1. kernel.syscall_entry: 6632.973601582 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ip = 0xb7f68430, syscall_id = 5 [sys_open+0x0/0x40]
2. fs.open: 6632.973606875 (./fs_0), 7103, 7103, ./test, , 6369, 0x0, SYSCALL fd=3, filename = "/home/hashem/test2.txt"
3. kernel.syscall_exit: 6632.973607677 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ret = 3
4. kernel.syscall_entry: 6632.973609431 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ip = 0xb7f68430, syscall_id = 6 [sys_close+0x0/0xf0]
5. fs.close: 6632.973610248 (./fs_0), 7103, 7103, ./test, , 6369, 0x0, SYSCALL fd = 3
6. kernel.syscall_exit: 6632.973612598 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ret = 0
7. kernel.syscall_entry: 6632.973613891 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ip = 0xb7f68430, syscall_id = 3 [sys_read+0x0/0xb0]
8. kernel.syscall_exit: 6632.973614247 (./kernel_0), 7103, 7103, ./test, , 6369, 0x0, USER_MODE ret = -9



1. Scenario based on multiple events.
2. Conditional Transitions.
3. Variables.
4. Grouping.

Good Traces

```
if( access(filename, W_OK) == 0 ){
  if( open(filename, O_WRONLY) == -1){
    perror(filename);
    return(0);
  }
  //Manipulate with the fd
  write(fd, "hello \n",6);
  close(fd);
}
```

Good Traces

- kernel.syscall_entry: 141730.167331518 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL ip = 0xb7f2c430, syscall_id = 33 [sys_access+0x0/0x30]
- kernel.syscall_exit: 141730.167276820 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, USER_MODE ret = 0
- kernel.syscall_entry: 141730.167331518 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL ip = 0xb7f2c430, syscall_id = 5 [sys_open+0x0/0x40]
- fs.open: 141730.167336200 (./fs_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL fd=3, filename = "/tmp/x"
- kernel.syscall_exit: 141730.167336977 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, USER_MODE ret = 3
- kernel.syscall_entry: 141730.167338546 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL ip = 0xb7f2c430, syscall_id = 4 [sys_write+0x0/0x3b0]
- fs.write: 141730.167360780 (./fs_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL count = 6, fd = 3

Good Traces

- kernel.syscall_exit: 141730.167361125 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, USER_MODE ret = 6
- kernel.syscall_entry: 141730.167363096 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL ip = 0xb7f2c430, syscall_id = 6 [sys_close+0x0/0x40]
- fs.close: 141730.167363898 (./fs_1), 6227, 6227, ./write, , 5870, 0x0, SYSCALL fd=3
- kernel.syscall_exit: 141730.167366575 (./kernel_1), 6227, 6227, ./write, , 5870, 0x0, USER_MODE ret = 0

Plan

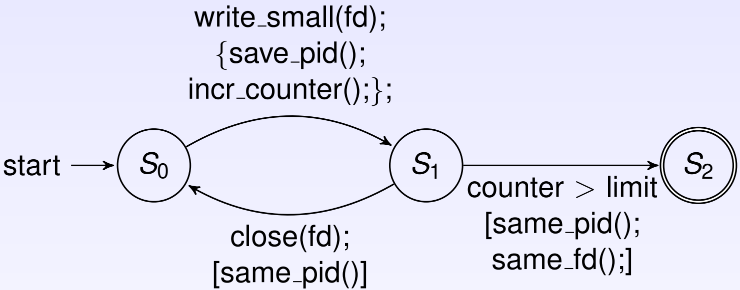
System Performance

- Inefficient I/O
  - Frequent writing of small chuncks of data.
  - Writing latency (timeout) to disk (maybe due to disk saturation, ...).
  - Reading twice the same data.
  - Reading the data that has been just written to disk.
- Real-time applications constraints.

System Performance

- Inefficient I/O
  - Frequent writing of small chuncks of data.
  - Writing latency (timeout) to disk (maybe due to disk saturation, ...).
  - Reading twice the same data.
  - Reading the data that has been just written to disk.
- Real-time applications constraints.

Inefficient I/O




```
fd = open("/home/hashem/test2.txt", O_RDONLY);
for(i=0;i<100;i++){
  write(fd,"a",1);
}
```

LTTng Trace Details

1. `kernel.syscall_entry: 103158.477573944 (./Trace/kernel_0), 19035, 19035, ./perf, , 17201, 0x0, USER_MODE ip = 0xb7f6d430, syscall_id = 5 [sys_open+0x0/0x40]`
2. `fs.open: 103158.47758115 (./Trace/fs_0), 19035, 19035, ./perf, , 17201, 0x0, SYSCALL fd = 3, filename = "/home/hashem/test2.txt"`
3. `kernel.syscall_exit: 103158.477582114 (./Trace/kernel_0), 19035, 19035, ./perf, , 17201, 0x0, USER_MODE ret = 3`
4. `kernel.syscall_entry: 103158.477582836 (./Trace/kernel_0), 19035, 19035, ./perf, , 17201, 0x0, USER_MODE ip = 0xb7f6d430, syscall_id = 4 [sys_write+0x0/0xb0]`
5. `fs.write: 103158.477637465 (./Trace/fs_0), 19035, 19035, ./perf, , 17201, 0x0, SYSCALL count = 1, fd = 3`
6. `kernel.syscall_exit: 103158.477582114 (./Trace/kernel_0), 19035, 19035, ./perf, , 17201, 0x0, USER_MODE ret = 1`

Plan

Discussion

1. Scenario based on multiple events.
2. Conditional Transitions.
3. Variables.
4. Grouping.
5. Counting.
6. Real-time constraints.
7. Non-Occurrence of events.
8. Synthetic events.

Name	1	2	3	4	5	6	7	8
chroot jail	✓	✓	✓	✓	-	-	-	-
Abusing setuid	✓	✓	✓	✓	-	-	-	-
Race condition	✓	✓	✓	✓	-	-	✓	-
SYN Flood	✓	✓	✓	✓	✓	✓	-	✓
File descriptors	✓	✓	✓	✓	-	-	-	-
Writing small data	✓	✓	✓	✓	✓	-	-	-

Table: Language properties

Plan

Scenario Description Language

Automating the detection of faulty behavior needs a simple and unambiguous language.
The languages are divided into the following categories:

1. Domain Specific Languages.
   - Declarative DSL.
     - Rule-Based Languages (snort, and SECnology).
     - Policy-based Languages (Blare, and BlueBox).
   - Imperative DSL (RUSSEL, BRO, DTrace, and SystemTap).
2. Automata-Based Languages (STATL, State Machine Compiler, Ragel, BSML, and IDIOT).
3. Temporal Logic Languages (ADele, Chronicle, and LogWeaver).
4. Expert systems (P-Best, and Lambda).

Plan

Domain Specific Languages (DSL)

DSL are dedicated to solve a particular problem or implement a well-defined domain task.

Declarative DSL

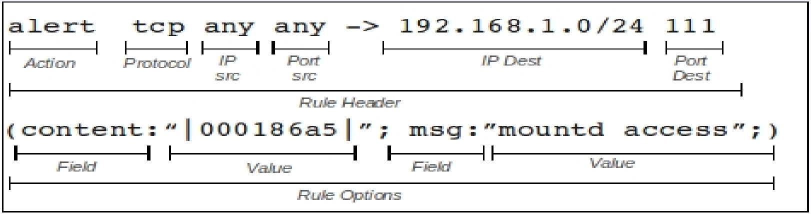
Describes **what** is to be done, the logic of computation.

Imperative DSL

Describes **How** something could be done, the control flow of the program.

Snort 2009

- Free, open-source, and Well-known system used Network-Based Intrusion Detection System (NIDS).
- Could be used as packet-sniffer, packet logger and NIDS.
- Network packets are checked agains the occurence of specific values in **fields**.
- If found a specific **action** should be taken.
- Snort is based on one packet (event) evaluation.



Discussion

Using Declarative DSL in Kernel Tracing.

- Writing patterns at high level of abstratction (no awareness about implementation).
- High speed detection (one event evaluation).
- Cannot represent patterns based on multiple events.

RUSSEL 2006

- RUle-baSed Sequence Evaluation Language.
- Used in audit trace analysis as part of ASAX IDS.

```
rule Failed_login (maxtimes , duration :  integer)
#This rule detects a first failed login and triggers off
#an accordig rule with an expiration time
begin
if evt='login' and res='failure' and is_unsecure (terminal)
   -->Trigger off for next Count_rule1 (maxtimes-1, timestp+duration)
fi;
Trigger off for next Failed_login ( maxtimes , duration)
end;
```

RUSSEL Rule

```
rule Count_rule1 (countdown , expiration :  integer)

#This rule counts the subsequent failed logins,

#it remains active until its expiration time or until the countdown becomes 0

if evt='login' and res='failure'

and is_unsecure(terminal) and timesto < expiration

   -->if countdown > 1

     -->Trigger off for next Count_rule1(countdown-1, expiration);

     countdown=1

     -->SendMessage("too much failed login's")

fi;

   timestp >= expiration

   Skip;

   --> Skip;

   true

   -->Trigger off for next Count_rule1(countdown, expiration);

fi;
```

Discussion

- Provides a mechanism for relating different events.
- Intrusion detection domain related.
- Only one active rule is available at a time (Rule triggering mechanism).

Plan

- Language used in STAT for IDS.
- STATL is translated into C++.
- Contains a lot of extensions like: NetStat, WinStat, LinStat, ... (Contains a set of pre-defined scenarios)
- Visualization tool could be used.
- provide Timers.

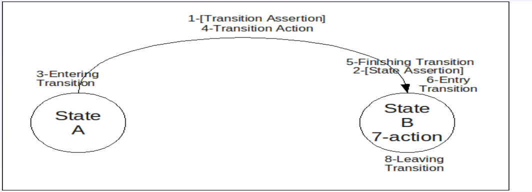


Figure: General Finite State Machine Architecure

- Three types of transitions:
  1. Consuming: Normal transition, system changes it state.
  2. Non-Consuming: Create a copy of the system state, and then moves to next state.
  3. Unwinding: Delete all states.

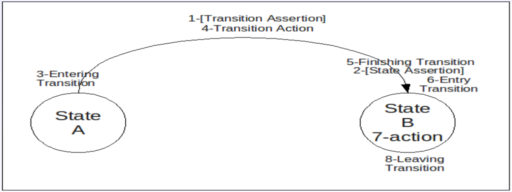


Figure: General Finite State Machine Architecure

Scenario Example

```
use netstat;
scenario halfopentcp(int timeout)
{
   IPAddress victim_addr;
   Port victim_port;
   IPAddress attacker_addr;
   Port attacker_port;
   timer t0;
   initial state s0 {}
   transition SYN (s0 -> s1) nonconsuming
   {
      [IP ip [TCP tcp]] :
      (tcp.tcp_header.flags & TH_SYN) && !(tcp.tcp_header.flags & TH_ACK)
      {
         victim_addr=ip headerdst;
         victim_port=tcp headerdst;
         attacker_addr=ip headersrc;
         attacker_port=tcp headersrc;
      }
   }
   state s1
   {
      { timer_start(t0, timeout); }
   }
   transition ACK (s1 -> s0) unwinding
   {
      [IP ip [TCP tcp]] :
      (ip headerdst==victim_addr) && (tcp headerdst==victim_port) &&
      (ip headersrc==attacker_addr) && (tcp headersrc==attacker_port) &&
      !(tcp header.flags & TH_SYN) && (tcp header.flags & TH_ACK)
   }
   transition RST (s1 -> s0) unwinding
   {
      [IP ip [TCP tcp]] :
      (ip headersrc==victim_addr) && (tcp headersrc==victim_port) &&
      (ip headerdst==attacker_addr) && (tcp headerdst==attacker_port) &&
      (tcp header.flags & TH_RST)
   }
   transition Timed_out (s1 -> s2) consuming
   {
      timer t0;
   }
   state s2
   {
      {
         HALFOPENTCP e;
         e = new HALFOPENTCP(attacker_addr, attacker_port, victim_addr,
         victim_port, start);
         enqueue_event(e, HALFOPENTCP, start);
      }
   }
}
```

Discussion

- Provide a simple, efficient and expressive way for describing a wide-variety of attacks.
- Very applicable to Trace Analysis.
- Conversion of code to C++ make it more powerful.
- Describe efficiently complex attacks.

Plan

Chronicle

- Temporal logic that permits the recognition of chronics in a flow of events.
- Verified by the online system ”Chronicle Recognition System”.
- Chronicle operators:
  - **hold(P;v;(t;t2))**: The attribute P holds the value of v, in the interval t1 to t2.
  - **event(P;(v1;v2);):** the attribute P changes from value v1 to v2 in time t.
  - **event(P;t)**: The attribute p occurs at time t.
  - **noevent(P;(t1;t2))**: The value of attribute P has not changed in the interval (t1,t2)
  - **occurs((n1;n2);P;(t1;t2))** In the interval (t1,t2), the attribute t occurs n1 to n2 times.

Chronicle Scenario example

```
chronicle portscan[source,target]{
      event(alarm[sid_1, source, target], t1)
      occurs(1,+∞, alarm[sid_2,source,target], (t1+1,t2))
      noevent(1,+∞, alarm[sid_2,source,target], (t1,t2))
      event(alarm[sid_3,source,target], (t2+1))
     t1 < t2
     when recognized {
        emit event(alarm[portscan, source, target], t2)
     }
}
```

Discussion

- Valid for the trace analysis.
- The time-constraints between events is important in a lot of attacks.
- The *non-occurence* of events.
- The context of events (hold).
- The Counting (occurs).
- Un-wise use of memory could cause performance degradation and even memory explosion.
- Generates a lot of alarms of the same problem (multiple instances).

Plan

Lambda

- Part of exploit systems.
- Describes the attack from the attacker point of view.
- Each attack is divided into the following:
  - pre-condition.
  - scenario.
  - post-condition.

Scenario example

attack attack_name($arg_1, arg_2, \ldots$)
pre : ϕ_{pre}
post : ϕ_{post}
scenario : ϵ_s
where : ψ_s
detection : ϵ_d
where : ψ_d
verification : ϵ_v
where : ψ_v

où ϕ_i est une formule de la logique du deuxième ordre
ψ_i est une formule de la logique du premier ordre
ϵ_i est une formule du calcul des événements

Action *touch(Agent,File)* Pre : *true* Post : *file(File), owner(Agent,File)*	Action *block(Agent,Printer)* Pre : *printer(Printer), physical_access(Agent,Printer)* Post : *blocked(Printer)*
Action *lpr-s(Agent,Printer,File)* Pre : *printer(Printer), file(File), authorized(Agent,read,File)* Post : *queued(File,Printer)*	Action *remove(Agent,File)* Pre : *owner(Agent, File)* Post : not *(file(File))*
Action *ln-s(Agent,Link,File)* Pre : not *(file(Link))* Post : *linked(Link,File)*	Action *unblock(Agent,Printer)* Pre : *printer(Printer), blocked(Printer), physical_access(Agent,Printer)* Post : not *(blocked(Printer))*
Action *print-process(Printer,Link)* Pre : *queued(Link,Printer), linked(Link,File),* not *(blocked(Printer))* Post : *printed(Printer,File),* not *(queued(Link,Printer))*	Action *get-file(Agent,File)* Pre : *printed(Printer,File), physical_access(Agent,Printer)* Post : *read_access(Agent,File)*

Discussion

- Interesting in the trace analysis
- Accumulation, inference and decision making is useful to detect maybe unkown attacks.
- Interesting way of dealing with synthetic events (Knoweldge database).
- Simple way of describing attacks (pre, scenario and post conditions).
- Describing the attacks from the attacker point of view.

Plan

Discussion

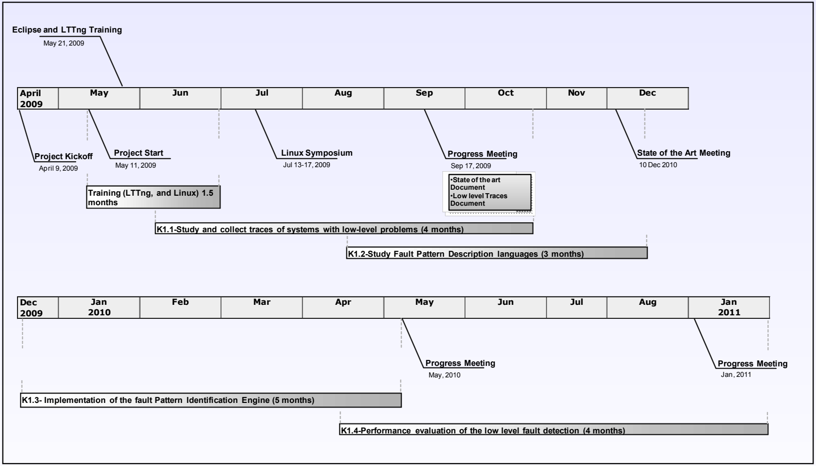
Studied so far 18 different languages.

- Sequence of event.
- Non-occurrence of events.
- Time constraint.
- Number of occurrence of an event
- Context sensitive.
- Online analysis.
- Simplicity.
- Suitable for kernel tracing.
- Possibility of inferring new facts.

Name	1	2	3	4	5	6	7	8	9
Snort	-	-	-	X	X	X	-	-	-
SECnology	-	-	-	-	-	Y	Y	-	-
Blare	X	X	X	X	Y	-	X	-	-
BlueBox	X	X	X	X	X	X	-	X	-
RUSSEL	X	-	X	X	-	-	-	-	-
BRO	X	X	X	X	X	-	-	-	-
DTrace	X	X	X	X	X	-	-	-	-
SystemTap	X	X	X	X	X	-	-	-	-
STATL	X	X	X	X	-	-	X	X	-
SMC	X	X	X	X	X	-	-	-	-
Ragel	X	X	X	X	X	-	-	-	-
BSML	X	X	X	X	X	-	-	-	-
IDIOT	X	X	X	X	X	-	-	-	-
ADele	X	X	X	X	X	-	-	-	-
Chronicle	X	X	X	X	X	X	-	X	-
LogWeaver	X	X	X	X	X	-	-	-	-
P-Best	X	X	X	X	X	-	-	-	-
Lambda	X	X	X	-	X	-	X	-	X

1. Sequence of event.
2. Non-occurrence of events.
3. Time constraint.
4. Number of occurrence of an event
5. Context sensitive.
6. Online analysis.
7. Simplicity.
8. Suitable for kernel tracing.
9. Possibility of inferring new facts.

Future Work


Eclipse and LTTng Training
May 21, 2009
April 2009
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Project Kickoff
April 9, 2009
Project Start
May 11, 2009
Linux Symposium
Jul 13-17, 2009
Progress Meeting
Sep 17, 2009
State of the Art Meeting
10 Dec 2010
Training (LTTng, and Linux) 1.5 months
•State of the art Document
•Low level Traces Document
K1.1-Study and collect traces of systems with low-level problems (4 months)
K1.2-Study Fault Pattern Description languages (3 months)
Dec 2009
Jan 2010
Feb
Mar
Apr
May
Jun
Jul
Aug
Jan 2011
Progress Meeting
May, 2010
Progress Meeting
Jan, 2011
K1.3- Implementation of the fault Pattern Identification Engine (5 months)
K1.4-Performance evaluation of the low level fault detection (4 months)

Thank You