A Signature Scheme for Distributed Executions based on Macro-Dataflow Analysis

S. Varrette, B. Bertholon and P. Bouvry
University of Luxembourg

<Firstname.Name@uni.lu>
Summary

- Context & Motivation
- Related work
- Checkable Signature of distributed execution flow
  - offline fingerprint generation
  - online signature generation & verification
- Implementation & experimentations
- Conclusion
Context

• Execution over a large-scale distributed platform
  ‣ Computing grid, desktop grid, Cloud
    - heterogeneous (processor, network ...)
    - dynamic (failures, reservations ...)

• Bad things happens
  ‣ [D]DoS, malware, trojan horse, vulnerability exploit etc.
    - Crash, buffer overflows, machine-code injection

• Global Purpose: ensure execution integrity
Execution model

- [Parallel] program $P$ executed over $M$
  - single machine, grid etc.
- Abstract representation of the distributed execution of $P$
  - Bipartite DAG $G = (V, E)$
    - $V = V_t \cup V_d$
- execution of $T$ in $P$ unfold $G(T)$
- the set of all $G(T)$ characterize $G$
Distributed execution & dataflow graphs

- Permits to handle various class of fault
  - crash-fault: efficient checkpoint/rollback [CCK:BesseronGautier08]
  - cheating-fault: handle task forgery/result falsification
    - efficient detection on FJ/recursive programs [Varrette07]
    - avoid full program duplication yet costly in general
  - flow-fault: result of malicious code injection
    - general manifestation of cheating faults
Distributed execution & dataflow graphs

- Permits to handle various class of fault
  - crash-fault: efficient checkpoint/rollback [CCK:BesseronGautier08]
  - cheating-fault: handle task forgery/result falsification
    - efficient detection on FJ/recursive programs [Varrette07]
    - avoid full program duplication yet costly in general
  - flow-fault: result of malicious code injection
    - general manifestation of cheating faults

⇒ In this talk: flow-fault detection in distributed computations
Related work

- prior to a remote execution
  - Static analysis/malware fingerprint detection [Christodorecu&al.05]
  - Proof Carrying Code (PCC) [NeculaLee97]
  - Control-flow checking at the assembly level [Abel05]
Related work

• prior to a remote execution
  ‣ Static analysis/malware fingerprint detection [Christodorecu&al.05]
  ‣ Proof Carrying Code (PCC) [NeculaLee97]
  ‣ Control-flow checking at the assembly level [Abel05]

⇒ does not cover dynamic attacks in distributed environment
Related work (2)

- Control-flow integrity on sequential execution
  - Operate at the assembly level
  - include result-checking [Castro et al. 06]
  - with graphs (node~block) & XOR signature [Oh et al. 02]
Related work (2)

- Control-flow integrity on sequential execution
  - Operate at the assembly level
    - include result-checking [Castro&al06]
    - with graphs (node~block) & XOR signature [Oh&al02]

⇒ extension at middleware level to distributed computation
Definition 1 (Flow fault). Let $G^{ref}$ denotes the (reference) fault-free execution of $P$ over $M$. Let $G$ be the representation of an execution of $P$ over $M$. Then $G$ is said faulty or victim of a flow fault if the graphs $G$ and $G^{ref}$ differs i.e. $G \cap G^{ref} \neq \emptyset$. Otherwise, $G$ is said correct.

2 phases approach:
1. Offline fingerprint (reference signature) generation
2. Online signature generation & verification
Offline fingerprint

- Based on source code analysis (C/C++) (extends CFG)
- For each task $T$ (function): build a NFA $\mathcal{A}_T$
  
  Path $\text{Begin} \rightarrow \text{End} = \text{valid flow}$
  
  state = sub-task called in $T$
  
  transition $s_i \rightarrow s_j = t_j = H(s_j)$
  
  - derived from the graph unfold
  
  - Special transition $H(\text{nil})$
  
  - implicit transition ‘\’ to the Error state
Offline fingerprint

- Structure of control impact on the fingerprint
Offline fingerprint

- Once all $A_T$ are generated:
  - **optimization phase**
    - accelerate future online verification for long path
    - transition values derived from intermediate values
    - not mandatory
Fingerprint example

\begin{verbatim}
void f1 (int n) {
    if (...) {
        f2(n) ;
        return;
    }
    f2(n−2);
    f3();
}
\end{verbatim}

- Demonstrate the non-deterministic aspect of $A_{f_1}$
  - conflict handled as GLR parser do
- No optimization operated here
Hash value construction

**Definition 2 (Flow hash).** Let $G$ represents an execution of $\mathbf{P}$ over $M$. Let $T \in \mathcal{V}_t$. The flow hash associated to the execution of $T$ is defined by

$$H(T) = (\text{prototype}, \text{flow_detail})$$

- **prototype**: function signature (C sense)
  - Ex: see `__PRETTY_FUNCTION__`
- **flow_detail**: summary of the execution flow of $T$
  - data-flow graph unfolded at execution of $T$
  - should correspond to $G^{ref}(T)$
Dynamic hash building

```c
int f(int a, int b){
    int c = g(a);
    int d = h(a,b);
    return h(c,d);
}
```

Graph traversing (sequential execution order)

Execution dataflow graph unfolding

flow_detail: $G(f)$

Signature automaton for $f$
generated from $G^{ref}(f)$
Execution engine

- **Hypothesis**: *TRUSTABLE* execution engine
  - dynamic construction of the macro-dataflow graph
  - online dynamic task scheduling by work stealing
  - Execution agents spread on the resources of the [distributed] computing platform
Signature verification

• Fully distributed & recursive process
  ▶ Agent/Process P responsible to execute \( f \) (called in F)
  ▶ \( f \) composed by sub-tasks \( f_1, \ldots, f_n \) / \( f_i \) executed on \( P_i \)
    - the \( P_i \) may be different processors
    - after execution of \( f_i : P_i \) returns \( H(f_i) \) to P which check:
      • \( H(f_i).\)prototype is correct (later used to feed \( H(f) \))
      • \( H(f_i).\)flow_detail permits to reach state \( \text{End} \) in \( A_{f_i} \)
    - after execution of all \( f_i \) and successful signature verification:
      • \( H(f) = [\text{Compress}]H(f_1).\)prototype||...||H(f_n).\)prototype
Execution agent on
\(P\)
\[
\text{int } t = f(14, 27); 
\]

\[
\text{int } f(int, int)
\]
\[
\begin{array}{c}
\text{int } a = 14 \\
\text{int } b = 27 \\
\end{array}
\]

\[
\begin{array}{c}
\text{int } c = g(a) \\
\text{int } d = h(a, b) \\
\end{array}
\]

\[
\text{return } h(c, d);
\]

Execution agent on
\(P_i\)
\[
\text{return computed } \text{H(h)}
\]

Execution agent on
\(P_j\)
\[
\text{write result}
\]

\[
\text{Execution agent on}
\]

\[
\text{int } t = f(14, 27);
\]

\[
\text{int } f(int, int)
\]
\[
\begin{array}{c}
\text{int } a = 14 \\
\text{int } b = 27 \\
\end{array}
\]

\[
\begin{array}{c}
\text{int } c = g(a) \\
\text{int } d = h(a, b) \\
\end{array}
\]

\[
\text{return } h(c, d);
\]
Proposition 1. As soon as the execution of the program \( P \) ends, the verification process ends in a finite time.

Proposition 2. Let \( G^{\text{ref}} \) denotes the (reference) fault-free execution of \( P \) over \( M \). Let \( \{A_{T_1}, \ldots, A_{T_n}\} \) denotes the set of automaton signatures elaborated from the analysis of \( P \)'s source code. Let \( G \) be the representation of an execution of \( P \) over \( M \). Then \( G \) is faulty \( \iff \exists i \in [1, n] \) such that the verification process of the automaton signature \( A_{T_i} \) ends in the \textit{Error} state.
Verification process properties

Proposition 1. As soon as the execution of the program $P$ ends, the verification process ends in a finite time.

Proposition 2. Let $G^{ref}$ denotes the (reference) fault-free execution of $P$ over $M$. Let $\{A_{T_1}, \ldots, A_{T_n}\}$ denotes the set of automaton signatures elaborated from the analysis of $P$’s source code. Let $G$ be the representation of an execution of $P$ over $M$. Then $G$ is faulty $\iff \exists i \in [1, n]$ such that the verification process of the automaton signature $A_{T_i}$ ends in the Error state.

$\Rightarrow$ any flow fault is detected assuming trustable agents
Implementation

• Based on Kaapi

  ▸ C++ middleware library for distributed computing
  ▸ Build dynamic macro-dataflow graph
  ▸ High level interface with global address space
    - Data (Shared<...>): declares an object in the global space
    - Tasks (Fork<...>): declares a new [concurrent] task
    - Access mode given by the task (read, write, exclusive etc.)

http://kaapi.gforge.inria.fr
Athapascan interface of Kaapi

#include <athapascansc-1>

int Fiboseq(int n); // Sequential version

void Sum(Shared_w<int> res, Shared_r<int> res1, Shared_r<int> res2) { res = res1+res2; }

void Fibo(Shared_w<int> res, int n, int threshold int n) {
    if (n < threshold)
        res = Fiboseq(n);
    else {
        Shared<int> res1;
        Shared<int> res2;
        /* the Fork keyword is used to spawn new task */
        Fork<Fibo>(res1, n−1, threshold);
        Fork<Fibo>(res2, n−2, threshold );
        Fork<Sum>(res, res1, res2);
    }
}

4.2 Implementation details of the proposed signature scheme

As precised in the section our approach involves two different phases, one off-line to generate the code fingerprint, the other on-line to check the signature in a fully distributed process.

Off-line fingerprint generation.

As mentioned in §upvn a set of Nonodeterministic Finite Automata \(A_jT_k\) are generated off-line for each task \(T\) that compose the source code of the kaapi program. In this context, we developed a dedicated software that parses the preprocessed code of the program job obtained via the GNU Compiler Collection [tv] and the command g++ -E k with the Cmm parser Elsa [tt] and the Generalized LR parser Elkhound [sx]. From the parse tree it creates the signature automaton for each function as describe in §upvk. NFAs are stored in encrypted files under the dot format [z]. They are read at run time by the kaapi agents who hold the key to decrypt them to operate the signature verification process. The dot format simplified the automatic generation of a human-readable view of the signatures.
Offline fingerprint generator

- Permit to generate the NFA $A_T \forall T$ in $\mathbb{P}$
- Analyse Kaapi source code
  - exploit preprocessed code by GCC
  - C++ parser Elsa & Generalized LR parser Elkhound
  - NFA stored encrypted in DOT format
  - decrypted at runtime for signature verification
Offline fingerprint in action

```c
int main() {
    ...
    for (int i=0; i < MAX; i++)
        Fork<Fibo>(res, n, threshold);
    ...
}
```

```c
#include <athapscan-1>

int Fiboseq(int n); // Sequential version

void Sum(Shared_w<int> res, Shared_r<int> res1, Shared_r<int> res2) { res = res1+res2; }

void Fibo( Shared_w<int> res, int n, int threshold int n) {
    if (n < threshold)
        res = Fiboseq(n);
    else {
        Shared<int> res1;
        Shared<int> res2;
        /* the Fork keyword is used to spawn new task */
        Fork<Fibo>(res1, n-1, threshold);
        Fork<Fibo>(res2, n-2, threshold);
        Fork<Sum>(res, res1, res2);
    }
}
```
Online signature verification

• Add a new internal task to Kaapi execution engine
  ‣ TaskVerification - responsible to:
    - check sub-tasks execution flow (using associated NFAs)
    - build the hash (in the verif shared data) to be returned to the mother task handler

• Fully transparent to the user
  ‣ extension of the middleware library
Online signature verification

- Affect the data-flow graph unfolded

  Example:
Experimental validation

- Validation on one of the clusters of UL
  - 16 computing nodes, Intel Dual Core 3.2Ghz, 4G RAM

- Two applications evaluated:
  1. Naive fibonacci
     - illustrate massive task creation (worst case for us)
     - granularity controlled by the threshold parameter
  2. N-Queens
     - parallel implementation based on sequential code by Takaken
Experiments #1: Fibo(39)

\[ R_{\text{overhead}} = \frac{\text{Execution time with signature checking}}{\text{Execution time without signature checking}} \]

Finally it is important to notice that the approach is fully transparent to the user as the implementation details provided in this paragraph have been applied at the middleware level.

5 Experimental Validation

The proposed signature scheme has been validated on two typical applications: Fibonacci and NQueen. Version plr of the kaapi library have been used for those experiments which have been conducted on the clusters of the University of Luxembourg. Each computing node can have one of the following configurations:

- C1: Intel Dual Core Pentium Dual Core 2.8 GHz and 2 GBytes of main memory
- C2: Two Intel Xeon Quad Core 2.6 GHz and 8 GBytes of main memory

5.1 Fibonacci computation.

A first set of experiments of the folk recursive Fibonacci number computation has been executed based on the code provided in the listing olol This benchmark program demonstrates a configuration with massive task creation, which is the worst configuration for our signature scheme as every new task created is associated with a verification procedure. The granularity of the program is fully controlled by the threshold parameter: a small value increases drastically the number of forked tasks, letting the sequential "leaf" functions of the dataflow graph, i.e., Fibonacci tasks with little work to operate. On the contrary, bigger values for the threshold limits the number of spawned tasks and makes the sequential functions longer, i.e., able to cover the task creation process or in our case, the signature checking operation. This aspect is illustrated in Figure (a) where the Fibonacci program is evaluated for different values of the parameters \( n \) and threshold on one or two computing nodes each in configuration C1.

Fig. 9. Overhead of the embedded signature checking process when executing the Fibonacci program (a) absolute (b) relative (for configuration C1 and 4 cores).
Experiments #1: Fibo(39)

- up to 5.5x overhead
  - huge yet hold in unused parameter area
- for a more realistic threshold (0.5n):
  - relatively low overhead
- scalable approach!
  - Fibo(39)=10^8 tasks checked
Experiments #1bis: Fibo(42)

- **Speedup evaluation (threshold=20)**

![Graph showing speedup evaluation for different configurations of Kaapi.](image)
Experiment #2: N-Queens

![Graph showing the relationship between N (chess board size) and time (s) for signature checking activated and disabled.]
Conclusion

• Signature scheme to detect flow faults in distributed computations via macro data-flow analysis
  ▶ offline fingerprint generation by code analysis
  ▶ online distributed & recursive verification
  ▶ fully transparent to the user + working implementation

• Assumes trustable execution agents
  ▶ [Re-trust contribution] investigate way to get ride of this
Last word: conference future

• Mentioned by Yoram
  ▷ C. Collberg wanted to create a more formal conf.
  ▷ ReTrust2008: idea to join our effort on this issue
    - also on board now: Yuan Gu (Cloackware), Paolo Falcarin (P.Torino)

• Current plan: Workshop at ACM Conference on Computer and Communications Security (CCS) 2010
  ▷ Topic: Software protection and Secure computation
    • Paper submission: April 2010 / Conf: Nov 2010
  ▷ A+ conference, kindly join the program committee!