Verification Strategies

SHRUTI PATEL, Indian Institute of Technology Dharwad

A hardware trojan, when present in an integrated circuit, may alter its intended function or cause it to perform an additional malicious operation. However, to add additional capabilities to our application, we must rely on third party cores. The document provides methods for safe incorporation of the third party cores into a system. The third party core actions are further verified by trusted home grown cores. The verification strategy, DMR shows a performance increase of 10% to 35% compared to individual home-grown core when run on a heterogeneous Odroid board.

1 INTRODUCTION

Third-party cores allows us to add multiple abilities to your application. However, even if the vendor claims that the core is secure, it may not necessarily comply with safety standards. The presence of a Hardware Trojan in the core is highly possible if the vendor company is a rival company or if it can gain something by using your confidential information. However, this may be a major threat to your application. A trojan can not be detected passively. We therefore need to find ways to prevent the trojan attacks and their effects from spreading to the other parts of the system.

SecCheck presents four different strategies for the verification of results by 3PC namely, Dual Modular Redundancy, DIVA, Invariant and Extended DIVA. DIVA and extended DIVA are not feasible on an Odroid board. The document shows the analysis carried out on the DMR strategy. The Odroid board is made up of 16 complex cores and 16 simple cores. Complex cores may be considered as third-party cores, and simple cores may be considered as home-grown cores to carry out the experiments.

2 RELATED WORK

SecCheck [1] is an architectural framework that allows us to integrate untrusted third-party cores into the system, along with simple, trusted home-grown cores, in order to realize a trustworthy system as a whole. The goal of SecCheck is to prevent miscalculations from affecting the other parts of the system. The HGC shall verify the results of the 3PC. If a miscalculation has been detected, a trojan is present in the third-party core. If such a situation arises, we do not allow the miscalculated task to proceed further and use only HGC to proceed with the application. Alternatively, we can restart the 3PC so that the trojan triggering condition is not met because the preconditions for activation of trojans is a very rare event. Any communication with the devices takes place only after the tasks have been fully verified by the HGC. Therefore, no confidential data or malicious signals can be leaked from the third-party core.

2.1 Application Model

SecCheck adopts a task-graph model. A task is a sub-program with a single entry point, and a single exit point. It can take zero or more inputs and zero or more outputs. A program is represented as a directed graph, where each node is a task. The tasks receive their inputs and send their outputs through messages from/to each other. The dependencies form the edges of the graph. The task graph is processed in top to bottom order. Figure 1 and 2 represent the task graph description language.

Author's address: Shruti Patel, 160010002@iitdh.ac.in, Indian Institute of Technology Dharwad.

```
TASK src ID 1
TASK fft ID 2
TASK fir ID 3
TASK angle ID 4
TASK matrix ID 5
TASK ifft ID 6
TASK road ID 7
TASK table ID 8
TASK sink ID 9
```

Fig. 1. Graph Nodes

```
ARC a2_0 FROM 1 TO 2
ARC a2_1 FROM 1 TO 3
ARC a2_2 FROM 2 TO 5
ARC a2_0 FROM 3 TO 4
ARC a2_0 FROM 5 TO 6
ARC a2_0 FROM 6 TO 4
ARC a2_0 FROM 4 TO 7
ARC a2_0 FROM 7 TO 8
ARC a2_0 FROM 8 TO 9
```

Fig. 2. Graph Edges

2.2 Verification Strategies

There are four different verification strategies mentioned in the SecCheck document for tasks to be verified. For each strategy, one instance of the task runs on a 3PC and the other instance runs on an HGC.

- **Dual Modular Redundancy :** This strategy requires the primary task instance running on the 3PC and the other instance running on the HGC. Since the HGC performs slower than the 3PC, the results can only be verified after HGC has completed its execution. Here, the time for verification is high.
- **Invariant :** For some applications where we know the solution, we can check whether or not the result is correct by invariant. For such tasks, the 3PC is used to solve the problem while the HGCs verify the results. In this case , the time of verification is much shorter than the strategies mentioned above.
- **DIVA :** The primary instance of a task runs on a 3PC and the other instance runs on an HGC. The 3PC shares execution hints to the HGC. Since, 3PC has to send additional information to the HGC, the time to complete the execution in 3PC increases, but the time taken to verify the result by HGC decreases. This strategy, however, requires both instances to run simultaneously. This strategy is not feasible because the core itself needs to be changed. This type of strategy is useful for tasks that require continuous interaction with the outside world.
- **Extended DIVA :** Extended DIVA is similar to DIVA, but it uses one more instance of 3PC. The verification time of DIVA is less than that of DMR, but the execution time is longer than DMR. We therefore combine the two strategies to make the most of the two strategies. One task runs on a 3PC, the other instance running on 3PC which sends execution hints and the third instance running on a HGC. It uses 3 resources, hence it is the most resource intensive. It can therefore be used when the graph is being processed in a narrow breadth phase.

3 EVALUATIONS

3.1 Preparation of benchmarks

An E3S application is a task-graph, where each task is an EEMBC benchmark, and the edges represent data dependency between the nodes. Each EEMBC benchmark was modified to read its input from a socket and write its output to another socket. The socket numbers should be passed as arguments.

3.2 A stand-alone application

A stand-alone application was hand-coded to process the task graph. The application consists of a master and multiple slaves. The master application is responsible for assigning tasks to the slave applications. A task is assigned to a slave only after all its inputs are ready. A slave application performs the task and returns the result to the master application.

3.3 Scheduling Scheme

The algorithm uses ASAP Scheduling Scheme. The task graph is processed in top to bottom order. Whenever, the resource becomes free, we schedule the next task as soon as possible. If multiple tasks are ready to be executed, we schedule any one of the tasks available.

3.4 Algorithms

Algorithm 1: 3PC Scheduler Algorithm

```
/* This algorithm is run on a single 3PC, the 3PC Master.                    */
Input: A task-graph of the format above
Function schedule_tasks():
    /* Go through the tasks in topological sort order                        */
    Store each task's in-degree ;
    while there are no more tasks to be scheduled do
        Find a task with in-degree zero ;
        if any 3PC is free then
            Schedule the task on the free 3PC by sending a message <task name, args> to it.;
            Mark the 3PC as busy.;
        end
    end
end
Function receive_results():
    /* Receive results from 3PC slaves and stores the result                 */
    while while there are running tasks that needs to complete do
        Wait for an incoming message <task name, result> from any 3PC ;
        if message received then
            Save the result.;
            Mark the 3PC as free;
            Send the result to the HGC master. ;
            Reduce the in-degree of all its children tasks by one. ;
        end
    end
end
process1 = schedule_tasks() ;
process2 = receive_results() ;
Execute process1 and process2 in parallel.
```

Algorithm 2: 3PC Slave Algorithm

```
/* This algorithm is run on every 3PC.                                    */
while True do
    Receive message of the form <task name, args> ;
    Execute the task ;
    Send message <task name, task result> to 3PC Scheduler.
end
```

Algorithm 3: HGC Scheduler Algorithm

```
/* This algorithm is run on a single HGC, the HGC Master                  */
Input: A task-graph of the format above
Function schedule_tasks():
    while all the tasks are verified do
        /* Find a task t, whose predecessor tasks are the set P, such that
           for each p in P, p has either completed execution on a 3PC or an
           HGC                                                            */
        for t in tasks do
            P = predecessors(t);
            all_predecessors_executed = True ;
            for p in P do
                if p not executed in 3PC or HGC then
                    all_predecessors_executed = False ;
                end
            end
            if all_predecessors_executed then
                break;
            end
        end
        if any HGC is free then
            Schedule the task t on the free HGC by sending a message <task name, args> to
              it.;
            Mark the HGC as busy.;
        end
    end
end
```

Algorithm 4: HGC Slave Algorithm

```
/* This algorithm is run on every HGC                                     */
while True do
    Receive message of the form <task name, args> ;
    Execute the task ;
    Send message <task name, task result> to HGC Scheduler.
end
```

Algorithm 5: HGC Comparator Algorithm

```
/* This algorithm is on a single HGC, the HGC Master simultaneously with
   HGC scheduler algorithm                                              */
Function receive_results():
    while all the tasks are verified do
        Listen for message <task name, task result> from a 3PC or an HGC;
        if received from 3PC then
            Save the result;
            if the task is executed by HGC then
                if the results match then
                    Mark the task as verified;
                else
                    Go to HGC-only fail-safe mode ;
                    Schedule the task and its successors again on an HGC;
                end
            end
        end
        if received from HGC then
            Save the result;
            Mark the HGC free;
            if the task is executed by 3PC then
                if the results match then
                    Mark the task as verified;
                else
                    Go to HGC-only fail-safe mode ;
                    Schedule the task and its successors again on an HGC;
                end
            end
        end
    end
end
```

3.5 Hardware Description

The experiments are run on Intel Core i7-5500U processor. The processor has 2 cores. Each core can support two simultaneous threads of execution. Thus, the CPU has 2 cores that together support 4 threads. Each thread is also called a "logical core". Thus, 2 physical cores and 4 logical cores. The logical cores 0 and 2 belong to physical core (core ID) 0 and the logical cores 1 and 3 belong to physical core 1.

3.6 Experiments

Third party cores are faster than home-grown cores. Let us say that the time taken by the application to run on the HGC is k times the time taken by the 3PC. But when we run an application on 3PC and verify the results of each task, the time taken should be less than that of HGC-only. The following sections present the analysis of time taken for securely running an application on a combination of 3PCs and HGCs.

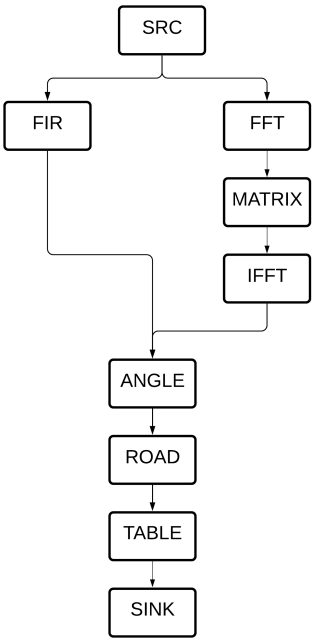

Fig. 3. Application 1

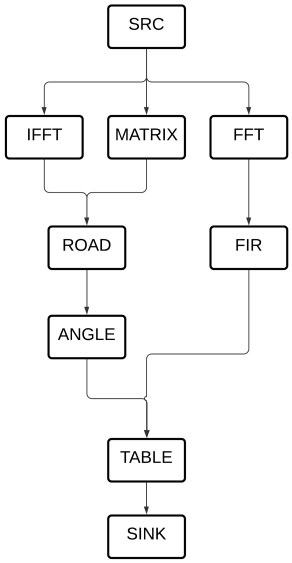

Fig. 4. Application 2

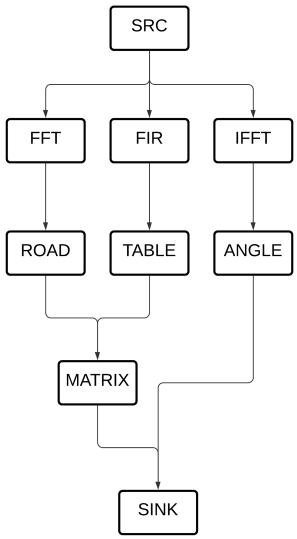

Fig. 5. Application 3

The experiments are performed for the task graphs shown in the figures 3, 4 and 5. The EEMBC benchmarks used in the graph are Fast Fourier Transform (FFT), Inverse Fast Fourier Transform (IFFT), Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Road Speed (ROAD), Table Lookup (TABLE), and Matrix Decomposition (MATRIX).

Each benchmark is modified, such that it takes a n 1-dimensional array as input and gives a 1-dimensional array as output. Here, n is the number of parents of a task. The experiments are conducted on a single chip with 4 cores. To simulate the execution time of the application, manual sleeps are induced for some cores, which works as home-grown cores.

3.6.1 *Time taken by individual benchmarks on 3PC and HGC.*

The time taken by the stand-alone benchmarks to run on an home-grown core is approximately 4.5-5 times greater than that of a third-party core.

Task	3PC time (in s)	HGC time (in s)	k = $\frac{HGC}{3PC}$
FFT	0.0665	0.326	4.9
FIR	0.0941	0.423	4.49
ANGLE	0.1085	0.553	5.1
IFFT	0.0897	0.417	4.64
MATRIX	0.0332	0.144	4.33
ROAD	0.0427	0.188	4.4
TABLE	0.0495	0.21	4.24

Table 1. Individual benchmarks run time

3.6.2 *Time taken by the applications to run on 3PC.*

The time taken by the applications to run on a single 3PC core is very high because the single core as master and slave itself. The time taken to run the applications on 3 third-party cores and 4 third-party cores is approximately the same. It is because the maximum number of tasks that can run in parallel are three. Therefore, the performance does not increase on increasing the number of cores beyond three. Refer figure 6, 7 and 8.

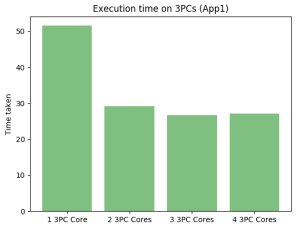

Fig. 6. Application 1

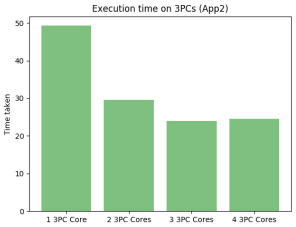

Fig. 7. Application 2

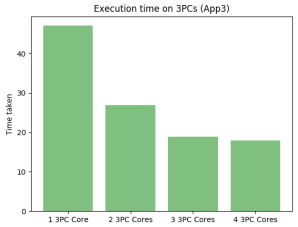

Fig. 8. Application 3

3.6.3 *Time taken by the applications to run on HGC.*

The time taken by an application to execute on home-grown cores is nearly 3 to 4 times that of third-party cores. The time taken to execute the task graph on 3 cores and 4 cores is nearly same because of the same aforementioned reason. Refer figure 9, 10 and 11.

3.6.4 *Time taken by the applications to run securely on 3PC and HGCs combined.*

When a single 3PC core is used with HGC cores to perform the same computation, the time taken is lesser than HGC-only mode and greater than 3PC-only mode.

- When 1 3PC core is used along with 3 HGC cores, 1 HGC core works as a master/scheduler. The other two cores works as slaves. Since the 3PC performs computations faster, the results obtained early for the tasks. Hence, the HGC can use the results of 3PCs to continue processing the graph. In this case, we can perform more than two tasks in parallel. hence, the amount of multi-processing is high and hence, the time taken is very less compared to HGC-only

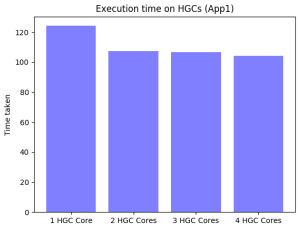

Fig. 9. Application 1

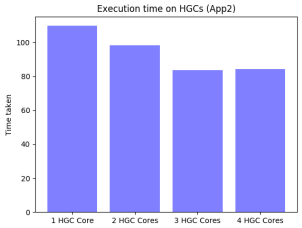

Fig. 10. Application 2

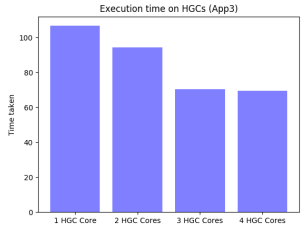

Fig. 11. Application 3

mode. When application 1 runs on HGC-only mode, one core is always idle. Hence, when it receives results of other tasks from 3PC, it can schedule it to the idle core. However, in case of application 2 and 3, results of tasks received from 3PC serves little purpose because the HGC cores are always busy. Therefore, the performance gain obtained in application 1 is high compared to the other applications,

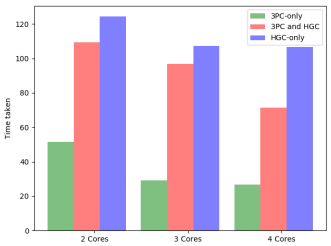

Fig. 12. Application 1

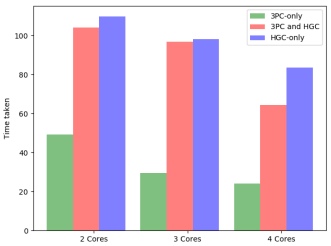

Fig. 13. Application 2

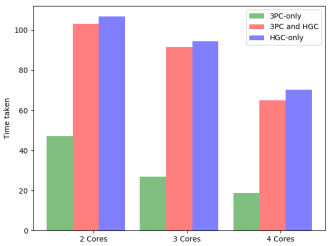

Fig. 14. Application 3

- When 1 3PC core is used along with 2 HGC cores, 1 HGC core works as a master/scheduler. The time taken in this case is higher than the previous combination of cores because the level of multi-processing decreases due to reduced number of cores. The performance gain is very less due to the overhead of receiving results from 3PC.
- When 1 3PC core is used along with 1 HGC core, the HGC core acts both as a scheduler and master. Since the HGC core works both as a master and slave, there is almost no performance gain compared to HGC-only mode execution in all the applications.

3.6.5 ***Report and Graph Generation.*** The system generates a report as given in figure 15. The columns in the report represent :

- task name
- the core on which the task is scheduled
- start time
- end time

On analysing the report, we can find which tasks ran in parallel. In the given report, the tasks IFFT, MATRIX and FFT started at the same time on different cores and hence they executed simultaneously. The visual representation of the report is presented in the figure 16. The report and the graph are corresponding to application 1 when 1 3PC core and 3 HGC Cores are used.

```
src     0       4.01    10.24
fft     1       9.02    16.53
fir     0       12.24   23.62
angle   0       35.64   49.11
matrix  0       25.63   33.63
ifft    1       30.63   38.64
road    1       40.64   54.12
table   0       51.11   62.19
sink    1       60.13   68.01
```

Fig. 15. Report

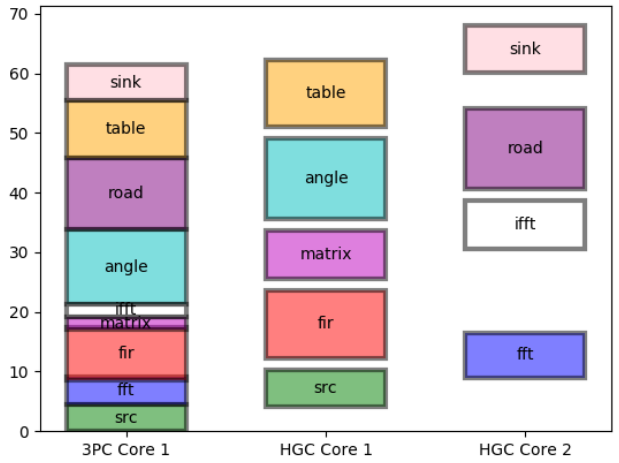

Fig. 16. Graph

4 CONCLUSIONS

When only 1 or 2 HGC cores are used for verification, the performance gain is not noticeable. When 3 HGC cores are used for verification, the performance gain varies from 10% to 35%.If we use more number of HGCs the performance gain will increase further. However, if we keep on increasing the number of HGCs after a certain number, the overhead of scheduling and listening on the cores will increase and the performance gain will decrease.

5 FUTURE WORK

- Automate the script by giving parameters (number of 3PCs, HGCs, IPs, port numbers, etc.)
- Think of a different scheduling strategy which can add to the performance gain.

6 LEARNT IN THE PROCESS

- Sockets can communicate through different languages.
- Due to the extensive use of sockets, I overcame the fear of sockets.

REFERENCES

[1] Rajshekar Kalayappan and Smruti R. Sarangi. 2016. SecCheck : A Trustworthy System with Untrusted Components. *IEEE* (2016).