SecCheck : A Trustworthy System with Untrusted Components

Rajshekar Kalayappan and Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
{*rajshekark,srsarangi*}*@cse.iitd.ac.in*

***Abstract*—Mission critical applications face a security risk when they use third-party ICs for their speed and/or technology benefits. *SecCheck* is an architectural framework that securely incorporates fast, untrusted third-party cores (3PCs). It takes a comprehensive approach, providing for all of the different traditional fault tolerance techniques, to verify the 3PCs' functioning. The verification is done at run-time by slow, trusted, homegrown cores (HGCs). The overhead of providing security is reduced through intelligent scheduling exploiting task-level parallelism. The average performance penalty for achieving security under *SecCheck* is just 10-17% (optimal schedule), even when the HGCs are only half as fast as the 3PCs. We also devise a heuristic-based scheduler that is 500X faster than an ILP-based optimal one, with a relative penalty less than 1%.**

***Keywords*-security; trojan; redundancy; scheduling; heterogenous**

I. Introduction

The failure of Syria's radar systems was instrumental in Israel's successful air strike on it in 2007 [1]. The failure is believed to have been brought about by *Hardware Trojan Horses* (HTHs) in the control systems. Governments across the world are beginning to take this problem very seriously. The United States Department of Defense has launched the Trusted Foundry Program [2], attempting to implement the entire design and fabrication process within a small group of trusted organizations. The resulting systems however fall short of the global IC industry in many aspects such as yield, turnaround time, cost and technology [3]. An organization, by itself, cannot produce chips that are advanced enough to meet its application requirements. The reliance on third-party ICs is, at least in the immediate future, unavoidable.

SecCheck is an architectural framework that allows the secure incorporation of high performance, untrusted, third party, general purpose cores (3PCs). There are four types of attacks an HTH may carry out [4]: (i) wrong computation, (ii) malicious signaling of devices, (iii) denial of service (DoS) and (iv) leakage of data. In this work, we primarily focus on detecting whether a 3PC miscomputed a task, and on preventing the effects of this attack from reaching the rest of the system. Thus *SecCheck* handles all attacks of the first class, and to a large extent the other classes as well. All communication with devices are made only after they are fully verified, thereby preventing malicious signaling of devices and leakage of data (akin to the gateway-based approaches of [5], [6]). Incorporating watchdog timers will help detect DoS attacks.

The detection of an HTH through pre- and post-silicon techniques is extremely hard [7]. Trojans can lie dormant for long periods of time before becoming active. Thus run-time techniques are required to detect malicious activity. The principles behind run-time techniques are largely inspired by fault tolerance research (see the survey by Kalayappan et.al [8]): dual modular redundancy (DMR) and invariant verification. The efficacy and limitations of each of these strategies by themselves has been well studied in prior works [7], [9], [10]. We observe that intelligent choice of the verification strategy based on the task at hand can significantly reduce the performance burden of security. This brings us to the first important facet of *SecCheck* : **it is a generic and comprehensive approach**, that brings together (i) DMR, (ii) invariant verification, (iii) the seminal soft-error detection technique *DIVA* [11], and (iv) a novel combination of DMR and DIVA called *Extended DIVA* (E_DIVA), all within a single framework.

The second important facet of *SecCheck* is that **it uses low performance, trustworthy, home-grown, general purpose cores (HGCs)** to verify the untrusted 3PCs. HGCs are designed and fabricated *in-house*, or within a trusted group of organizations like that realized by the Trusted Foundry Program [2]. Prior DMR-based strategies like [9], [10] employ two functionally-equivalent ICs from two different third parties and check if their results match. These strategies are based on the assumption that the two third parties do not collude. Since *SecCheck* need not rely on such assumptions, it provides a stronger security guarantee. Additionally, to the best of our knowledge, there is no other work that has studied heterogeneous systems of this kind, with different trust domains at the granularity of processor cores. Prior works using trusted components are typically simple invariant checkers [7], [12], [13].

The different verification strategies that *SecCheck* employs are known to provide complete security. However, since the HGCs are slower and/or simpler than the 3PCs, simple application of these redundancy techniques results in large performance overheads, negating the benefit of the 3PCs. The solution to counter this gives *SecCheck* its third facet. ***SecCheck* seeks to mask the overhead of verification by overlapping the 3PCs' computation with the HGCs' verification**. This is possible due to the task-level parallelism exposed by the popular task-based application model. With the right verification strategy for each task and the right mapping of task instances to cores, the overhead of redundancy can be masked significantly. Our experiments show that an optimal schedule of the application on the *SecCheck* system results in a performance overhead as low as 10% (as compared to an insecure 3PC-only run) when the HGCs are half as fast as the 3PCs. In contrast, a system consisting of only HGCs displays a 100% overhead. We admit that *SecCheck* has high area and power overheads – these are expected traits of all redundancy-based architectures.

The optimal scheduler is an ILP-based one and hence is quite slow. Based on certain non-trivial heuristics, we propose the *SecCheck Scheduling Algorithm* (SSA). SSA computes near-optimal schedules (average penalty $< 1\%$), while being much faster ($\approx$500X) than the optimal scheduler.

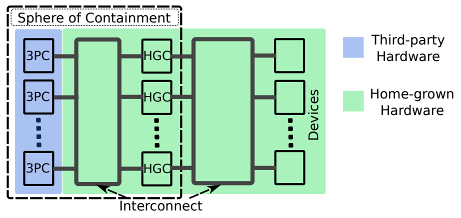


Figure 1: *SecCheck* Hardware Architecture

II. Related Work

HTHs can be inserted at any point during the design or the fabrication phase. Pre-silicon design verification techniques are impractical – RTL-level simulators are too slow, while formal verification techniques have scalability issues. Post-silicon functionality testing faces coverage issues. Side-channel analyses require a *golden* model against which the chip under test is compared. Process variation phenomena force the *golden* model to be loosely specified. Also, **pre- and post-silicon techniques are ineffective in the face of triggered HTHs [7]**.

Given the shortcomings of both pre-silicon and post-silicon techniques, run-time techniques are being sought that can detect a HTH when it is activated in-field. Prior run-time techniques can be classified into two categories: redundancy-based and invariant checking. Redundancy-based techniques such as [9], [10] suggest that multiple functionally equivalent IPs/ICs be procured from different vendors. Comparing their results at run-time tells us if a Trojan was activated. **These works are based on the assumption that the different third party vendors do not collude**. Given the sensitive nature of mission critical applications, we believe this assumption to be too risky. *SecCheck* suggests using home-grown HGCs to verify the 3PCs. Even though the HGCs are much slower and/or simpler than the 3PCs, clever utilization of them can ensure minimal impact to the performance. To the best of our knowledge, **no prior work attempts to employ home-grown circuitry to verify complex foreign ICs such as processor cores**. The general approach in using home-grown security elements is to verify certain properties or invariants associated with the third-party IC [3], [7], [13]. Simple functionalities such as bus protocols allow complete coverage through elegant invariants. However, in more sophisticated designs like processor cores, not all sub-functionalities display elegant invariants.

III. SecCheck Architecture

A. Hardware Architecture and Attack Model

Figure 1 shows the hardware architecture. The system contains N_{3PC} number of 3PCs and N_{HGC} HGCs. The system also contains devices that may be read from or written to by the HGCs. The 3PCs are potentially malicious and may miscompute the task assigned to it. A result derived from a miscomputed task must not reach the devices; only fully verified read and write requests must do so. Any malicious 3PC activity must be detected and contained within the "*sphere of containment*" (which is the system of cores; see Figure 1). We term this

The Principle of Containment: *Any attack by a Trojan must be contained within the sphere of containment and not allowed to affect the rest of the system.*

Once the Trojan is detected, one or more of a set of measures (not exhaustive) may be carried out: (i) move to a fail-safe HGC-only mode (ii) disable the guilty 3PC (iii) restart the guilty 3PC in a bid to upset the triggering condition.

B. Application Model

Prior works in the domain of such multi-SoC systems that possibly contain heterogeneous computing elements, commonly assume the task based application model [9], [10] (survey in [14]). A *task* is a sub-program with a single entry point, and a single exit point. Regular tasks are self-contained, and do not have side effects. A program is represented as a graph of tasks, where tasks receive their inputs and send their outputs via messages from/to each other. Due to the message passing model, it is easy to parallelize such programs. Additionally, task graphs present isolation properties desirable when fault tolerance and/or security are design goals. Acknowledging the emergence of this paradigm at the level of embedded processors, GPUs, and servers, several vendors have launched programming languages/runtimes in this space namely Ericsson's Erlang, Intel's Cilk+ and TBB, Microsoft's Task Parallel Library, Google's Go and IBM's X10. These paradigms have software runtimes that govern the execution of the task graphs on the available processors, and the movement of data between tasks. *SecCheck* also adopts the task graph model, and assumes a software runtime (running on an HGC) that manages the execution of the tasks and their instances (see Section III-C), the movement of data between them, and the performing of I/O. Just like in other works in this domain, we evaluate our proposal using the standard E3S benchmark suite [15], which contain task graphs composed of benchmarks from the standard EEMBC suite [16], as well as synthetic graphs generated using TGFF [17] (also used in [9]).

Formally, a task graph is a directed acyclic graph $G = (V, E)$. The set of vertices V represents tasks. An edge $e = (u, v), e \in E$ indicates that the output of task u forms the input of task v. Consequently, u must complete before v begins. A task may have zero or more inputs, and zero or more outputs. All outputs of a task are available at the same time at the end of the task's execution.

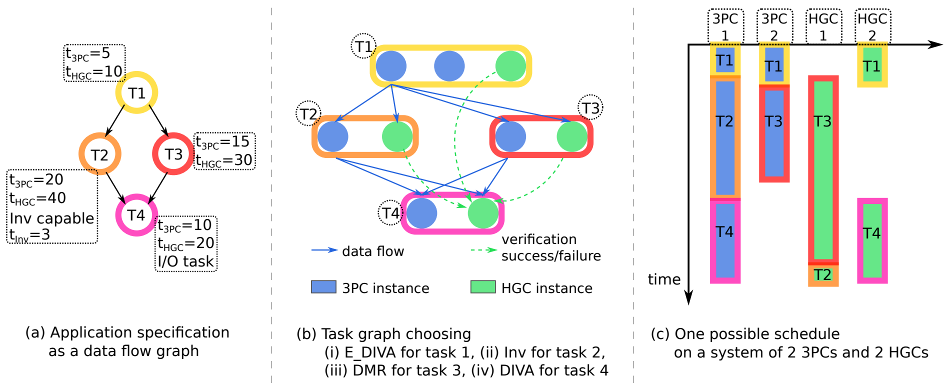
I/O tasks: Some of the tasks access devices. To realize the *principle of containment*, an I/O task is allowed to complete only when it and all its ancestors have been verified.

Task annotations: Each task is also annotated with (i) an estimate for the time taken to execute on a 3PC, t_{3PC} (notation: t will be used to describe time intervals, and $\mathbb{T}$ to describe absolute times), (ii) an estimate for the time taken to execute on an HGC, t_{HGC}, (iii) an estimate for the time taken to execute under the DIVA scheme (see Section III-C), t_{DIVA}, (iv) whether it is invariant capable (see Section III-C), (v) an estimate for the time taken to verify under the invariant scheme, t_{INV}. These annotations are provided by the programmer and/or a profiling compiler.

Figure 2(a) shows the specification of an example application. The task graph model for applications reveals *task-level parallelism* (TLP) that is critical for *SecCheck* . The parallelism across different tasks, as well as their verification instances (see Section III-C), is the reason for *SecCheck* 's superior performance.

C. Verification Strategies

A task can be verified through any of four verification strategies (or execution types), each having its pros and cons. At run-time, for each task, multiple execution instances are spawned. The number of instances, the cores on which they run, and their run duration depend upon the verification strategy chosen. Appropriate selection of the verification strategy for each task can reduce the time to securely execute the application. For all strategies, there exists one instance that runs on a 3PC termed the *primary*



t_{3PC}=5
t_{HGC}=10
T1
T2
T3
T4
t_{3PC}=15
t_{HGC}=30
t_{3PC}=20
t_{HGC}=40
Inv capable
t_{Inv}=3
t_{3PC}=10
t_{HGC}=20
I/O task
(a) Application specification as a data flow graph
data flow
verification success/failure
3PC instance
HGC instance
(b) Task graph choosing (i) E_DIVA for task 1, (ii) Inv for task 2, (iii) DMR for task 3, (iv) DIVA for task 4
3PC 1
3PC 2
HGC 1
HGC 2
time
(c) One possible schedule on a system of 2 3PCs and 2 HGCs

Figure 2: Application specification and execution in *SecCheck*

instance. Data dependencies are resolved through the primary instances. That is, all the instances of the successor tasks of a task i can begin only after the primary instance of i completes. The time required for this completion is denoted by t_{dep_i}. A task i can be deemed verified only after all its instances have completed. This time is denoted by t_{ver_i}.

Dual Modular Redundancy (DMR): Under DMR, the primary instance runs on a 3PC and another runs on an HGC, and their outcomes are compared. This gives $t_{dep} = t_{3PC}$, allowing quick spawning of successor tasks. However, the verification time is high, $t_{ver} = t_{HGC}$. This allows for potential overlapping of the task's verification and the execution of its successors. Note that the executions of the 3PC and HGC instances need not overlap.

DIVA: DIVA [11] verifies by assisted re-execution. Two instances of the task are run simultaneously, one on a 3PC (*leader*), the other on an HGC (*checker*). The leader passes execution hints to the checker thereby improving the latter's performance. The time taken for the instances to finish, t_{DIVA}, typically follows the inequality $t_{3PC} < t_{DIVA} < t_{HGC}$. Under DIVA, $t_{dep} = t_{ver} = t_{DIVA}$. From a resource utilization standpoint, DIVA results in a longer 3PC occupation (than DMR), and a shorter HGC occupation.

Invariants (INV): Applications that exhibit easy-to-verify invariants are termed *invariant capable*. The class of NP problems is a classic example. The fast 3PC can be used to solve the problem, and the slower HGC can be used to quickly verify the result. Under this technique, $t_{HGC} << t_{3PC}$, making $t_{ver} \approx t_{dep} \approx t_{3PC}$.

Extended DIVA (E_DIVA): DIVA's $t_{ver}(= t_{DIVA})$ is lower than that of DMR. However, DIVA's $t_{dep}(= t_{DIVA})$ is greater than that of DMR. To capture the best of both worlds, we propose extended DIVA, where three execution instances of the task exist: one running standalone on a 3PC to help dependent tasks start as soon as possible, and two instances running in DIVA mode. This gives $t_{ver} = t_{DIVA}$ and $t_{dep} = t_{DMR}$. However, since E_DIVA employs three instances, it is more resource intensive as compared to the other techniques, and therefore must be prudently used. Note that it is also verified that the output of the HGC instance matches with the stand-alone 3PC instance.

Figure 3 shows a comparative summary of the techniques.

D. *Offline Scheduling*

Prudent scheduling of tasks on cores improves performance. Alternatively, it reduces the number of HGCs required to attain a certain level of performance. A commonly used metric to evaluate performance is the *makespan* – the time elapsed between the

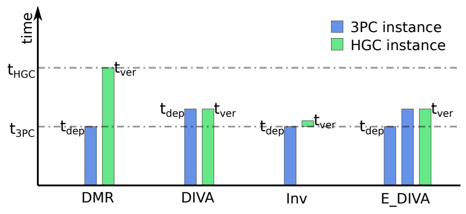


Figure 3: Comparison of different verification techniques

starting of the first task and the completion of the last task. This is ideal if only the result of the very last task is to be considered. However, as commonly seen in typical applications, many intermediate tasks also perform I/O. Thus, a more apt metric would be one that favors a highly responsive system. We choose the popular *average completion time* metric [18] – the arithmetic mean of the completion times of the different I/O tasks. Formally,

Task-verification: a task is said to be *task-verified* when all its execution instances have completed execution. Let the time when a task i is task-verified be given by $\mathbb{T}_{ver_i}$.

Output-verification: a task is said to be *output-verified* when it is *task-verified* and all its predecessors are *output-verified*. Let the time when a task i is output-verified be given by $\mathbb{T}_{OPver_i}$. $\mathbb{T}_{OPver_i} = \mathbb{T}_{ver_i}$ for root tasks.

The performance metric that must be minimized is:

$$PM = \frac{\sum_{v \in V_{IO}} \mathbb{T}_{OPver_v}}{|V_{IO}|} \quad (1)$$

where V_{IO} is the set of all I/O tasks.

The scheduler is responsible for determining for each task, (i) what verification technique must be employed, (ii) when the different instances should be scheduled, and (iii) on which cores must the different instances be scheduled.

In the example in Figure 2, panel (b) shows the different 3PC and HGC instances required if the scheduler determines E_DIVA to be used for task '1', INV for task '2', DMR for task '3' and DIVA for task '4'. Panel (c) shows one possible schedule of the application on a system having 2 3PCs and 2 HGCs. Prudent scheduling can help increase the temporal overlap of different instances, and therefore reduce the value of PM. For instance, it can be seen that the HGC instance of task 3 overlaps with instances of all tasks. The advantage of E_DIVA is also seen – the HGC instance of task 3 begins before task 1 has completed verification, which itself does not take too long (as compared to when the DMR strategy is employed for task 1). Note that in *INV*, the HGC instance can

Table I: Parameters in the ILP formulation

V:	set of tasks
t_{3PC_i}:	time taken to execute task i on a 3PC
t_{HGC_i}:	time taken to execute task i on an HGC
t_{DIVA_i}:	time taken by a DIVA execution of task i
IO_i:	is task i an I/O task (boolean)
INV_i:	is task i invariant capable (boolean)
t_{INV_i}:	time taken for *INV* verification of task i
E_{ij}:	is task j data dependent on task i (boolean)
N_{3PC}:	number of 3PC cores
N_{HGC}:	number of HGC cores

Table II: Variables in the ILP formulation

$VS_{ij} =$	$\begin{cases} 1, & \text{if task } i \text{ is executed by verification strategy } j \\ 0, & \text{otherwise.} \end{cases}$ where $j = 1$ refers to DMR, $j = 2$ refers to DIVA, $j = 3$ refers to INV and $j = 4$ refers to E_DIVA.
$A_{HGC_{ij}}$:	is the HGC instance of task i assigned to the jth HGC (boolean)
$\mathbb{T}_{startHGC_i}$:	start time of HGC instance of task i
$A_{3PC\_1_{ij}}$:	is the 3PC instance of task i assigned to the jth 3PC (boolean)
$\mathbb{T}_{start3PC\_1_i}$:	start time of 3PC instance of task i
$A_{3PC\_2_{ij}}$:	is the DIVA-3PC instance of task i assigned to the jth HGC (boolean); applicable only for the E_DIVA case
$\mathbb{T}_{start3PC\_2_i}$:	start time of DIVA-3PC instance of task i; applicable only for the E_DIVA case

begin only after the 3PC instance finishes. In DIVA and E_DIVA, the *leader-checker* pair have to execute at the same time.

IV. ILP Formulation of the Scheduling Problem

The **parameters** in the formulation are given in Table I. The **variables** are given in Table II. Table III lists the variables added for ease of formulation. The **objective function** is:

$$minimize \sum_{v \in V} (\mathbb{T}_{OPver_v} \times IO_v) \tag{2}$$

where $\mathbb{T}_{OPver_i}$: time when i^{th} task is output-verified.

The **constraints** of the ILP are:

✠ Every task must have exactly one verification strategy.

$$\forall_{i \in V} : \sum_{j=1}^{4} VS_{ij} = 1 \tag{3}$$

✠ A task can be assigned the *invariant* verification strategy only if it is invariant capable.

$$\forall_{i \in V} : VS_{i3} \leq INV_i \tag{4}$$

✠ The run times of the different instances of a task depend upon the verification strategy.

$$\begin{aligned} \forall_{i \in V} : t_{run3PC\_1_i} = t_{3PC_i} \times VS_{i1} + t_{DIVA_i} \times VS_{i2} \\ + t_{3PC_i} \times VS_{i3} + t_{3PC_i} \times VS_{i4} \end{aligned} \tag{5}$$

$$\forall_{i \in V} : t_{run3PC\_2_i} = t_{DIVA_i} \times VS_{i4} \tag{6}$$

$$\begin{aligned} \forall_{i \in V} : t_{runHGC_i} = t_{HGC_i} \times VS_{i1} + t_{DIVA_i} \times VS_{i2} \\ + t_{INV_i} \times VS_{i3} + t_{DIVA_i} \times VS_{i4} \end{aligned} \tag{7}$$

✠ All instances of a task can begin execution only when the primary instances of all its parents have completed.

$$\begin{aligned} \forall_{i,j \in V} : \mathbb{T}_{start3PC\_1_i}, \mathbb{T}_{start3PC\_2_i}, \mathbb{T}_{startHGC_i} \\ \geq (\mathbb{T}_{start3PC\_1_j} + t_{run3PC\_1_j}) \times E_{ji} \end{aligned} \tag{8}$$

Table III: Additional variables in the ILP formulation

$B_{ij} =$	$\begin{cases} 1, & \text{if tasks } i \text{ and } j \text{ are scheduled on the same core and } i \text{ is scheduled before } j \\ 0, & \text{otherwise.} \end{cases}$
t_{runHGC_i}:	
$t_{run3PC\_1_i}$:	run times of the different instances of task i
$t_{run3PC\_2_i}$:	

✠ DIVA instances have to be scheduled together.

$$\begin{aligned} \forall_{i \in V} : \mathbb{T}_{start3PC\_1_i} \leq M \times (1 - VS_{i2}) + \mathbb{T}_{startHGC_i} \\ \forall_{i \in V} : \mathbb{T}_{startHGC_i} \leq M \times (1 - VS_{i2}) + \mathbb{T}_{start3PC\_1_i} \end{aligned} \tag{9}$$

where M is a very large value. The DIVA pair in an E_DIVA task have to be scheduled together.

$$\begin{aligned} \forall_{i \in V} : \mathbb{T}_{start3PC\_2_i} \leq M \times (1 - VS_{i4}) + \mathbb{T}_{startHGC_i} \\ \forall_{i \in V} : \mathbb{T}_{startHGC_i} \leq M \times (1 - VS_{i4}) + \mathbb{T}_{start3PC\_2_i} \end{aligned} \tag{10}$$

✠ In an invariant verified task, the HGC instance must begin after the 3PC instance.

$$\begin{aligned} \forall_{i \in V} : \mathbb{T}_{start3PC\_1_i} + t_{run3PC\_1_i} \leq t_{startHGC_i} \\ + M \times (1 - VS_{i3}) \end{aligned} \tag{11}$$

✠ A task is output-verified only when all its instances have completed and all its parents have been output-verified.

$$\begin{aligned} \forall_{i \in V} : \mathbb{T}_{OPver_i} \geq \mathbb{T}_{start3PC\_1_i} + t_{run3PC\_1_i} \\ \forall_{i \in V} : \mathbb{T}_{OPver_i} \geq \mathbb{T}_{start3PC\_2_i} + t_{run3PC\_2_i} \\ \forall_{i \in V} : \mathbb{T}_{OPver_i} \geq \mathbb{T}_{startHGC_i} + t_{runHGC_i} \\ \forall_{i,j \in V} : \mathbb{T}_{OPver_i} \geq \mathbb{T}_{OPver_j} \times E_{ji} \end{aligned} \tag{12}$$

✠ The different instances of a task must be assigned to exactly one unit. Note that A_{3PC_2} is relevant only in the E_DIVA case.

$$\begin{aligned} \forall_{i \in V} : \sum_{j=1}^{N_{3PC}} A_{3PC\_1_{ij}} = 1, \forall_{i \in V} : \sum_{j=1}^{N_{3PC}} A_{3PC\_2_{ij}} = VS_{i4}, \\ \forall_{i \in V} : \sum_{j=1}^{N_{HGC}} A_{HGC_{ij}} = 1 \end{aligned} \tag{13}$$

✠ Tasks mapped on the same core must not execute simultaneously. For the sake of brevity, we show here only the constraint for HGCs. The constraints for the 3PC instances follow likewise.

$$\begin{aligned} \forall_{i,j \in V} : \mathbb{T}_{startHGC_i} + t_{runHGC_i} \leq \mathbb{T}_{startHGC_j} \\ + M \times (1 - B_{ij}) \end{aligned} \tag{14}$$

V. SecCheck Scheduling Algorithm (SSA)

The large computation time of the ILP scheduler entails a faster algorithm. SSA is a multi-pass greedy scheduler (Algorithm 1). The key intuitions behind SSA are:

As soon as possible (ASAP) scheduling

The task graph is processed in topological order. The tasks are allocated *ASAP* on any of the available resources. This greedy approach provides near optimal results, while skipping the exploration of more involved schedules.

Prioritizing important tasks

Tasks with more number of I/O tasks as descendants, and with more temporally proximal I/O descendants, are given higher priority. Since a highly responsive system is desired, SSA greedily tries to reach I/O nodes as quickly as possible. Formally, the definition of *Task Importance* (TI) is as follows:

$$TI_u = \sum_{v \in V} \begin{cases} \frac{1}{TT_{uv}} & \text{if } R_{uv} = 1 \text{ and } IO_v = 1 \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

where TT_{uv} is the minimum time required for task v to complete, after task u has completed. Formally it is defined as follows:

$$TT_{uv} = \begin{cases} 0 & \text{, if } R_{uv} = 0 \\ t_{3PC_v} + \max_{w \in V} TT_{uw} & \text{, if } R_{uw} = 1 \\ & \text{and } E_{wv} = 1 \end{cases} \tag{16}$$

where $R_{uv} = 1$ if task v is reachable from task u, 0 otherwise.

Basic choice of verification strategy

INV is given the highest preference as it has the most desirable values of t_{dep} and t_{ver} , and is the least resource intensive. For all I/O tasks, *DIVA* is employed as it greedily ensures the smallest t_{ver} for that particular I/O task. For all other tasks, *DMR* is employed, as it provides the lowest possible value of t_{dep}, and is

Algorithm 1 schedule()

```
function SCHEDULE
    verStratDirectives[] ← $NULL$                    ▷ used in Algorithm 2
    assign()
    optimizeIO(DMR)
    optimizeIO(E_DIVA)
    do
        improved ←False
        improved ← improved |optimizeExtend(DIVA)
        improved ← improved |optimizeExtend(E_DIVA)
        improved ← improved |optimizeStructural()
    while improved = True
```

Table IV: Parameters of the synthetic task graphs

Param	Value	Param	Value
Number of tasks	$\mathcal{N}(6, 2)$	$P[IO_i = 1]$	0.3
t_{3PC}	$\mathcal{N}(199, 102) \times 10^6$	$P[INV_i = 1]$	0.2
t_{HGC}	$2 \times t_{3PC} + 10^4$ †	Max in-deg/out-deg	2/2
t_{DIVA}	$\mathcal{N}(1.3, 0.168) \times t_{3PC}$	P[multiple roots]	0.2
t_{INV}	1		

($\mathcal{N} \Rightarrow$ normal distribution; † : 10^4 $\Rightarrow$comparison overhead)

not resource-hungry (Algorithm 2).

Moving from DIVA to DMR/E_DIVA for I/O tasks

Though counter-intuitive, it is sometimes beneficial to verify an I/O task by *DMR* instead of *DIVA* (Algorithm 3). Consider three I/O tasks u, v and w, such that $R_{uv} = R_{uw} = 1$. By changing the execution type of u from *DIVA* to *DMR*, $\mathbb{T}_{OPver_u}$ may increase. However, since t_{dep_u} has now reduced, it is possible that $\mathbb{T}_{OPver_v}$ and $\mathbb{T}_{OPver_w}$ decrease. The net effect may be a decrease in PM. By the same reasoning, a change to *E_DIVA* may also be beneficial, subject to resource availability (or if the task graph is in a “narrow” phase). SSA greedily attempts the $DIVA \rightarrow DMR/E\_DIVA$ changes in decreasing order of TI.

Moving from DMR to DIVA/E_DIVA for long interior tasks

It is sometimes beneficial to employ *DIVA* (or *E_DIVA*) for long interior non-I/O tasks (Algorithm 4). Consider a task graph with two tasks u and v, with $E_{uv} = 1$ and $IO_v = 1$. Now, let $t_{3PC_u} >> t_{3PC_v}$. This results in u dominating PM, with $\mathbb{T}_{OPver_v} = t_{ver_u} = t_{HGC_u}$. Now, if *DIVA* is employed for u, t_{ver_u} decreases to t_{DIVA_u}. This results in $\mathbb{T}_{OPver_v}$ decreasing to $t_{DIVA_u} + t_{DIVA_v}$. SSA greedily attempts the $DMR \rightarrow DIVA/E\_DIVA$ changes on those ancestor tasks u of I/O tasks v that satisfy the condition $\mathbb{T}_{OPver_v} = \mathbb{T}_{ver_u}$.

Reducing the impact of structural hazards

Employing *DIVA* instead of *DMR* for interior tasks sometimes reduces structural hazards (Algorithm 5). Since we follow a greedy *TI*-ordered allocation of resources, the utilization of 3PCs is near optimal. However, the same cannot be said for HGCs. The DMR strategy displays a large t_{HGC}. If there is a shortage of HGCs (or if the graph is in a “wide” phase), then this has a cascading effect on lower tasks. These lower tasks, in spite of having their inputs ready, are forced to delay their HGC instances, i.e, $\mathbb{T}_{startHGC_i} > \mathbb{T}_{ready_i}$ (where $\mathbb{T}_{ready_i}$ is the time at which the input data of i are ready). SSA, in topological order, tracks tasks i displaying this condition, and greedily attempts the $DMR \rightarrow DIVA$ change on the task assigned before i on the same HGC.

VI. Evaluation

The experiments consisted of five real world benchmarks from the E3S suite [15] (based on the automotive and telecom suites of EEMBC [16]), and 50 synthetic task graphs generated using TGFF [17]. The parameters of the synthetic graphs were derived from the E3S benchmarks (see Table IV). P_{IO} is the probability

Algorithm 2 assign() : assign verification strategies to tasks

```
function ASSIGN
    Q ← all root tasks
    while Q not empty do
        curTask ← remove task in Q with highest TI
        if verStratDirectives[curTask] not NULL then
            curTask.verStrat ← verStratDirectives[curTask]
        else
            if $INV_{curTask}$ =TRUE then
                curTask.verStrat ← INV
            else
                if $IO_{curTask}$ =TRUE then
                    curTask.verStrat ← DIVA
                else
                    curTask.verStrat ← DMR
        allocate cores for all instances of curTask ASAP
        add all successors of curTask, whose parents have been allocated, to Q
```

Algorithm 3 optimizeIO() : optimize I/O tasks

```
function OPTIMIZEIO(verStrat)
    candidates ← all I/O tasks
    while candidates not empty do
        candidate ← remove task in candidates with highest TI
        attemptUpdate(candidate,verStrat)
```

that a task performs I/O. Additionally, all leaf nodes are I/O nodes. P_{INV} is the probability that a task is invariant capable. t_{3PC}, t_{HGC} and t_{DIVA} of the different EEMBC benchmarks are obtained by simulation using the Tejas simulator [19]. The core configuration is based on the Intel Sandybridge microarchitecture, with 2MB LLC. For evaluation purposes, we assume that the 3PCs and the HGCs have the same architecture, with the frequency of operation of a 3PC equal to twice that of an HGC. This assumption is not an artifact of *SecCheck* and is made only for presenting the evaluation. We also account for the overhead of output comparison by adding a fixed penalty of 10k cycles to all HGC tasks. This gives $t_{HGC} = 2 \times t_{3PC} + 10^4$. Experiments were performed for a range of values of N_{3PC} and N_{HGC} – ranging from a severely resource constrained system having just 1 3PC and 1 HGC to a resource rich system having 4 3PCs and 6 HGCs. The experiments were performed on an Intel Xeon server with Intel Sandybridge based cores, operating at 2.53 GHz, with 12 MB LLC.

***SecCheck* Overhead** We shall now discuss the performance penalty of secure computation in the *SecCheck* system, relative to insecure computation. We compare the PM of *SecCheck* , $PM_{secureILP}$ (computed using GLPK solver v4.52), against the PM of a system having only 3PCs, $PM_{insecureILP}$. The latter is obtained using a reduced version of the ILP formulation in Section IV. Table V lists the average overhead $(= (\frac{PM_{secureILP}}{PM_{insecureILP}} - 1) \times 100)$ across the different real world

Table V: *SecCheck* overhead $((\frac{PM_{secureILP}}{PM_{insecureILP}} - 1) \times 100)$

N_{HGC}	1	2	3	4	5	6
N_{3PC}	**E3S benchmarks**					
1	19.17%	16.4%	16.4%	16.4%	16.4%	16.4%
2	22.08%	16.64%	15.42%	12.35%	12.35%	12.35%
3	22.08%	15.43%	15.43%	15.43%	12.35%	12.35%
4	22.08%	12.45%	12.45%	12.45%	12.35%	12.35%
	Synthetic benchmarks					
1	22.11%	17.48%	15.81%	16.81%	15.81%	15.81%
2	33.15%	13.66%	12.85%	12.85%	12.61%	12.61%
3	30.62%	11.29%	10.37%	10.21%	10.21%	10.21%
4	31.53%	10.47%	10.08%	10.08%	10.08%	10.03%

Algorithm 4 optimizeExtend() : handle extending tasks

```
function OPTIMIZEEXTEND(verStrat)
    tasks ← all tasks
    improved ← False
    while tasks not empty do
        i ← remove task in tasks with highest $\mathbb{T}_{OPver}$
        if $\mathbb{T}_{OPver_i} > \mathbb{T}_{ver_i}$ then
            candidate ← ancestor of i with $\mathbb{T}_{ver} = \mathbb{T}_{OPver_i}$
            improved ← improved |attemptUpdate(candidate,verStrat)
    return improved
```

Algorithm 5 optimizeStructural() : reduce structural hazards' impact

```
function OPTIMIZESTRUCTURAL
    Q ← all root tasks
    improved ← False
    while Q not empty do
        i ← remove task in Q with highest TI
        if $\mathbb{T}_{startHGC_i} > \mathbb{T}_{ready_i}$ then
            candidate ← previous task assigned on same HGC as i
            improved ← improved |attemptUpdate(candidate,DIVA)
        add all successors of i, whose parents have been processed, to Q
    return improved
```

and synthetic benchmarks, for different values of N_{3PC} and N_{HGC}. Unless severely constrained ($N_{HGC} = 1$), the overhead is typically between 10-17%. The *SecCheck* system provides security with a meager loss in performance, proving itself a worthy technique to incorporate 3PCs securely.

SSA Penalty SSA was found to produce near-optimal schedules, while being about 500X faster on average than the optimal scheduler. Table VI lists the average penalty ($= (\frac{PM_{SSA}}{PM_{secureILP}} - 1) \times 100$, PM_{SSA} :PM when SSA is used to compute the schedule) observed. The average penalties were less than 1% in all cases. The maximum penalty observed was 43.7% in one of the synthetic benchmarks. The reason for this was found to be the ASAP scheduling policy. The optimal solution delayed the scheduling of some tasks even when resources were available. The time taken by SSA (implemented in Java7) to compute a schedule was in the order of 10s of milliseconds. In contrast, the ILP-based scheduler took anywhere from a few seconds to a few days to complete.

The number of tasks in the synthetic graphs were randomly generated according to the distribution observed in the E3S suite. Since it is a suite of embedded benchmarks, the graph size is typically less than ten tasks. We performed experiments with synthetic graphs having hundreds of tasks, and found SSA's runtime to be in the order of seconds. However, we are unable to present any comparisons with the optimal scheduler in terms of schedule quality or scheduler running time because the ILP solver fails to converge on a solution in a reasonable amount of time.

Table VI: SSA penalty ($(\frac{PM_{SSA}}{PM_{secureILP}} - 1) \times 100$)

N_{HGC}	1	2	3	4	5	6
N_{3PC}	**E3S benchmarks**					
1	0.03%	0.0%	0.02%	0.02%	0.02%	0.02%
2	0.03%	0.66%	0.49%	0.39%	0.49%	0.49%
3	0.03%	0.0%	0.0%	0.0%	0.0%	0.0%
4	0.03%	0.85%	0.0%	0.0%	0.0%	0.0%
	Synthetic benchmarks					
1	0.13%	0.84%	0.36%	0.24%	0.36%	0.27%
2	0.0%	0.73%	0.5%	0.43%	0.39%	0.37%
3	0.2%	0.89%	0.12%	0.23%	0.21%	0.1%
4	0.14%	0.8%	0.35%	0.11%	0.12%	0.11%

Algorithm 6 attemptUpdate() : attempt verification strategy update

```
function ATTEMPTUPDATE(candidateTask,newVerStrat)
    oldPM ← compute PM of current schedule
    oldVerStratDirectives ← verStratDirectives
    verStratDirectives[candidateTask] ← newVerStrat
    assign()
    newPM ← compute PM of current schedule
    if newPM < oldPM then
        return True
    else
        verStratDirectives ← oldVerStratDirectives
        assign()
        return False
```

VII. Conclusion

The utilization of untrusted third-party hardware is inevitable. The step forward is to work towards designs where trusted home-grown hardware ensure that the untrusted hardware are operating correctly. This paper proposes *SecCheck* , a generic architecture that allows safe incorporation of third-party general purpose cores with a minimal loss in performance (10-17%). This paper also proposes the *SecCheck Scheduling Algorithm* that computes near-optimal (<1% overhead) offline application schedules for the *SecCheck* architecture, while being about 500X faster than an ILP solver.

References

[1] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, 2008.
[2] "Trusted foundry program," www.trustedfoundryprogram.org.
[3] J. R. Rilling, "Persistent monitoring of digital ics to verify hardware trust," Master's thesis, Iowa State University, 2011.
[4] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *Design Test, IEEE*, 2013.
[5] R. Kalayappan and S. R. Sarangi, "Secx: A framework for collecting runtime statistics for socs with multiple accelerators," in *ISVLSI*, 2015.
[6] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Micro*, 2015.
[7] M. Abramovici and P. Bradley, "Integrated circuit security: New threats and solutions," in *CSIIRW*, 2009.
[8] R. Kalayappan and S. R. Sarangi, "A survey of checker architectures," *ACM Comput. Surv.*, vol. 45, no. 4, Aug. 2013.
[9] C. Liu, J. Rajendran, C. Yang, and R. Karri, "Shielding heterogeneous mpsocs from untrustworthy 3pips through security-driven task scheduling," in *DFT*, 2013.
[10] J. Rajendran, H. Zhang, O. Sinanoglu, and R. Karri, "High-level synthesis for security and trust," in *IOLTS*, 2013.
[11] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Micro*, 1999.
[12] L.-W. Kim, J. Villasenor, and C. Koc, "A trojan-resistant system-on-chip bus architecture," in *MILCOM*, 2009.
[13] J. Dubeuf, D. Hly, and R. Karri, "Run-time detection of hardware trojans: The processor protection unit," in *ETS*, 2013.
[14] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *DAC*, 2013.
[15] R. P. Dick, "Embedded system synthesis benchmarks suites (e3s)," http://ziyang.eecs.umich.edu/~dickrp/e3s/.
[16] "Eembc, the embedded microprocessor benchmark consortium," www.eembc.org.
[17] R. Dick, D. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *CODES/CASHE*, 1998.
[18] L. A. Hall, D. B. Shmoys, and J. Wein, "Scheduling to minimize average completion time: Off-line and on-line algorithms," in *SODA*, 1996.
[19] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*, 2015.