Received 19 May 2013; revised 17 January 2014; accepted 11 July 2014. Date of publication 17 August 2014;
date of current version 4 February 2015.

Digital Object Identifier 10.1109/TETC.2014.2348182

Shielding Heterogeneous MPSoCs From Untrustworthy 3PIPs Through Security-Driven Task Scheduling

CHEN LIU[1], JEYAVIJAYAN RAJENDRAN[2], CHENGMO YANG[1], AND RAMESH KARRI[2]

[1]Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716 USA
[2]Department of Electrical and Computer Engineering, New York University Polytechnic School of Engineering, Brooklyn, NY 11201 USA
CORRESPONDING AUTHOR: C. LIU (liuchen@udel.edu)

ABSTRACT Multiprocessor system-on-chip (MPSoC) platforms face some of the most demanding security concerns, as they process, store, and communicate sensitive information using third-party intellectual property (3PIP) cores. The complexity of MPSoC makes it expensive and time consuming to fully analyze and test during the design stage. This has given rise to the trend of outsourcing design and fabrication of 3PIP components, that may not be trustworthy. To protect MPSoCs against malicious modifications, we impose a set of security-driven diversity constraints into the task scheduling step of the MPSoC design process, enabling the system to detect the presence of malicious modifications or to mute their effects during application execution. We pose the security-constrained MPSoC task scheduling as a multidimensional optimization problem, and propose a set of heuristics to ensure that the introduced security constraints can be fulfilled with a minimum impact on the other design goals such as performance and hardware. Experimental results show that without any extra cores, security constraints can be fulfilled within four vendors and 81% overhead in schedule length.

INDEX TERMS Security, hardware Trojan, heterogeneous MPSoCs, task scheduling, multi-dimensional optimization.

I. INTRODUCTION

HETEROGENEOUS Multiprocessor System-on-Chip (MPSoC) architectures have become a routine way of building embedded systems such as smart phones, network routers, storage and web servers, and gaming systems [1]. MPSoC designers typically integrate third-party intellectual property (3PIP) cores and outsource fabrication and testing steps. This allows designers to quickly respond to the increasing demands in functionality, power consumption and programmability without sacrificing design productivity [2].

Heterogeneous MPSoCs are vulnerable to malicious modifications (also known as *Hardware Trojan Horses*) in the 3PIPs and in the manufactured IC during fabrication. Trojans may cause system failures at some key point during application execution or could create backdoors to leak confidential information back to the attacker. Existing techniques use functional testing [3] and side-channel analysis [4], [5] to detect trojans inserted in a foundry. The complexity of MPSoCs makes it expensive and time consuming to fully test or analyze a system for the presence of trojans, since they are purposefully inserted in hard-to-detect sites in the design.

Since it is not possible to guarantee trustworthiness (i.e., 100% trojan free-ness) of 3PIPs, one needs to enable MPSoC to *detect* trojans or *mute* their effects during application execution. We propose security-driven MPSoC task scheduling to account for the untrustworthiness of the 3PIP cores. Comparing to existing trojan detection and prevention techniques [6]–[8], our main contribution is the incorporation of *diversity* into MPSoC task schedules. As multiple copies of the same 3PIP are instantiated in the target MPSoC, *diversity* is essential to reduce *false negatives*: on one hand it prevents two copies of a task from producing the same incorrect outputs, and on the other hand it isolates potential trojans, preventing them from sending triggers through undesired communication paths. We will describe diversity-based scheduling that enables the target MPSoC to (1) detect trojans that maliciously alter task outputs, and (2) mute trojan effects or prevent collusion between 3PIP cores from the same vendor.[1]

Incorporating security constraints into heterogeneous MPSoCs needs to accommodate the performance, power, cost, and design complexity constraints. We model the security-driven MPSoC task scheduling as a multi-dimensional optimization problem, wherein the level of security will be considered along with schedule length, communication overhead, and hardware cost. In our previous work [37], we built an ILP (integer linear programming) model for this problem. In this paper, we will outline task scheduling heuristics which, by exploiting the flexibility inherent in schedule generation, are able to minimize the performance, power, and hardware overhead.

The paper is organized as follows. Section II reviews the security challenges due to hardware trojans. Section III motivates the proposed technique. Section IV presents the security-driven scheduling constraints and heuristics. Section V experimentally verifies the technique, while Section VI summarizes the paper.

II. BACKGROUND AND RELATED WORKS

A. SECURITY CHALLENGES IN MPSoC DESIGN

Globalization of the IC design flow allows MPSoC designers to meet the tight time-to-market deadlines and to reduce the design, fabrication, and test costs. For example, Apple Inc. is a fabless integrator that purchases IP cores from third parties (e.g., ARM and Marvell), integrates these cores, generates the layout, and sends it to foundries (e.g., Samsung Foundries) for fabrication. However, outsourcing of the design steps is making MPSoCs prone to insider attacks.

From the perspective of their insertion method and payload, hardware trojans can be classified into two categories. A rogue insider in the foundry may make subtle mask changes, or alter chemical compositions to accelerate failures in critical circuitry [9]. On the other hand, a rogue insider in a third-party design house may insert malicious logic in an IP to modify functionality, deny service, or create a backdoor to leak confidential information back to the attacker [7], [9], [10].

From the perspective of their activation methods, hardware trojans can be classified as either *always-on* or *conditionally triggered* [11]. To avoid being detected during testing, an always-on trojan is inserted in rarely accessed places and its footprint is kept small [11]. An attacker may distribute trojans across multiple IPs in order to reduce the trojan footprint per module and hence reduce the probability of detection [11]. Conditionally triggered trojans hibernate initially, and are activated either by the trojan implanter or by on-chip triggers [7]. Similar to always-on trojans, the triggering conditions should be relatively hard to reach so as to prevent detection during testing.

[1] It is less possible for different IP vendors to collude since to do so, a rogue element has to expose itself to other rogue elements.

B. RELATED WORKS

1) DETECT TROJANS IN MANUFACTURED ICs

Many trojan detection techniques target malicious modifications during fabrication. These techniques are based on functional testing [3] and/or side-channel analysis [4], [5]. Path delays and power consumption can be characterized. These and similar side channels of manufactured chips can be measured and compared against the expected values to detect trojans [4], [5]. Side channel analysis is usually limited by the measurement capabilities of analog probes. For small trojans, the subtle differences in power and delay can be masked by process variations and measurement errors [12]. Non-destructive techniques that detect hardware trojans in the presence of process variations have been proposed [12]. These techniques combine algebraic, numerical, and statistical methods with power and delay measurements.

2) DETECT TROJANS IN 3PIP CORES

Detecting hardware trojans in 3PIPs is even more challenging because there is no golden (trojan-free) model for the designer to refer to. The work closest to that proposed here is an online trojan detection and prevention scheme for homogeneous systems [6]. It redundantly executes each program on three or more cores simultaneously, with their results verified by voting before being written to memory. This technique also partitions a program into segments that are executed by different sets of cores, aiming at limiting the data access capability of each core. Our technique not only requires less redundancy, but also explores vendor diversity to further protect task execution and communication.

A register transfer level (RTL) technique for designers to detect trojan attacks is proposed in [7]. Communications between different units are monitored to detect malicious behavior. This work is extended [8] to prevent trojans from being triggered by obfuscating and scrambling the inputs to infected units. These techniques require detailed RTL information of the procesesing cores and therefore their applications to 3PIP cores are limited.

Another work does trust verification of 3PIPs using code coverage analysis [13]. However, one cannot effectively detect non-trivial trojans in the absence of a golden model [14]. Another approach uses pre-defined agreements on security-related properties provided by a 3PIP vendor, so that MPSoC designers can check the 3PIP against these properties [15]. However developing security-related properties for a 3PIP is still in its infancy. Further, there may still be opportunities for a rogue designer to deliver malicious 3PIP cores that honor these security properties.

3) SECURITY ENHANCEMENTS IN MPSoCs

Previous work in MPSoC security targets software attacks such as buffer overflow, stack overflow, and software-based side-channel attacks. Architectural enhancements to MPSoCs have been proposed to detect software-based attacks through monitoring timing, control flow, and instruction execution counts at runtime [16], [17]. MPSoCs are protected against software-based side-channel attacks by creating a trusted execution environment that isolates the cores that execute critical tasks from the rest [18]. Support for customizing the security policies for different applications executing on an MPSoC has been proposed in [19].

III. THREAT MODEL AND TECHNICAL MOTIVATION

Our work questions the implicit assumption underlying MPSoC design, that is, *3PIPs procured from IP vendors are trustworthy*. There may be a rogue insider in a 3PIP house who may insert malicious logic in 3PIPs coming out of the IP house. The attacker may modify function, deny service, or create a backdoor to leak confidential information. The trojan may cause the task running on the malicious 3PIP either to produce incorrect output or to generate additional output to trigger trojans in another 3PIP core from the same vendor. We assume that the defender is in the design house, knows the task graph, has the flexibility of binding tasks to cores and choosing cores from different vendors, and designs glue logic to improve security.

Existing solutions aim to capture hardware trojans during testing. These offline techniques are limited not only by their measurement accuracy, but also by the lack of golden models [20]–[22]. In constrast, we adopt an *online* strategy to capture those attack patterns and the effects that a trojan may have. We exploit *vendor diversity*, an inherent in heterogeneous systems to eliminate false negatives and improve the accuracy of our solutions.

For trojans that modify functionality, it is highly possible that the outputs of the infected cores will be altered. These trojans can be *detected* by duplicating each task on two different cores and comparing the two outputs to capture any mismatch. *Vendor diversity* is essential to prevent two copies of a task from producing the same incorrect outputs. This is because trojans, unlike random faults, are inherent in 3PIPs. If multiple copies of the same 3PIP are instantiated in the MPSoC, each instantiation will contain the same trojan and hence produce the same incorrect output under the same input. To prevent such false negatives, each task needs to be executed by cores from different IP vendors.

To keep the trojan footprint small, an attacker may distribute trojans in multiple 3PIP cores and establish some secret communication channel between them to leak confidential information or, to trigger hibernating trojans. A concrete example is shown in Figure 1, wherein *Core*1, *Core*2, and *Core*4 share the same trigger value. During normal execution these cores will receive different inputs and will have different temperature and power values (also influenced by process variation) that diversify their triggering conditions. However, with the secret communication paths, if any core (e.g., *Core*1 in Figure 1) reaches the triggering condition, it can send messages to other cores by secretly writing the trigger value in a memory location (shown in red). As a result,

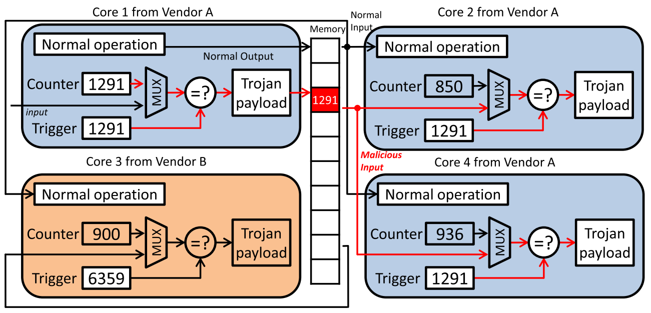


FIGURE 1. Example of a distributed trojan between cores from the same vendor. *Core*1 maliciously writes the trigger value to a memory location that is accessed by *Core*2 and *Core*4, triggering trojans in them. *Core*3 does not access this location, as it is produced by a different vendor.

all trojans of the same type become active within a short time period.

These distributed trojans may escape the duplication-based detection as they do not alter task outputs. They can, however, still be captured by monitoring the communication paths at runtime and invalidating suspect communications, if any. *Vendor diversity* is essential to eliminate false negatives, that is, malicious messages camouflaged as normal messages. Usually different vendors do not share a backdoor or a trigger pattern in common since to share this common element, a rogue element has to expose itself to the other rogue elements. This is shown in Figure 1, wherein the triggering condition of *Core*3 is different from the other three cores. Given this fact, collusion between malicious 3PIPs can be *prevented* if all the valid communication paths are confined to be between 3PIP cores produced by different vendors. This way, at runtime, even if a core (e.g., *Core*1 in Figure 1) silently produces unexpected data at runtime, such data are only accessible to cores from different vendors (e.g., *Core*3 in Figure 1) which will not use them for trojan triggering.

Vendor diversity incurs extra design cost since the designer needs to purchase licenses from multiple IP vendors. However, for applications with high security requirements (for example, banking and military systems), guarding the system has a higher priority than reducing the design cost. In some cases, *vendor diversity* may even reduce design cost. Previously a designer tends to purchase IPs from a company with a more established/good reputation, even though the price may be higher. Yet with *vendor diversity*, one can use multiple cheaper IPs without worrying about their individual security problems.

To summarize, proposed security enhancement adopts an online strategy to detect and prevent trojans, using *vendor diversity* inherent in heterogeneous MPSoC. Two security constraints are proposed to handle two major types of trojans: those tampering program outputs and those leaking information through undesired communication paths. These security constraints are incorporated into the MPSoC task scheduling step, as described next.

IV. SECURITY-DRIVEN TASK SCHEDULING

Task scheduling binds tasks to cores and coordinates data accesses, communication, and synchronization among the tasks [23]. This step determines the performance and power consumption characteristics of an application, as well as the types and number of the 3PIP cores needed in the MPSoC. Whereas, a traditional MPSoC scheduler explores a two-dimensional design space of performance (modeled as schedule length) and cost (modeled as types and total number of cores), we explore a third dimension – security.

In this section, we first define a set of security constraints, which are then embedded into the MPSoC task schedule. To satisfy these security constraints with minimum performance and hardware overhead, we develop heuristics that exploit the flexibility in task scheduling. The approach is adopted at the MPSoC design stage, under the assumption that the task graph information is available and the designer has the flexibility of binding tasks to cores, procuring cores from different vendors, and designing glue logic for security.

A. SECURITY-DRIVEN SCHEDULING CONSTRAINTS

As outlined in Section III, the goals of the proposed technique are to (1) detect trojans that affect task outputs, and (2) detect and prevent cores from passing undesired messages. These goals are achieved by imposing two security constraints.

1) DUPLICATION-WITH-DIVERSITY

To detect trojans, every task is redundantly executed on two 3PIP cores each from a different vendor. The same inputs will be sent to the task and its duplicate, while the outputs of both copies are compared by a trusted component (not designed by the third party) to ensure the trustworthiness of the comparison step. Techniques to compare results from different cores have been developed in [24] and [25]. Such techniques are complementary and applicable to our technique.

The proposed comparison is at a relatively coarse granularity: instead of performing cycle-by-cycle comparison of signals and instruction results, our technique compares the final task outcome. Coarse granularity offers a set of benefits. First it relaxes the synchronization constraint, allowing tasks to be executed on cores of different processing speeds and different instruction sets. This also minimizes the amount of data to be compared, enabling the use of low-cost comparators which can even be shared among multiple tasks. Moreover, even if the task produces a big chunk of output (e.g., an array), the comparison latency can be hidden by speculatively sending the output of a task to dependent task(s). Later, if the comparison fails, all the dependent tasks are terminated and a security flag is raised.

This constraint ensures the detection of any core that produces an incorrect output (because of a trojan), as long as attackers in two independent 3PIP design houses will not collude to develop identical trojans that produce identical incorrect outputs (which is highly unlikely). Note that this constraint is stricter than the straightforward duplication used for fault tolerance, which assumes that fault behavior is random and hence the occurrence of two identical faults in different cores is extremely low (unless these faults are induced by design errors). In contrast, as trojans are inherent in 3PIPs, multiple instantiations of the same 3PIP will contain the same trojan. Therefore, *diversity* is essential to prevent two copies of a task from producing the same incorrect outputs.

TABLE 1. Solution toward different Trojan classification.

	Tamper Output	Not Tamper Output
Create undesired communication	Isolation w/ Diversity & Duplication w/ Diversity	Isolation w/ Diversity
Not create undesired communication	Duplication w/ Diversity	Not critical, Left for future study

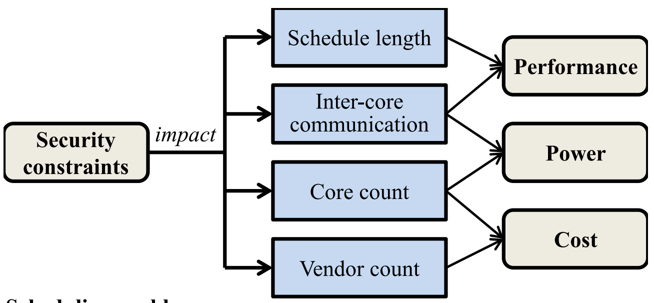
2) ISOLATION-WITH-DIVERSITY

This constraint prevents malicious messages from being passed between 3PIPs from the same vendor. In order to mute undesired and potentially malicious communication paths and at the same time isolate a trojan from the rest of the system, a task and its predecessor need to be scheduled on 3PIP cores from different vendors. This way, all the desired inter-core communications in the MPSoC will be between 3PIPs from different vendors.

The two security constraints together prevent the triggering of distributed trojans. The first constraint exposes incorrect task outputs, preventing them from providing incorrect information to any dependent core. The second constraint ensures that all the valid communication paths are between 3PIPs from different vendors. During application execution if one core silently produces unexpected data in addition to valid task outputs, the dependent cores, as they come from different vendors and hence are unlikely to collude with this core, will not use such data for trojan triggering. If two cores from the same vendor access the same data object[2] (i.e., have communication), a security flag will be raised indicating an invalid communication path.

To summarize, our schemes can successfully deal with the two categories of hardware trojans discussed in Section III. As illustrated in Table 1, the trojans tampering task outputs can be detected by *duplication with diversity*, while the trojans distributed among multiple IPs can be muted by *isolation with diversity*. A trojan will escape from our two schemes only if it neither tampers task outputs nor sends any message through an invalid communication path. We consider such trojans less critical.

[2]This can be detected by monitoring the communication channels. For shared-memory MPSoCs, an ownership vector (similar to the one used in directory-based cache coherency protocols [26]) can be used. For MPSoCs using on-chip networks, the routers can be augmented to check the types of cores sending and receiving the message. For bus-centralized MPSoCs, the bus controller can be configured to detect undesired communication paths.



Scheduling problem:

Inputs: 1) an upper bound of processing cores
2) speed of different cores (to model heterogeneity)
3) an application represented as a DAG
- Nodes = tasks, weight = execution time
- Edges = communications, weight = communication cost

Outputs: a start time and a core assignment for each task

Goals: 1) fulfill security constraints
2) minimize the performance, power, and cost at a priority specified by the designer

FIGURE 2. **Security-driven task scheduling—a multidimensional optimization.**

B. REFORMULATION OF MPSoC SCHEDULING

While vendor diversity is essential to the proposed security constraints, an increased level of diversity elevates the design cost. The two security constraints furthermore contradict the traditional scheduling goals shown in Figure 2 by requiring more cores, and creating more inter-core communications that may increase schedule length and energy consumption. To consider security along with the traditional scheduling goals, we reformulate the scheduling problem.

Given an application represented as a weighted *directed acyclic graph* (DAG), task scheduling can be formalized as the association of a start time and the assignment of a core with each node of the DAG. The inputs, outputs, and the goals of scheduling are shown in Figure 2. To incorporate security constraints, we make three modifications:

- Consider the number of vendors as an additional scheduling metric that impacts hardware and design cost.
- In line with existing scheduling algorithms for heterogeneous systems [28], [29], focus on modeling the performance differences created by heterogeneity, and assign different speeds to cores from different vendors.
- Fulfill security constraints while minimizing the associated overhead in hardware, performance, power, and cost, at a priority specified by the designer.

The security constraints can be incorporated into various scheduling algorithms [23], [30], [31]. To demonstrate the effectiveness and overhead of these constraints, we select a classic list scheduling algorithm as the baseline, and present two approaches for incorporating security constraints.

C. A STRAIGHTFORWARD SCHEDULING APPROACH

As security constraints define conflicts between tasks, they can be incorporated into the task graph. We first color the task graph to embed security constraints, and then schedule the colored graph onto the target MPSoC. To ensure security while maximizing performance, we decouple coloring from core speed assignment, enabling cores with higher speed to execute critical tasks. This approach consists of four steps, as shown in Figure 3(a):

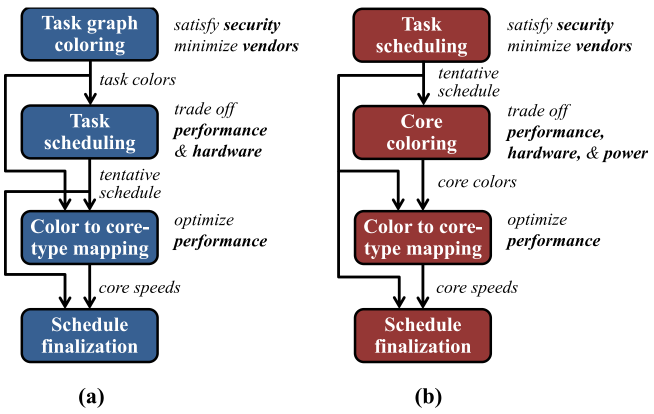


FIGURE 3. **Flows of (a) the straightforward scheduling approach and (b) the cluster-based approach. Each box represents one step, with its main optimization goals listed to the right. Arrows show data flow between steps, with the exact data type listed to the right.**

- *Task graph coloring* enforces security constraints and determines the number of vendors needed in the target MPSoC.
- *Color-constrained scheduling* determines the total number of cores, the core assignment of each task and its tentative start time.
- *Color to core-speed mapping* determines the exact type and speed of each 3PIP core.
- *Schedule finalization* determines the start time and finish time of each task based on core speed.

1) TASK GRAPH COLORING

Given the security constraints, determining the minimum number of vendors needed in the target MPSoC platform can be modeled as *graph coloring*.[3]

Given the original task graph, a *task conflict graph* can be constructed for coloring. Vertices in this graph represent tasks, while edges represent conflicts. The two security constraints can be imposed in this graph:

- To model the *duplication-with-diversity* constraint, the task graph is duplicated, and an edge is inserted between each task and its duplicate.

[3]Graph coloring (or vertex coloring) is the problem of finding the minimum number of colors and coloring the vertices of a graph such that no two adjacent vertices share the same color.

Algorithm 1 Schedule a Task in the Straightforward Scheme

```
Initialization:
 1: $V$ = task to be scheduled;
 2: $Corelist = \phi$;
Procedure:
 3: for all $core_i$ do
 4:     if ($core_i$.color = V.color) or ($core_i.color$ = blank) then
 5:         $Corelist \Leftarrow Corelist + \{core_i\}$;
 6:     end if
 7: end for
 8: for $core_i \in Corelist$ do
 9:     if $V.finishTime$ on $core_i$ is the earliest then
10:         Schedule $V$ on $core_i$;
11:         if $core_i.color$ = blank then
12:             $core_i.color \Leftarrow V.color$;
13:         end if
14:     end if
15: end for
```

- To model the *isolation-with-diversity* constraint, an edge is inserted between any pair of dependent tasks.

If the original task graph contains V nodes and E edges, the task conflict graph will have $2V$ nodes and $2E + V$ edges.

Standard graph coloring algorithms [32] can be applied to determine the minimum number of colors needed to color the graph, which is the minimum number of vendors needed in the MPSoC.

2) COLOR-CONSTRAINED TASK SCHEDULING

The key difference from standard list scheduling is that coloring constraints need to be fulfilled. Only tasks of the same color can be scheduled on the same core. This is achieved by assigning colors to cores during the scheduling in the following way: at the beginning, all the cores are colorless; at each scheduling step, a task is placed on colorless cores or cores of the same color as the task to find its early finish time; if the task is assigned to a colorless core, the core inherits this color of the task, and only accommodates tasks of the same color from then on.

Details of the color-constrained task scheduling algorithm are shown in Algorithm 1. Given the color of a task, the first loop (steps 3 to 7) identifies all the cores that a task V can be scheduled on (according to its coloring constraint) and puts them in *Corelist*. Then in the second loop (steps 8 to 15), all the cores in *Corelist* are iteratively checked to find the core that minimizes the finish time of V. The time complexity of Algorithm 1 for scheduling one task is $O(C)$. As a result, the complexity for scheduling the entire task graph is $O(VEC)$, with V representing the total number of tasks, E the total number of edges in the task graph, and C the total number of cores (for this and the remaining algorithms as well).

3) COLOR TO CORE-SPEED MAPPING

In the prior step, coloring constraints are embedded into the schedule and the color of each core is determined. Yet the exact speed of each core is not determined. Since tasks on critical paths have direct impact on schedule length, we develop a heuristic that assigns a higher speed to a core with more critical tasks in order to maximize schedule performance.

Algorithm 2 Core Criticality Ranking

```
 1: for all $core_i$ do
 2:     $metric_1(i), metric_2(i), metric_3(i) \Leftarrow 0$;
 3:     for $T_j \in$ all tasks scheduled on $core_i$ do
 4:         if $T_j$ is on critical path then
 5:             $metric_1(i) \Leftarrow metric_1(i) + 1$;
 6:             if $T_j.startTime < metric_2(i)$ then
 7:                 $metric_2(i) \Leftarrow T_j.startTime$;
 8:             end if
 9:             for all $T_k \in Child\{T_j\}$ do
10:                 if $T_k$ is on critical path then
11:                     $metric_3(i) \Leftarrow metric_3(i) + 1$;
12:                 end if
13:             end for
14:         end if
15:     end for
16: end for
17: for all $core_i$ and $core_j$ $(i \neq j)$ do
18:     if $metric_1(i) > metric_1(j)$ then
19:         $core_i.priority > core_j.priority$
20:     else if $metric_1(i) = metric_1(j)$ then
21:         if $metric_2(i) < metric_2(j)$ then
22:             $core_i.priority > core_j.priority$
23:         else if $metric_2(i) = metric_2(j)$ then
24:             if $metric_3(i) > metric_3(j)$ then
25:                 $core_i.priority > core_j.priority$
26:             end if
27:         end if
28:     end if
29: end for
```

The heuristic defines *critical tasks* as the tasks with 0 timing slack in the schedule, and *core criticality* as the number of critical tasks a core has. Critical tasks can be identified by computing the timing slack of each task (i.e., the difference between its latest and earliest start time) in the schedule obtained in the prior step. For cores with the same number of critical tasks, higher criticality is given to the one whose critical tasks either have more critical child tasks or are on the upper levels of the task graph. In this way, the benefit of scheduling critical tasks earlier can be maximized.

Ranking the cores according to their criticality is shown in Algorithm 2. It uses the following three metrics to measure criticality:

- $metric_1(i)$ is the number of critical tasks on $core_i$.
- $metric_2(i)$ is the start time of the earliest critical task on $core_i$.
- $metric_3(i)$ is the total number of *critical child tasks* of all the critical tasks on $core_i$. Here the *critical child tasks* are the child tasks that are on critical paths.

In Algorithm 2, the three metrics are first calculated (steps 1 to 16), and then the cores are ranked based on their criticality (steps 17 to 29). Higher priority is given to the core with higher values of $metric_1$ and $metric_3$ and lower values of $metric_2$. In this algorithm, the complexity of calculating the metrics is $O(VEC)$, and the complexity of ranking the cores is $O(C^2)$. Since the number of cores is smaller than

Algorithm 3 Color to Core-Speed Mapping

```
Initialization:
 1: $CoreList$ = {all cores}, sorted in descending order;
 2: $SpeedList$ = {all core-speed}, sorted in descending order;
Procedure:
 3: while $CoreList \neq \phi$ do
 4:     Find $C_{cri} \in CoreList$, the most critical core in $CoreList$;
 5:     Find $s \in SpeedList$, the fastest speed in $SpeedList$;
 6:     $C_{cri}.speed \Leftarrow s$;
 7:     $CoreList \Leftarrow CoreList - \{C_{cri}\}$;
 8:     $SpeedList \Leftarrow SpeedList - \{s\}$;
 9:     for $C_i \in CoreList$ do
10:         if $C_i.color = C_{cri}.color$ then
11:             $C_i.speed \Leftarrow C_{cri}.speed$;
12:             $CoreList \Leftarrow CoreList - \{C_i\}$;
13:         end if
14:     end for
15: end while
```

the number of tasks, the overall complexity of the algorithm is $O(VEC)$.

After determining the criticality of each core, the color to core-speed mapping can be performed iteratively as shown in Algorithm 3. During initialization, all the cores in *CoreList* are sorted based on the priority determined in Algorithm 2. At each iteration, among all the ''not-assigned-yet'' cores, the most critical one is selected from the pre-sorted list. This core, along with the ones that have the same color as it, is assigned the highest speed among all the ''available'' speeds. Then, that speed is marked as ''unavailable'' and all these cores are marked as ''assigned''. This process iterates until all the cores have been assigned a specific speed. The complexity of this algorithm is $O(SC)$, with C denoting the number of cores and S the number of core speeds (i.e., vendors).

4) SCHEDULE FINALIZATION

Since core speeds assigned in the prior step impact task execution time, it is necessary to adjust task start time (ST) and finish time (FT). The start time of task j is constrained by either the current available time of the core or the ready time of incoming data as described in Equation (1):

$$ST_j = max(\max_{1\leq i\leq p}(FT_i + CT_{ij}), FT_k) \tag{1}$$

Here task i is one of the p parent tasks of task j while task k is the last task scheduled on the same core before task j. CT_{ij} is the communication time from task i to j.

Meanwhile, the finish time of a task is the sum of its start time and its real execution time, determined by core speed:

$$FT_j = ST_j + \frac{exe_j}{CoreSpeed} \tag{2}$$

Equation (2) shows that the diversity in core speed may either accelerate or decelerate the execution of task j, while Equation (1) shows that this effect propagates, influencing all the descendants of j and all the tasks that are on the same core as j. Using these two equations, tasks are processed one-by-one in their scheduling order. The complexity of this process is $O(VE)$.

D. SCHEDULE QUALITY ENHANCEMENT

Straightforward scheduling fulfills the security constraints at the finest granularity: duplication-with-diversity constraints are added to each node in the task graph and isolation-with-diversity constraints are added to each edge in the task graph. One disadvantage of this approach is that all the communications are forced to be between 3PIP cores. This prohibits the scheduler from putting dependent tasks on the same core to hide communication latency and save energy.

To reduce the performance and energy overhead caused by the security constraints, we explore the possibility of grouping dependent tasks on critical paths into a cluster and scheduling the entire cluster to a single core. This will reduce the number of inter-core communications while satisfying the security constraints. Unlike the straightforward approach that schedules dependent tasks across different vendors, the cluster-based approach schedules dependent tasks either to the same core (for the intra-cluster cases) or across different vendors (for the inter-cluster cases).

Clustering of critical tasks necessitates information of task criticality. We develop a cluster-based scheme that first generates a performance-driven schedule and then colors the schedule to fulfill security constraints. As shown in Figure 3(b), the revised scheme includes four steps: task scheduling, core coloring, color to core-speed mapping, and schedule finalization. As the latter two steps are identical to the straightforward approach, we explain the first two steps next.

1) TASK SCHEDULING

This step generates a schedule and determines the total number of cores needed in the target MPSoC. To maximally explore the scheduler's ability in grouping critical tasks on a single core, we impose a loose constraint in this step to preclude false negatives during trojan detection, that is, a task v and its duplicate v' are scheduled on the same core.

2) CORE COLORING

This step embeds security constraints into the schedule, determining the exact color of each core and the number of vendors needed in the target MPSoC.

Since tasks are grouped into clusters, security constraints are imposed between cores instead of tasks. We construct a *core conflict graph* based on the schedule generated in the prior step. Each vertex in this graph is a core and each edge represents a conflict. An edge is inserted between cores i and j in two cases: (1) there exists one or more pairs of duplicated tasks on i and j (to satisfy *duplication-with-diversity*); (2) there exists one or more communication paths between i and j (to satisfy *isolation-with-diversity*).

Standard graph coloring algorithms [32] are first used to color the core conflict graph, and then each task inherits the color of the core it is scheduled on.

Algorithm 4 Schedule a Task in the Cluster-Based Scheme

```
Initialization:
 1: $v$ = task to be scheduled;
 2: $v'$ = duplicate of $v$;
 3: $N$ = current max-clique size;
 4: $CCG$ = current core conflict graph;
 5: $UpperBound$ = the given upper bound of clique size;
 6: $Corelist_A = \phi$;
 7: $Corelist_B = \phi$;
Procedure:
 8: for all $core_i$ do
 9:     if ($v'$ not on $core_i$) then
10:         if Putting $v$ on $core_i$ increases maximum clique size then
11:             $Corelist_A \Leftarrow Corelist_A + \{core_i\}$;
12:         else
13:             $Corelist_B \Leftarrow Corelist_B + \{core_i\}$;
14:         end if
15:     end if
16: end for
17: for all core $\in Corelist_A$ do
18:     Find $core_A$ that allows $v$ to have earliest start time $t_A$;
19: end for
20: for all core $\in Corelist_B$ do
21:     Find $core_B$ that allows $v$ to have earliest start time $t_B$;
22: end for
23: if ($N \geq UpperBound$ or $t_A > t_B$) then
24:     Schedule $v$ on $core_B$;
25: else
26:     schedule $v$ on $core_A$;
27:     $N \Leftarrow N + 1$;
28: end if
29: update $CCG$;
```

Algorithm 5 Estimate the Maximum Clique Size

```
Initialization:
 1: $v_m$ = The task to be scheduled;
 2: $core_i$ = Core that $v_m$ to be scheduled on;
 3: $V_{parent}$ = {all the parent tasks of $v_m$};
 4: $E_c$ = {all the edges in core conflict graph};
 5: $N$ = current max-clique size;
 6: $count = 0$;
Procedure:
 7: for all $v_n \in V_{parent}$ do
 8:     $core_j \Leftarrow$ the core that $v_n$ is scheduled on;
 9:     if $e_{ij} \notin E_c$ then
10:         $E_c \Leftarrow E_c + \{e_{ij}\}$;
11:         $core_i.degree{+}{+}$;
12:         $core_j.degree{+}{+}$;
13:         for all $core_k$ = common neighbor of $core_i$ and $core_j$ do
14:             $core_i.triangle{+}{+}$;
15:             $core_j.triangle{+}{+}$;
16:             $core_k.triangle{+}{+}$;
17:         end for
18:     end if
19: end for
20: for all $core$ do
21:     if ($core.degree \geq N$) and ($core.triangle \geq \frac{N(N-1)}{2}$) then
22:         $count{+}{+}$;
23:     end if
24: end for
25: if $count \geq N + 1$ then
26:     $N \Leftarrow N + 1$;
27: end if
```

E. HEURISTIC FOR VENDOR COUNT CONTROL

While the cluster-based approach is capable of placing critical tasks on a single core to boost performance, it may potentially increase the number of vendors needed in the MPSoC. This is because for the cluster-based approach, the number of vendors is bounded by the maximum clique size[4] of the core conflict graph. Unfortunately, maximum clique size is not considered during traditional performance-driven list scheduling. To solve this problem, we propose a heuristic to exploit the flexibility in prioritizing scheduling decisions. A traditional scheduler randomly picks a core assignment if a task has the same earliest start time on multiple cores. In contrast, the proposed scheme evaluates the different options based on their impact on the maximum clique size of the corresponding *core conflict graph* and selects the one with the smaller impact. In this way, the scheduler is able to minimize the number of vendors without degrading schedule performance.

Details of this clique-size-aware scheduling heuristic are shown in Algorithm 4. Step 9 is to identify the set of cores that satisfy the constraint of separating a task and its duplicate on different cores. Then, all the identified cores are classified into two lists, such that scheduling Task v on cores in $Corelist_A$ will increase the maximum clique size of the entire graph while scheduling v on cores in $Corelist_B$ will not (steps 10–14).

Given the two lists of cores, if the current maximum clique size already reaches the upperbound, the best core from $Corelist_B$ will be selected, ensuring that scheduling of v would not increase the maximum clique size. On the other hand, if the current maximum clique size is smaller than the upperbound, both $Corelist_A$ and $Corelist_B$ will be examined, and the core that allows v to have the earliest start time will be selected. This process is shown in steps 17–28. After determining the core assignment, the corresponding communication graph and maximum clique size will be updated if needed. Overall, Algorithm 4 has the same time complexity of $O(VEC)$ as Algorithm 1.

Computing the maximum clique size of a graph is NP-complete. However, as the scheduler processes one task at a time, an efficient heuristic can be developed. The observation is that scheduling a task on core i may add one or more edges[5] into the core conflict graph. As these edges share the same vertex i, they at most may increase the maximum clique size by 1. Based on this obervation, we develop a heuristic, shown in Algorithm 5. This algorithm computes a tight upper-bound of the maximum clique size based on the number of triangles in the conflict graph. It contains two major loops. The first loop (steps 7 to 19) iteratively updates two critical metrics, the *degree* (i.e., the number of edges)

[4]The maximum clique of a graph is its largest complete subgraph. A graph with a maximum clique size k needs at least k colors to color it.

[5]If no edge is added, the maximum clique size retains intact.

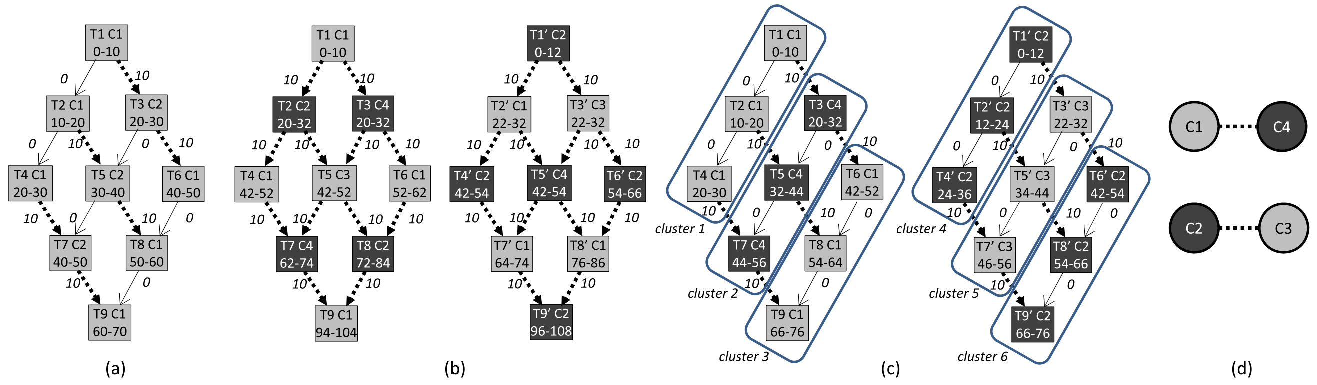


FIGURE 4. Scheduling Gaussian elimination (task graph). (a) Baseline schedule without security constraints, C1 and C2 have fastest speed. (b) Schedule generated by straightforward approach. It uses four cores from two vendors, with dark nodes 1.2× slower than light nodes. (c) Schedule generated by cluster-based approach. Six clusters are scheduled on four cores from two vendors. Scheduling a cluster on one core allows 30% speed up than (b). (d) Core conflict graph of (c). Cores connected with an edge have different colors.

of each node and the *number of triangles* that each node is involved in. During every iteration, upon inserting an edge e_{ij} into the core conflict graph, the degrees of $core_i$, $core_j$, and the triangle information of $core_i$, $core_j$, and their common neighbors are updated. Then, in the rest of the algorithm (steps 20 to 27), based on these updated metrics a tight upper-bound of the maximum clique size is computed using a heuristic introduced in [33]. This heuristic is based on the observation of two necessary conditions for a graph to have a maximum clique size of $N + 1$. First, there should be at least $N + 1$ nodes with a degree of N. Second, each of these $N + 1$ nodes should be involved in at least $N(N - 1)/2$ triangles. Same as Algorithms 1 and 4, the time complexity of Algorithm 5 is $O(EC)$ for each task and $O(VEC)$ for the entire task graph.

F. ILLUSTRATIVE EXAMPLE

A comparison between the two schemes shows that they have the same granularity in detecting trojans but different granularity in preventing trojans. To illustrate the differences between the two schedules, an example of applying them to the standard task graph of Gaussian Elimination is shown in Figure 4. Each node includes four values: *Ti* denoting its task ID *i*, *Cj* denoting that it is assigned to core *j*, and *m-n* denoting its start time *m* and finish time *n*. For simplicity, the original task execution time and the inter-core communication overhead are 10. The dark nodes are tasks assigned to the slower core, resulting in an execution time of 12. The solid arrows represent intra-core communication (with 0 overhead), while the bold, dashed arrows represent inter-core communication (with 10 overhead).

The baseline schedule in Figure 4(a) does not satisfy the security constraints. Tasks are not duplicated, and cores from the same vendor ($C1$ and $C2$) directly communicate. The schedule in Figure 4(b) satisfies both security constraints at the task level. Every task is duplicated on two distinct cores, and every communication is between distinct cores, resulting in a 50% longer schedule than Figure 4(a). The schedule in Figure 4(c) is obtained by duplicating the performance-driven schedule in Figure 4(a) and then coloring it to fulfill security constraints. The schedule contains 6 clusters, and 50% of all the communications are intra-cluster, resulting in a 30% improvement in schedule length over Figure 4(b).

V. EXPERIMENTAL RESULTS

A. METHODOLOGY

We select a standard list scheduling algorithm as the baseline [34]. Three approaches – baseline, straightforward, and cluster-based – are implemented with C language.

TABLE 2. Configurations of random generated task graphs.

	High-comm. Task Graphs	Low-comm. Task Graphs
Number of tasks	50, 100, 150	50, 100, 150
Number of start nodes	1-10	1-10
Task input/output degree	4/4, 8/8	4/4, 8/8
Average of comp/comm overhead	50/50	50/5
Variation of comp/comm overhead	20/20	20/2

The test set is composed of both standard parallel task graphs and random task graphs. Standard task graphs include *fork-join*, *LU decomposition*, *Laplace equation solver*, *Gaussian elimination*, and *FFT* [35]. We use TGFF [36] to generate 100 random task graphs. Table 2 reports the configurations for critical parameters in TGFF. High-communication and low-communication task graphs are generated by adjusting computation and communication overhead parameters.

TABLE 3. Security-induced schedule length overhead.

	Ration of Schedule Length Increase Δ_{SL}		
	Duplication only	Straight -forward	Cluster -based
Standard	0.371	0.653	0.500
Random, low-comm, 3-vendor	0.738	1.095	0.956
Random, low-comm, 4-vendor	0.726	1.196	1.079
Random, high-comm, 3-vendor	0.497	0.979	0.667
Random, high-comm, 4-vendor	0.573	1.205	0.857
Average	0.581	1.026	0.812

We assumed that the underlying MPSoC platform can accommodate up to 16 cores. Typically the cores produced by different vendors exhibit variations in many parameters (e.g., speed, area, and power consumption). However, since performance is the main concern of scheduling, we focus on speed variations across 3PIP cores, in line with most scheduling approaches of heterogeneous systems [28], [29]. We set the step of speed differences equal to 10% of the fastest core speed in our experiments. The core speed only affects task execution time, while the inter-core communication overhead remains intact.

B. RESULTS

1) NUMBER OF VENDORS NEEDED

Our first set of experiments explores the minimum number of vendors required for incorporating the proposed security constraints. As this impacts both the design cost and the hardware cost, our goal is to control this factor within a reasonable range.

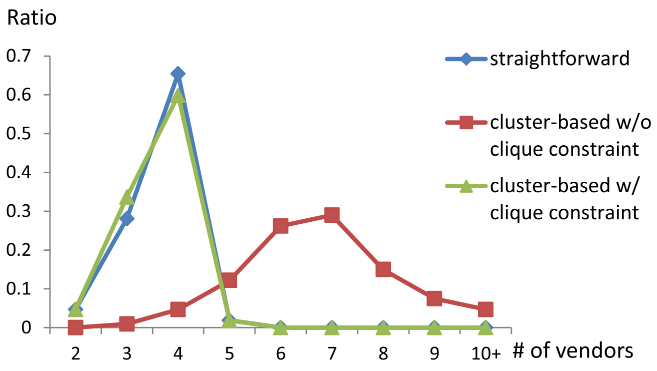


FIGURE 5. Ratio distribution of minimum number of vendors required to fulfill the security constraints

Figure 5 shows the ratio distribution of the minimum number of vendors required for all the task graphs. Data of both approaches are reported, while the cluster-based approach is scheduled with and without optimizing the maximum clique size. The ratio is calculated using the following formula:

$$ratio_i = \frac{\#\ of\ task\ graphs\ requiring\ i\ vendors}{total\ \#\ of\ task\ graphs} \tag{3}$$

The results show that for straightforward scheduling, 4 vendors are sufficient for most of the tested task graphs (105 out of the 107 graphs, including 7 standard and 100 random). For the cluster-based approach, more vendors are required when the maximum clique size is not restrained during scheduling. However, when Algorithms 4 and 5 are applied, the cluster-based approach has almost the same distribution as the straightforward approach, with 2 task graphs performed even better. This clearly confirms the effectiveness of the proposed clique size minimization heuristic, which is adopted consistently in the rest of our study. Overall, we believe that this extra cost for vendors is acceptable for building a trustworthy MPSoC, especially when it is used in critical infrastructure.

2) LENGTHS OF TWO SECURITY-DRIVEN SCHEDULES

We evaluate the performance of the straightforward and the cluster-based schemes by comparing their schedule lengths to the baseline list scheduling scheme [34]. To ensure fairness in comparison, the three schemes use the same number of cores (the one that allows the baseline scheme to deliver best performance), and the two security-driven schemes use the same number of vendors. For the two security-driven schemes, their ratio of schedule length increase (Δ_{SL}) over the baseline is calculated using the following equation:

$$\Delta_{SL} = \frac{1}{n}\sum_{i=1}^{n}\frac{SL(i)}{SL_{base}(i)} - 1 \tag{4}$$

where n is the number of task graphs and $SL(i)$ stands for the schedule length of the ith task graph. The lower the Δ_{SL} is, the better the performance.

The results of schedule lengths are shown in Table 3. We tested a *duplication-only* scheme to reflect the impact of the first security constraint individually. Both the standard and the random task graphs are reported, with the random task graphs divided into four categories based on the number of vendors needed and the different communication overhead. With the same number of cores, duplicating every task increases the schedule length by 58%. When both security constraints are imposed, the cluster-based scheme always outperforms the straightforward scheme. The straightforward approach has 21%–40% degradation on top of duplication-only, while the cluster-based approach only has 9%–20%. This confirms that by grouping critical tasks together and scheduling them to a single core, inter-core communication latency can be largely hidden, and hence schedule length can be sizably reduced. This benefit is more notable for high-communication cases because the profit of hiding communication overhead is higher. Task graphs requiring 4 vendors have larger overhead in schedule length than those with 3 vendors. This is because slower cores need to be used for execution when the number of vendors increases.

3) COMMUNICATION BREAKDOWN

Finally, we evaluate the impact of the two security-driven scheduling schemes on inter-core communications. There are three types of communication paths: *intra-core*, *inter-core&intra-vendor*, and *inter-vendor*. The second type violates the *isolation-with-diversity* constraint, while a higher ratio of the third type implies that more communication paths satisfy this security constraint (while the *detection-with-diversity* constraint is always guaranteed). The *inter-vendor* ratio is obtained using the following formula:

$$inter\text{-}vendor = \frac{1}{n}\sum_{i=1}^{n}\frac{Comm_{inter-vendor}(i)}{Comm_{all}(i)} \tag{5}$$

with $Comm_{inter-vendor}(i)$ denoting the number of inter-vendor communication paths for task graph i and $Comm_{all}(i)$ denoting the total number of communication paths for i. The ratio of the other two types of communications can be calculated in the same way.

The results are shown in Table 4. For each scheme, the *intra-core*, *inter-core&intra-vendor*, and *inter-vendor* ratios

TABLE 4. **Communication ratio of the three schemes. For each scheme, the ratios of intra-core, inter-core&intra-vendor, and inter-vendor communications are listed from left to right.**

	Baseline			Straightforward			Cluster-based		
	Intra-core	Intra-vendor	Inter-vendor	Intra-core	Intra-vendor	Inter-vendor	Intra-core	Intra-vendor	Inter-vendor
Standard	0.395	0.605	0	0	0	1	0.354	0	0.646
Random, low-comm, 3-vendor	0.266	0.734	0	0	0	1	0.267	0	0.733
Random, low-comm, 4-vendor	0.236	0.764	0	0	0	1	0.240	0	0.760
Random, high-comm, 3-vendor	0.267	0.733	0	0	0	1	0.271	0	0.729
Random, high-comm, 4-vendor	0.258	0.742	0	0	0	1	0.253	0	0.747
Average	0.284	0.716	0	0	0	1	0.278	0	0.722

are listed from left to right. As expected, only the baseline has the second type of communication paths, implying that security is guaranteed for the two proposed schemes. In the straightforward scheme, all the communication paths are forced to be inter-vendor. For the cluster-based scheme, the ratio of inter-vendor communications is about at the same level as the ratio of inter-core&intra-vendor communications in the baseline scheme. This is because the cluster-based approach first generates a performance-driven schedule (in the same way as the baseline) and then imposes security constraints to protect these inter-core communication paths and hence results in a trustworthy MPSoC design.

TABLE 5. **Overhead comparsion of different schemes.**

	Scheme in [6]	Straight	Cluster
Performance overhead	230%	65.3%-121%	50%-108%
Hardware overhead	30 cores	16 cores	16 cores
Vendor overhead	0	1-3	1-3

C. COMPARISON WITH RELATED WORK

Table 5 summarizes the overheads of our schemes and the scheme in [6]. Both of our schemes are over 100% better than the scheme in [6] in performance. Also, regarding to the hardware overhead, our schemes use the same number of cores with the baseline as mentioned before. So there is no hardware overhead in our schemes. In comparison, the scheme in [6] needs to use 30 Leon processors whereas the baseline can execute the entire program with just one 1 Leon processor. Regarding the vendor count, our schemes require 1 to 3 extra vendors while the scheme in [6] is applied to homogeneous MPSoCs and therefore needs no extra vendors.

VI. CONCLUSIONS

We have presented a task scheduling approach to protect MPSoCs against malicious modifications in 3PIPs. We proposed two security constraints: the duplication-with-diversity constraint allows the detection of trojans that cause tasks to produce incorrect output, while the isolation-with-diversity constraint prevents collusion between 3PIPs from the same vendor. We proposed two approaches to incorporating the constraints into task scheduling, one at the task level and the other at the core level. The results show that using these scheduling heuristics, security constraints can be fulfilled within 4 vendors. By scheduling dependent tasks on the same core, the overhead due to the two security constraints can be reduced to 81% of schedule length without any extra core.

Future work will focus on the following directions: (1) constructing precise and detailed models of core processing speeds and power consumption, task execution time and variations, (2) extending the security constraints to protect MPSoCs against collusion even between different vendors, (3) reducing the hardware overhead by selectively protecting only security-sensitive tasks.

REFERENCES

[1] *International Technology Roadmap for Semiconductors*. [Online]. Available: http://www.itrs.net/, accessed Aug. 2014.

[2] W. Wolf, "The future of multiprocessor systems-on-chips," in *Proc. 41st Design Autom. Conf. (DAC)*, Jun. 2004, pp. 681–685.

[3] M. Banga and M. S. Hsiao, "A novel sustained vector technique for the detection of hardware Trojans," in *Proc. 22nd Int. Conf. VLSI Design*, Jan. 2009, pp. 327–332.

[4] S. Narasimhan *et al.*, "Multiple-parameter side-channel analysis: A non-invasive hardware Trojan detection approach," in *Proc. 3rd Int. Symp. Hardw.-Oriented Security Trust (HOST)*, Jun. 2010, pp. 13–18.

[5] F. Koushanfar and A. Mirhoseini, "A unified framework for multimodal submodular integrated circuits Trojan detection," *IEEE Trans. Inf. Forensics Security*, vol. 6, no. 1, pp. 162–174, Mar. 2011.

[6] M. Beaumont, B. Hopkins, and T. Newby, "SAFER PATH: Security architecture using fragmented execution and replication for protection against Trojaned hardware," in *Proc. Design Autom. Test Eur. (DATE)*, Mar. 2012, pp. 1000–1005.

[7] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *Proc. 31st Int. Symp. Security Privacy (SP)*, May 2010, pp. 173–188.

[8] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proc. 32nd Int. Symp. Security Privacy (SP)*, May 2011, pp. 49–63.

[9] S. Bhunia *et al.*, "Protection against hardware Trojan attacks: Towards a comprehensive solution," *IEEE Des. Test.*, vol. 30, no. 3, pp. 6–17, Jun. 2013.

[10] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building stealthy and malicious hardware," in *Proc. Int. Symp. Security Privacy (SP)*, May 2011, pp. 64–77.

[11] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Proc. IEEE Int. Workshop Hardw.-Oriented Security Trust (HOST)*, Jun. 2008, pp. 15–19.

[12] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan horse detection using gate-level characterization," in *Proc. 46th Design Autom. Conf. (DAC)*, Jul. 2009, pp. 688–693.

[13] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *Proc. IEEE Int. Symp. Hardw.-Oriented Security Trust (HOST)*, Jun. 2011, pp. 67–70.

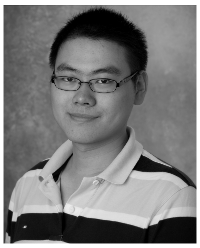
[14] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, vol. 74, no. 2, pp. 358–366, 1953.

[15] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 1, pp. 25–40, Feb. 2012.

[16] K. Patel, S. Parameswaran, and R. G. Ragel, "Architectural frameworks for security and reliability of MPSoCs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 9, pp. 1641–1654, Sep. 2011.

[17] K. Patel and S. Parameswaran, "SHIELD: A software hardware design methodology for security and reliability of MPSoCs," in *Proc. 45th Design Autom. Conf. (DAC)*, Aug. 2008, pp. 858–861.

[18] L. A. D. Bathen and N. D. Dutt, "TrustGeM: Dynamic trusted environment generation for chip-multiprocessors," in *Proc. 4th Int. Symp. Hardw.-Oriented Security Trust (HOST)*, Jun. 2011, pp. 47–50.
[19] L. A. D. Bathen and N. D. Dutt, "PoliMakE: A policy making engine for secure embedded software execution on chip-multiprocessors," in *Proc. 5th Workshop Embedded Syst. Security (WESS)*, 2010, pp. 1–10.
[20] M. Tehranipoor *et al.*, "Trustworthy hardware: Trojan detection and design-for-trust challenges," *IEEE Comput.*, vol. 44, no. 7, pp. 66–74, Jul. 2011.
[21] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware Trojans," *IEEE Comput.*, vol. 43, no. 10, pp. 39–46, Oct. 2010.
[22] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proc. ACM Conf. Comput. Commun. Security (CCS)*, Nov. 2013, pp. 697–708.
[23] H. Orsila, T. Kangas, and T. D. Hamalainen, "Hybrid algorithm for mapping static task graphs on multiprocessor SoCs," in *Proc. Int. Symp. Syst.-Chip*, Nov. 2005, pp. 146–150.
[24] D. Gizopoulos *et al.*, "Architectures for online error detection and recovery in multicore processors," in *Proc. Design Autom. Test Eur. (DATE)*, Apr. 2011, pp. 533–538.
[25] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, "Accelerating microprocessor silicon validation by exposing ISA diversity," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 386–397.
[26] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Mateo, CA, USA: Morgan Kaufmann, 1999.
[27] MIPS. *MIPS Licensees*. [Online]. Available: http://www.imgtec.com, accessed Aug. 2014.
[28] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, Feb. 1993.
[29] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proc. 8th Int. Heterogeneous Comput. Workshop (HCW)*, 1999, pp. 30–44.
[30] Y. Cho, S. Yoo, K. Choi, N. E. Zergainoh, and A. A. Jerraya, "Scheduler implementation in MP SoC design," in *Proc. 1st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2005, pp. 151–156.
[31] Y.-K. Kwok, I. Ahmad, and J. Gu, "FAST: A low-complexity algorithm for efficient scheduling of DAGs on parallel processors," in *Proc. 25th Int. Conf. Parallel Process. (ICPP)*, vol. 2. Aug. 1996, pp. 150–157.
[32] D. Brelaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979.
[33] K. Kim, "A method for computing upper bounds on the size of a maximum clique," *Commun. Korean Math. Soc.*, vol. 18, no. 4, pp. 745–754, 2003.
[34] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.
[35] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, Jun. 1996.
[36] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Workshop Hardw./Softw. Codesign*, Mar. 1998, pp. 97–101.
[37] C. Liu and C. Yang, "Exploiting heterogeneity in MPSoCs to prevent potential Trojan propagation across malicious IPs," in *Proc. 24th Great Lakes Symp. VLSI (GLSVLSI)*, May 2014, pp. 335–340.



CHEN LIU received the B.S. degree in communication engineering from Southeast University, Nanjing, China, in 2010. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, USA. His research interests include hardware security and computer architecture.

He was a recipient of the Best Student Paper Awards in the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems in 2013.

JEYAVIJAYAN RAJENDRAN is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Polytechnic Institute of New York University, New York, NY, USA. His research interests include hardware security and emerging technologies.

He was a recipient of the Best Student Paper Awards in the ACM Conference on Computer and Communications Security in 2013, the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems in 2013, and the IEEE International VLSI Design Conference in 2011.

He organizes the annual Embedded Systems Challenge, a hardware security competition. He was a co-organizer of the Army Research Office's Workshop on Trustworthy Hardware in 2013.

CHENGMO YANG received the B.S. degree in microelectronics from Peking University, Beijing, China, in 2003, and the M.S. and Ph.D. degrees in computer engineering from the University of California at San Diego, La Jolla, CA, USA, in 2005 and 2010, respectively. She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, USA. Her research interests include fault resilience, hardware support for system security, multi/many-core architectures, power- and thermal-aware systems, embedded system design, compile-time and run-time optimizations, scheduling and resource management, and nonvolatile memories.

She was a recipient of the University of Delaware Research Foundation Award and the National Science Foundation CAREER Award.

RAMESH KARRI is a Professor of Electrical and Computer Engineering at Polytechnic School of Engineering, New York University. He has a Ph.D. in Computer Science and Engineering, from the University of California at San Diego. His research interests include security and reliability.

He was the recipient of the Humboldt Fellowship and the National Science Foundation CAREER Award. He is the area director for cyber security of the NY State Center for Advanced Telecommunications Technologies at NYU-Poly; Hardware security lead of the Center for research in interdisciplinary studies in security and privacy -CRISSP (http://crissp.poly.edu/), co-founder of the Trust-Hub (http://trust-hub.org/) and organizes the annual red team blue team event at NYU, the Embedded Systems Challenge (http://www.poly.edu/csaw2014/csaw-embedded).

He cofounded the IEEE/ACM Symposium on Nanoscale Architectures (NANOARCH). He is the Program Chair (2012) and General Chair (2013) of IEEE Symposium on Hardware Oriented Security and Trust (HOST). He is the Program Co-Chair (2012) and General Co-Chair (2013) of IEEE Symposium on Defect and Fault Tolerant Nano VLSI Systems; General Chair of the 2009 and 2013 NANOARCH; General co-chair of ICCD 2015, RFIDSEC 2015 and WISEC 2015. He serves on several program committees.

He was the Associate Editor of IEEE Transactions on Information Forensics and Security (2010-2014), IEEE Transactions on CAD (2014-present), ACM Journal of Emerging Computing Technologies (2007-present) and ACM Transactions on Design Automation of Electronic Systems (2014-present).He is an IEEE Computer Society Distinguished Visitor (2013-present).