IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 22, 2017, accepted December 26, 2017, date of publication January 8, 2018, date of current version April 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2790392

Security-Aware Task Scheduling Using Untrusted Components in High-Level Synthesis

NAN WANG[1], SONG CHEN[2], (Member, IEEE) JIANMO NI[3], XIAOFENG LING[1], AND YU ZHU[1]

[1]School of Information Science and Engineering, East China University of Science and Technology, Shanghai 200237, China
[2]School of Information Science and Technology, University of Science and Technology of China, Hefei 230026, China
[3]Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093, USA

Corresponding author: Nan Wang (wangnan@ecust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61604054 and in part by the Fundamental Research Funds for the Central Universities under Grant 222201514332. The work of S. Chen was supported by the National Natural Science Foundation of China under Grant 61732020.

ABSTRACT The high penetration of third-party intellectual property is accompanied with severe security issues, and thus, security constraints during task scheduling have recently been proposed for protecting multiprocessor system-on-chip systems. However, these security constraints incur significant overheads in terms of the schedule length and design cost. In this paper, the multi-dimensional design optimization space (schedule length, design cost, area, and security) is explored, and two task scheduling approaches in the context of security constraints are proposed. In resource-constrained task scheduling approach, the maximum clique of a vendor violation graph is accurately calculated, enabling a minimized number of security constraint violations under the vendor constraint. In addition, task scheduling is conducted alongside vendor assignment to optimize the schedule length. In performance-constrained task scheduling approach, a max-flow min-cut-based task clustering method is first proposed to iteratively reduce the schedule length of the graph containing all critical paths. Then, vendor assignment is performed by solving a graph coloring problem, and all tasks are finally scheduled with an optimization of hardware resources. The experimental results demonstrate that our resource-constrained task scheduling approach reduces the schedule length by 28.2% with all security constraints satisfied; besides, 18.0% cores are saved by our performance-constrained task scheduling approach.

INDEX TERMS Hardware Trojan, 3PIP, security, task scheduling, high-level synthesis.

I. INTRODUCTION

Market scaling has increased the design productivity requirements for heterogeneous Multiprocessor System-on-Chip (MPSoC). Thus, MPSoC designers have started to utilize third-party intellectual property (3PIP) cores to meet the soaring demand of the global market. However, the high penetration of 3PIPs has brought new challenges for MPSoC hardware design, and security is one of the most important issues.

Indeed, the growing number of mission-critical applications (e.g., finance, military, and transportation) that use MPSoCs means that security is the highest priority issue. In these systems, MPSoCs are expected to be 100% trustworthy and their security-aware design is of great importance. In addition, the increasing integration of 3PIPs and the outsourcing of fabrication lead to the fact that heterogeneous MPSoCs have a high risk of malicious modification because the 3PIPs are not 100% trustworthy. Foundries may make subtle mask changes or insert disguised malicious modifications during fabrication, and this situation becomes more complicated when 3PIPs from the same vendor collude with each other. Thus, vendors may distribute Trojans on different 3PIP cores and activate them via secret communication paths only when these cores work together.

These emerging problems require that designers develop techniques for detecting possible hardware Trojan attacks or for muting their effects, and several areas of study inform the field of hardware trust [1]: 1) Trojan detection approaches, 2) design for trust approaches, and 3) runtime monitoring approaches. Trojans detection approaches typically attempt to detect the existence of Trojans at an IP level, and they can be further classified into follows: physical inspection [2] where an engineer must repeatedly scan the surface while grinding the layers of the chip after the molding coat is cut; functional

testing [3] by stimulating the input ports of a chip and monitoring the output to detect manufacturing faults; built-in tests [4]–[6] by adding circuitry to the chip to help verify that the as-built chip implements its functional specification; and side-channel analyses [7]–[13] of the signals generated by electric activity to obtain the information about the state of the device and the data it processes, thereby detecting tightly coupled Trojans based on these signals.

The high penetration of 3PIPs is accompanied with severe security issues because it is impossible to detect all of the hardware Trojans in 3PIPs, and thus, several *design for trust* approaches have been proposed in higher design abstraction levels. Beaumont *et al.* [14] developed an online Trojan detection architecture that implements fragmentation, replication, and voting. Rajendran *et al.* [15] focused on the Electronic System Level (ESL) design tools and added protection against attacks at the ESL to make it more robust. Jiang *et al.* [16] proposed a novel secure embedded systems design framework for efficiently optimizing the runtime quality with security constraints. Cui *et al.* [17] implemented both Trojan detection and recovery at run-time, which are essential for mission-critical applications.

Recent works have also proposed using multiple copies of the same IP from different vendors to compare using unrolling [18] and validation [19] techniques, and they incorporate security constraints in high level synthesis to identify malicious inclusions. Building on the preliminary version of security constraints, a set of researchers have proposed the design-for-trust techniques for MPSoCs [20], [21], and another set of researchers reduce the power/area/delay overhead of the technique [22].

However, fulfilling the security constraints always incurs a significant overheads in terms of design cost, area and performance delay, and we focus on how to balance these practical factors with security-aware design in this study. Security constraints are integrated into the task scheduling process in a more practical manner by considering design cost (modeled as the numbers of IP vendors required), area constraints (modeled as the number of cores available, which is also named as resource constraints) and performance constraints (modeled as schedule length). The main contributions of this study can be summarized as follows.

- In contrast to a previously proposed method [21] which conducts task scheduling and vendor assignment in different steps, we integrate task scheduling alongside vendor assignment. This reduces the schedule length by 28.2% with all security constraints satisfied.
- The maximum clique of a vendor violation graph, which equals the number of IP vendors required, is accurately calculated with a small computational cost.
- This proposed MPSoC scheduler explores multi- dimensional design optimization space (performance, design cost, area and security), and the number of security constraint violations is minimized. Thus, the proposed algorithms could be widely used.
- A max-flow min-cut-based algorithm is proposed for a performance-security trade-off by assigning adjacent timing-critical tasks to the same core, with a minimum number of security constraint violations.

The remainder of this paper is organized as follows. Section 2 discusses the threat model and presents the security architecture. Section 3 introduces the motivation and describes the problem optimization targets. Sections 4 and 5 describe two task scheduling approaches with resource constraints and performance constraints, respectively. Section 6 gives the experimental results and we conclude this paper in Section 7.

II. THREAT MODEL AND SECURITY ARCHITECTURE

At present, the industry needs to procure and use the latest Commercial-Off-The-Shelf (COTS) 3PIPs in order to track the most cutting edge technology while reducing the design and manufacturing costs. However, the hardware Trojans in 3PIPs components present high risks of malicious inclusions and data leakage in products [23].

A. TROJAN ATTACKS THROUGH HARDWARE IPs

In General, the Register Transfer-Level (RTL) files of IPs might have been imported from third party vendors, and 3PIPs procured from IP vendors are usually not 100% trustworthy. There may be a rogue insider in a 3PIP house who may insert Trojan logic in 3PIPs coming out of the IP house, and the Trojans may modify function, deny service, or create a backdoor to leak confidential information. The possibility of Trojan attacks in 3PIP poses a major integrity concern to SoC designers.

Verification of trust of an IP acquired from untrusted third-party sources can be extremely challenging due to lack of golden models. Conventional verification approaches, which rely on the presence of a golden or reference design, do not work in case of 3PIPs. Functional simulation or emulation does not provide adequate coverage due to often incomplete functional specifications, and it cannot provide assurance against additional functionality due to a Trojan. For example, if a processor IP core triggers a malicious memory write for an ''add'' instruction with a specific rare combination of operands, both simulation and emulation are very likely to fail because they may not excite all rare conditions [24].

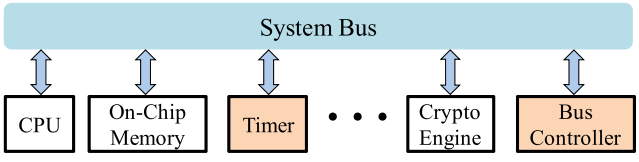


FIGURE 1. Wireless video capture SoC.

An example of Trojan attack in a wireless video capture System-on-Chip is shown in Fig. 1 [20]. The timer and bus controller (BC) are obtained from a 3PIP vendor and maybe infected with Trojans. During the normal operation, the BC

controls the system bus when the timer expires, and the timer sends a packet to the BC through the system bus to inform the expiration of time. During the malicious operation, the timer embeds a trigger in its packet, and this trigger activates the Trojan in the BC and puts the system bus in a tristate. All IPs connected in the system bus cannot communicated with each other, resulting in a denial-of-service attack.

B. TROJAN DETECTION IN HARDWARE IPs

Today's SoC designs use a large number of IP cores from different IP vendors, with varying degree of reliability associated with each vendor. Trojans may be inserted into IPs of different forms by a rogue designer or an untrusted CAD tool in an IP design house. Most of the existing solutions for trusted IP acquisition fall into the following classes.

1) CODE ANALYSIS

Detection all of the hardware Trojans in 3PIPs is extremely difficult since there is no known golden model for 3PIPs as IP vendors usually provide specification and source code, both of which may contain Trojans, and when a Trojan exists in an IP core, all fabricated ICs including these IPs will contain Trojans. A Trojan can be very well hidden during the normal functional operation of the 3PIP supplied as RTL code. An attacker may distribute few RTL codes so as to reduce Trojan footprint, and a large industrial-strength IP core can include thousands of lines of code, resulting in identifying the few lines of RTL code in an IP core that represent a Trojan to be an extremely challenging task.

The code coverage analysis on RTL codes may identify suspicious signals that may be a part of a Trojan [25], [26]. However, even 100% coverage of the RTL code does not guarantee that it is fault free [27]. Hence, the code coverage analysis does not guarantee its trustworthiness [20].

2) FORMAL VERIFICATION

An SoC integrator and a 3PIP vendor can agree upon predefined security properties that the IP should satisfy [28], and the SoC integrator can check the 3PIP for the properties. To check if a 3PIP honors these properties, the target 3PIP is converted into a proof checking format (e.g., Coq). This has been demonstrated to detect data leakage [5] and malicious modifications to registers [29]. In addition, conventional verification techniques are also used to detect data leakage [30] and modifications to registers [6].

However, formal verification also has several limitations, which are: 1) one cannot check if the 3PIP has vulnerabilities while satisfying the agreed-upon properties [5]; 2) the absence of automation tools that convert VHDL/Verilog into Coq format; 3) a Coq representation is considered trustworthy does not necessarily mean that the corresponding VHDL/Verilog representation is trustworthy [20].

3) DESIGN-FOR-TRUST TECHNIQUES

The SoC integrator modifies the target design to mute the effects of Trojans or detect them, and this proposed work falls under this category. Recently, researchers begin to use an integer linear programming formulation to model the constraints to detect run-time errors and recover from infected 3PIPs [17]. Building on the preliminary version of security constraints in [19], a set of researchers have proposed the design-for-trust techniques for MPSoCs [20], [21], and reduce the power/area/delay overhead of the technique [22].

C. SECURITY-DRIVEN CONSTRAINTS

An application is always decomposed into a set of computational entities, called *tasks*, and these tasks are linked by *precedence constraints*. Task scheduling schedules tasks to cores and coordinates data accesses, communication, and synchronization among tasks [21]. The number of cores required in the MPSoC as well as the schedule length is determined in this step.

The **task graph** is denoted by TG, where $TG = \{V, E\}$; $V = \{v_1, v_2, ...\}$ represents the set of tasks, and E is the set of edges representing the data dependencies between every pair of tasks. An example of TG is shown in Fig. 3(a).

One of the most common task scheduling methods to guard the MPSoC system has been recently proposed in [21], in which IPs are purchased from different vendors without worrying about their individual security problems. All tasks are then scheduled and bound to the cores under the following two kinds of security constraints: *task duplication* and *vendor diversity*, and these security constraints handle two major types of Trojans: those tampering program outputs and those leaking information through undesired communication paths [21].

1) TASK DUPLICATION

Trojaned hardware may produce incorrect outputs, which might cause the underlying systems to fail. To mute the attack footprint, the attacker only needs to tamper with the outputs of several tasks in complex systems that always comprise hundreds of tasks in practice. Thus, although a core can execute multiple tasks, its hibernating Trojan is triggered by a specific input. Diversifying the sources of a 3PIP will help detect the wrong outputs caused by Trojans, because 3PIPs from different vendors will have different implementations even for identical functionality [20].

Task duplication constraint duplicately executes each task on the cores from different vendors, and the outputs of these cores may vary and they will be compared by a trusted component (not designed by the third party) to ensure the trustworthiness of the comparison step. The methods to compare results from different cores have been demonstrated by Gizopoulos [31], which are applicable to our work. This security architecture compares the final task outcome instead of performing cycle-by-cycle comparison of signals and instruction results. If the comparison fails, all dependent tasks are terminated and a security flag is raised. In the following discussion, the duplicate task of v_i is denoted as v_i', and the *duplicated task graph* is denoted as TG'.

2) VENDOR DIVERSITY

To mute the Trojan footprint, the attacker always distributes Trojans in multiple IP cores and constructs secret communications between IP cores to leak information, or to trigger the hibernating Trojans [21]. In this study, we assume that the secret communication between IP cores from the same vendor cannot be acquired by other vendors and that the attackers of different vendors plant different hardware Trojans.

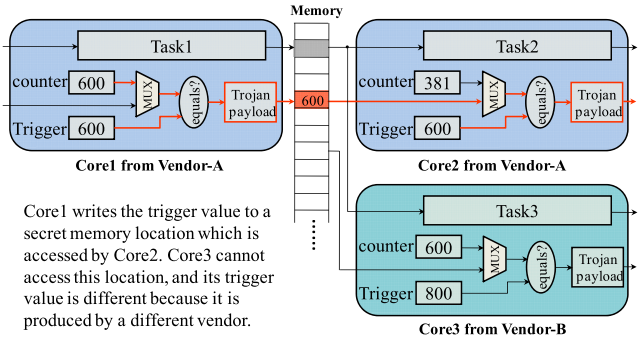


FIGURE 2. Example of Trojan triggering between cores from the same vendor.

An example of triggering Trojans between cores is illustrated in Fig. 2, where *Task1* is the parent of *Task2* and *Task3*, *Core1* and *Core2* are from the same vendor, and they share the same triggering value. If *Core1* reaches the triggering condition, it writes the triggering value to a secret memory location that is accessed by *Core2*, and thus, the Trojans implanted in *Core2* will be activated. However, *Core3* is unable to fetch the triggering value, and its triggering condition is also different because it is produced by a different vendor. This example demonstrates that executing adjacent tasks by the cores from different IP vendors cuts the secret communications between cores and isolates the Trojan from the rest of the system.

The *redundant execution* approaches, including voting architecture [14], dual modular redundancy (DMR) [31], [32], and task duplication constraint, detect the hardware Trojans by comparing the outputs of cores from different IP vendors with the same input; however, they cannot cut off these secret communications. Therefore, the vendor diversity constraint, which forces the adjacent tasks to be executed by the cores from different IP vendors, is also introduced to isolate the triggered hardware Trojans from the rest of the system.

III. MOTIVATION AND PROBLEM DESCRIPTION

A. MOTIVATION

1) MOTIVATION1: SCHEDULE LENGTH OPTIMIZATION

With all security constraints satisfied, Liu *et al.* [21] perform task scheduling and vendor assignment in different stages. However, the number of cores available from each vendor is ignored during scheduling, and this might cause additional delay to the schedule length because several data-dependency-free tasks can no longer be executed parallel due to the limited number of cores. Example of task scheduling results generated by Liu *et al.* [21] is presented in Fig. 3(b), with the input task graph given in Fig. 3(a). The solid lines and dashed lines represent inter-core and intra-core communications, respectively, and the communication delay is marked next to the edge. Each node has four values: V_i is the $i-$th task and its duplicated task is V_i'; C_j denotes the assigned core j; $p-q$ are the start and finish times. The computational cost of each task is the same.

In Fig. 3(b), tasks V_3 and V_4 are assigned to *vendor*3, and only one core $C5$ is available to compute these two tasks. Thus, V_3 and V_4 cannot be computed parallel, making the schedule length to be 100 $u.t.$. However, Fig. 3(c) show a task scheduling result with proper vendor assignment, and its schedule length is only 90 $u.t.$. To assign each task with a proper vendor so as to optimize the schedule length, we will conduct task scheduling alongside vendor assignment. This is because we decide the vendor assignment of a task only when we start to schedule this task, and the core assignments together with the scheduling results of its ancestors can be counted on, which helps to improve the final scheduling result.

2) MOTIVATION2: SECURITY CONSTRAINT VIOLATION MINIMIZATION

Fulfills the security constraints at the finest granularity, but this incurs significant overheads in terms of design cost, area and system performance. Therefore, researchers also explore the possibility of grouping dependent tasks into a cluster and scheduling the entire cluster to a single core [21] to hide the inter-core communication latency. This reduces schedule length and area of the MPSoC system, but violates the security constraints. In this study, the edge that connects two clustered tasks is called *contracted edge*.

Suppose a task graph contains n nodes and m edges, and the number of all security constraints (denoted as scy) is $n+2m$: The number of task duplication constraints is n, and the numbers of vendor diversity constraints in TG and its duplicated TG' are both m. However, several security constraints may be violated when optimizing the schedule length and the number of IP vendors required, and the two types of ***security constraint violations*** are: 1) task duplication violation, where a task and its duplicated task are conducted by the cores from the same vendor; 2) vendor diversity violation, where a task and its adjacent tasks are assigned to the cores from the same vendor. The number of security constraint violations is denoted as scy_v, and a large number of security constraint violations always lead a high hardware Trojan trigging risk.

Liu *et al.* [21] maximally explore the scheduler's ability in clustering timing-critical tasks on a single core to minimize the schedule length, however, they ignore the number of security constraint violations during task clustering. Suppose the task graph is given in Fig. 3(d), and the target is to optimize the schedule length to 60 $u.t.$. The cluster-based scheduling result of Liu *et al.* [21] is presented in Fig. 3(e), whose scy_v is 6. However, the best task clustering and scheduling results are presented in Fig. 3(f), and the resulting scy_v is only 4, meaning a smaller Trojan triggering risk.

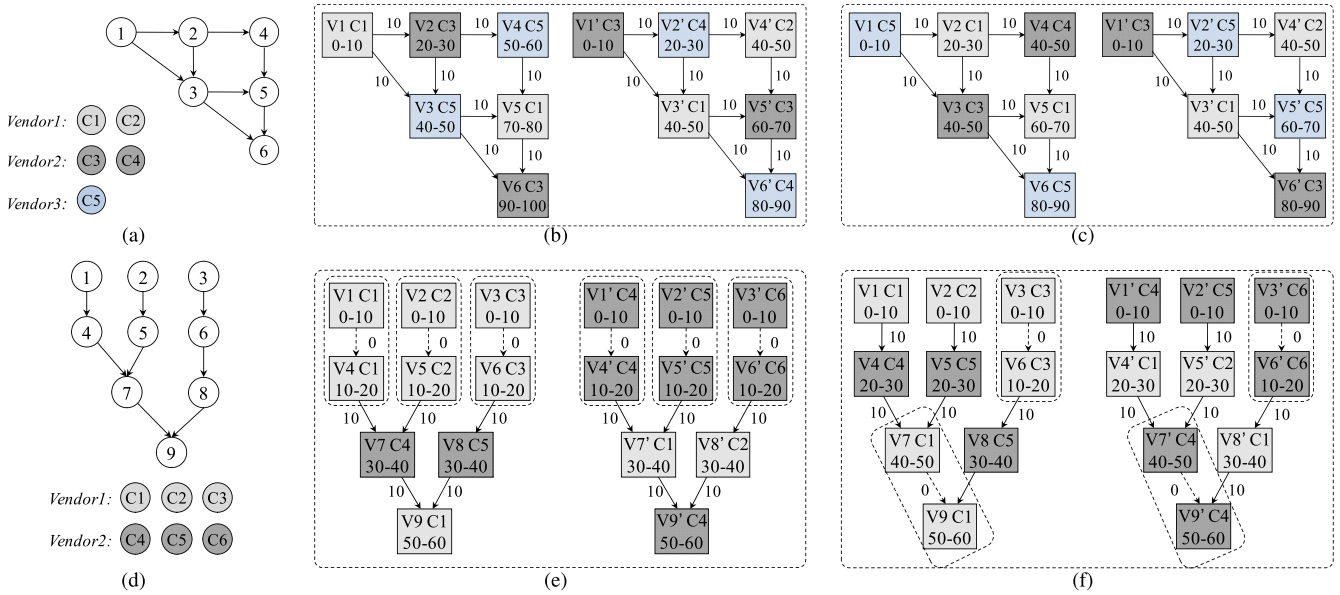


FIGURE 3. Examples that demonstrate our motivations. (a) 1st task graph with resource constraints. (b) Schedule of the 1st task graph with security constraints. (c) Schedule of the 1st task graph with schedule length optimized. (d) 2nd task graph with resource constraints. (e) Cluster-based task scheduling result of the 2nd task graph. (f) Schedule result of the 2nd task graph with a minimum number of security constraint violations.

To sum up, the following three aspects are ignored by most of the existed works.

1) The number of security constraint violations must be minimized when optimizing the schedule length and the number of IP vendors.
2) Rather than maximally minimizing the schedule length with a large number of security constraint violations, designers may prefer to make trade-off between security and performance.
3) The number of cores also needs to be optimized, because it is closely related to the circuit area and the power consumption.

B. PROBLEM DESCRIPTION

In this study, two security-aware task scheduling problems are studied, with design cost, area and system performance constrained. The design cost is modeled as the number of IP vendors, the area constraint (also named as resource constraint) is modeled as the number of cores, and the performance constraint is modeled as the maximum delay that the schedule length must not exceed.

The first optimization problem in this study is named as the security-aware resource-constrained task scheduling problem, which is described as follows.

Problem 1: Inputs: task graph *TG*, vendor constraints, and resource constraints. The target is to find a schedule that would minimize the number of security constraint violations; in addition, schedule length is minimized.

The objective function of *Problem 1* is formulated as follows.

$$min:\ \alpha_1 * scy_v + SL \tag{1}$$

where scy_v is the number of security constraint violations, and SL is the schedule length. α_1 is a constant large enough to keep the minimization of scy_v as the first priority.

Tasks are clustered to satisfy the vendor constraints, and if the vendor constraint is not given in *Problem 1*, this problem becomes a schedule length optimization problem with all security constraints satisfied.

The second optimization problem concerned in this study is the security-aware performance-constrained task scheduling problem, which is described as follows.

Problem 2: Inputs: task graph *TG*, vendor constraints, and performance constraints. The target is to find a schedule that would minimize the number of security constraint violations; in addition, the number of resources is minimized.

The objective function of *Problem 2* is given as follows.

$$min:\ \alpha_2 * scy_v + core_{req} \tag{2}$$

where $core_{req}$ is the number of cores required by the scheduling result to execute both *TG* and its duplicate *TG*′, and α_2 is a constant large enough to ensure that the minimization of scy_v is the first priority.

IV. SECURITY-AWARE RESOURCE-CONSTRAINED TASK SCHEDULING

The security constraints [20], [21] protect systems from potential attacks in the 3PIP cores, but they also incur overheads in terms of design cost, area and schedule length. Therefore, a resource-constrained task scheduling approach is proposed in this section, to solve the problem described in ***Problem 1***. Fig. 4 gives the flow of this approach, which consists of the following three steps.

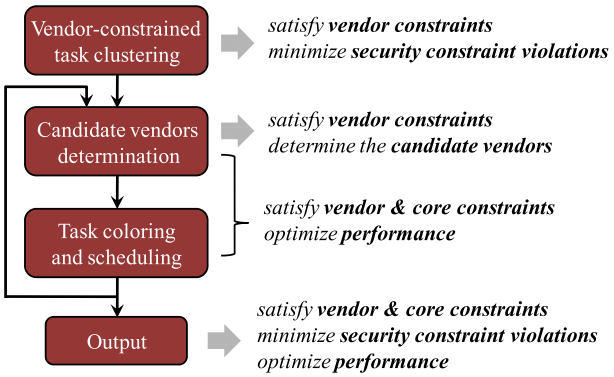


FIGURE 4. Flow of the resource-constrained task scheduling approach.

- *Vendor-constrained task clustering* enforces the vendor constraints, with a minimum number of security constraint violations.
- *Candidate vendors determination* figures out a set of vendors that can be assigned to each task during scheduling, and the vendor constraints are strictly satisfied. This process consists of *candidate color set determination* and *clique size updating*.
- *Task coloring and scheduling* determines the vendor and core assignments of a task, and simultaneously schedules this task to its best period.

A. VENDOR-CONSTRAINED TASK CLUSTERING

Definition 1 (Clique Size): $\omega(G)$ is the number of nodes in a *maximum clique* of G, and it is called the *clique size* of G.

The **vendor violation graph** (*VVG*) is constructed from *TG*: $VVG = (V_V, E_V)$, where $V_V = \{c_1, c_2, ...\}$ is the set of clusters. An edge in E_V means that the two connected clusters must be assigned to different vendors.

The index of a cluster is decided by the minimum index of the tasks in this cluster; e.g., if a cluster contains $\{v_3, v_5, v_9\}$, this cluster is then denoted as c_3. At the very beginning of executing task clustering, each cluster in V_V contains one task, and each edge in *VVG* represents a security constraint. Fig. 5(b) illustrates the *VVG* derived from Fig. 5(a). The problem of finding the minimum number of vendors required becomes the calculation of $\omega(VVG)$. However, the maximum clique problem is NP-hard in traditional graphs.

Definition 2 (Induced Subgraph): A graph $G_A = (A, E_A)$ is an *induced subgraph* of $G = (V_G, E_G)$ when: 1) $A \subseteq V_G$; and 2) $E_A = \{(v_i, v_j) \in E_G, v_i \in A \text{ and } v_j \in A\}$ [33].

Let $V'_V(c_i)$ be the set of all adjacent nodes of c_i in *VVG* and $V_V(c_i) = V'_V(c_i) \cup \{c_i\}$. $VVG_I(c_i)$ is an induced subgraph of *VVG*, and its node set is $V_V(c_i)$. An example of $VVG_I(c_3)$ is shown in Fig. 5(c).

The number of nodes in $V_V(c_i)$ is very limited because each task only have several adjacent tasks in task graph. This makes the time of calculating $\omega(VVG_I(c_i))$ to be constant in practice. $\omega(VVG)$ can be calculated as follows.

$$\omega(VVG) = max\{\omega(VVG_I(c_i)), \forall c_i \in V_V\},$$

where

$$\omega(VVG_I(c_i)) = \omega(VVG'_I(c_i)) + 1. \quad (3)$$

where $VVG'_I(c_i)$ is derived from $VVG_I(c_i)$ by removing c_i. All nodes are connects with c_i in $VVG_I(c_i)$, making $\omega(VVG_I(c_i)) = \omega(VVG'_I(c_i)) + 1$. Fig. 5(d) illustrates an example of $VVG'_I(c_3)$.

The number of edges in $VVG'_I(c_i)$ is much smaller than that in $VVG_I(c_i)$ because most of the edges in $VVG_I(c_i)$ are connected with c_i. In this study, we first remove the isolated nodes in $VVG'_I(c_i)$, and then execute the method proposed in [34] to calculate each $\omega(VVG'_I(c_i))$.

Definition 3 (Clique Size of c_i): The *clique size* of c_i is the cardinality of the maximum clique containing c_i in *VVG*, which is denoted as $\omega(c_i)$.

After calculating the clique sizes of all clusters, a **vendor clique graph** (*VCG*) is then constructed. *VCG* is an induced subgraph of *VVG*: $VCG = (V_C, E_C)$, where $V_C = \{c_i \mid \omega(c_i) == \omega(VVG), \forall c_i \in V_V\}$.

An edge (c_i, c_j) in *VCG* is a *clique edge* when c_i and c_j are in the same maximum clique set. Contracting the non-clique edge will not reduce the clique size of *VVG*, and thus these edges will be removed from *VCG* first.

Lemma 1: An edge (c_i, c_j) in *VCG* is a clique edge when at least $\omega(VVG) - 2$ nodes in *VCG* connect both c_i and c_j.

Fig. 5(e) illustrates the *VCG* derived from the *VVG* in Fig. 5(b). There are four different maximum clique sets, i.e., $\{c_2, c_3, c_5\}$, $\{c_2, c_3, c_6\}$, $\{c'_2, c'_3, c'_5\}$, $\{c'_2, c'_3, c'_6\}$. If we contract the edge (c_2, c_3), the clique sizes of $\{c_2, c_3, c_5\}$ and $\{c_2, c_3, c_6\}$ will reduce to 2; otherwise, two edges need to be contracted to reduce the clique sizes of these two cliques. To reduce the clique size of *VVG* by contracting the least edges, we first need to determine the edges that are the most appropriate ones for contraction.

Let $deg(c_i)$ be the number of clique edges connected to c_i and there are $deg(c_i) - \omega(VVG) + 2$ different maximum clique sets containing c_i. The number of maximum clique sets eliminated after contracting e_{ij} is denoted by $num(e_{ij})$, and the *priority* of a clique edge e_{ij} is denoted by $pri(e_{ij})$, which can be calculated as follows.

$$num(e_{ij}) = min\{deg(c_i), deg(c_j)\} - \omega(VVG) + 2$$
$$pri(e_{ij}) = \begin{cases} 0, & \text{if } e_{ij} \text{ cannot be contracted;} \\ \dfrac{num(e_{ij})}{\theta_{ij}}, & \text{otherwise;} \end{cases} \quad (4)$$

where θ_{ij} be the number of security constraint violations if we contract e_{ij} in *VVG*. In practice, not all of the adjacent tasks can be clustered because some tasks must be executed on particular IP cores, and thus, the priorities of corresponding edges are set as 0.

Our aim is to eliminate all of the maximum cliques by contracting the least edges, and thus, the clique edge with the highest priority should be contracted first. The steps involved in clustering tasks to satisfy the vendor constraint are summarized in Algorithm 1. This task clustering method

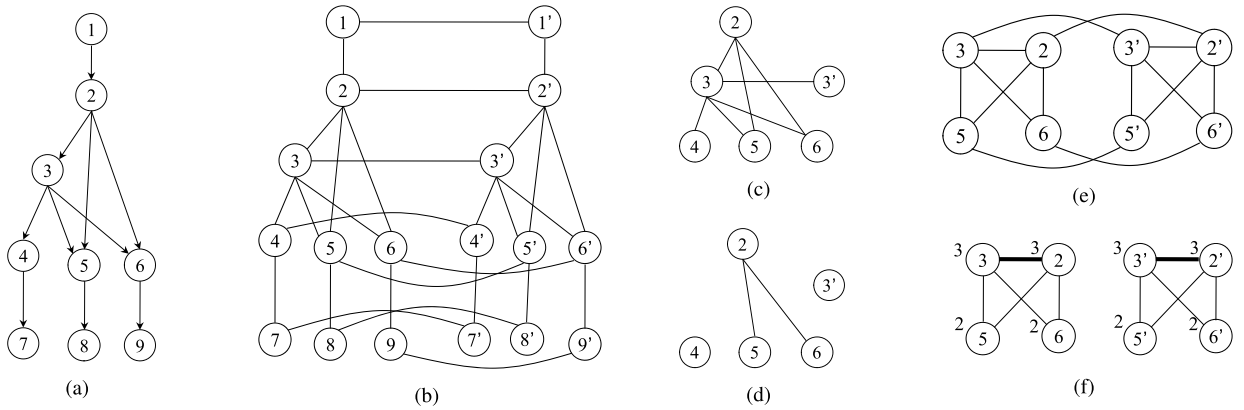


FIGURE 5. Example of calculation of $\omega(VVG_I(c_3))$. (a) Example of *TG*. (b) Corresponding *VVG*, whose clique size is 3, and each edge in this *VVG* represents a security constraint. (c) $VVG_I(c_3)$. (d) $VVG'_I(c_3)$ after c_3 is removed. (e) The *VCG* derived from *VVG*. (f) The *VCG* with only clique edges.

Algorithm 1 Vendor-Constrained Task Clustering: $Task\_Cluster_v(TG, TG', \lambda)$

```
λ: vendor constraint;
Construct VVG from TG and TG';
while ω(VVG) > λ do
    Construct VCG from VVG;
    for each e ∈ VCG do
        if e is not a clique edge then
            VCG.del_edge(e);
        end if
    end for
    while VCG.empty()! = true do
        Calculate pri(e), ∀e ∈ VCG;
        Find the edge e_ij ∈ VCG, with the highest priority;
        Cluster(c_i, c_j);
        Update VCG;
    end while
    Update VVG, and recalculate ω(VVG);
end while
```

is also applicable for the situation that there are not enough vendors for a particular IP. Suppose the vendor constraint for this particular IP is λ' and let VVG_p be an induced subgraph of *VVG* with the node set consisting of all tasks that can only be executed by this IP. The vendor-constrained task clustering is then executed on VVG_p with the vendor constraint λ' to generate the task clustering solution.

B. CANDIDATE COLOR SET DETERMINATION

After vendor-constrained task clustering, we will first decide the set of vendors that can be assign to each task, and this process is regarded as an graph coloring problem. Each color represents a IP vendor in *VVG*, and the *candidate color set* of c_i comprises all of the colors that can be assigned to cluster c_i, which is denoted as $clr(c_i)$. The candidate color set of each cluster contains all colors at the very beginning. Each time after assigning a cluster with a specific color, we need to update the candidate color sets of certain clusters.

To determine if c_i can be colored with $color_k$, we first connect c_i with the nodes whose candidate color sets do not contain $color_k$, and then check if the clique size of this *VVG* violates the vendor constraint.

Definition 4 (Sensitive Nodes): Let $clr(c_i)$ be the candidate color set of c_i before assigning a specific color. Then, if we assign c_i with a color $color_k \in clr(c_i)$, new edges will be introduced in *VVG* to connect c_i with the nodes that $\{\forall c_j | clr(c_i) \cap clr(c_j) \neq \emptyset \wedge color_k \notin clr(c_j)\}$. These newly added edges are denoted as *sensitive edges*. A node c_m is a *sensitive node* in *VVG* if: 1) c_m connects with c_i via a sensitive edge; or 2) exists two adjacent nodes of c_m that are connected by a sensitive edge.

The clique sizes of sensitive nodes may increase by one after assigning c_i with $color_k$, but the clique sizes of the other nodes in *VVG* remain the same. Let $VVG_s(c_i)$ be an induced subgraph of *VVG* with the node set consisting of c_i and all *sensitive nodes*, and we only need to check if $\omega(VVG_s(c_i))$ violates the vendor constraint after assigning c_i with a specific color. Let $VVG'_s(c_i) = VVG_s(c_i).del\_node(c_i)$, and $VVG'_s(c_i)$ is also an induced subgraph of *VVG* whose node set consists of all sensitive nodes. All sensitive nodes are connected with c_i in $VVG_s(c_i)$, making $\omega(VVG_s(c_i)) = \omega(VVG'_s(c_i)) + 1$. Examples of $VVG_s(c_6)$ and $VVG'_s(c_6)$ are illustrated in Fig. 6(b) and Fig. 6(c), respectively, where the dashed lines represent the sensitive edges.

Lemma 2: Let λ be the vendor constraint and we remove the nodes whose clique sizes are not λ from $VVG'_s(c_i)$. $VVG'_s(c_i)$ contains a $\lambda-$clique if: 1) exists λ nodes in $VVG'_s(c_i)$; and 2) their degrees are no smaller than $\lambda - 1$.

The vendor constraint will be violated after assigning c_i with $color_k$ if $VVG'_s(c_i)$ contains a $\lambda-$clique (λ is the vendor constraint). The steps of determining the candidate color set are given in Algorithm 2, and an example showing the calculation of $clr(c_6)$ is illustrated in Fig. 6, where the vendor constraint is $\lambda = 3$. Fig. 6(a) shows the *VVG* where c_5 and c_8 have already been assigned with colors, and we need to decide whether c_6 can be assigned with $color_1$. The $VVG_s(c_6)$ derived from *VVG* is shown in Fig. 6(b), where the clique

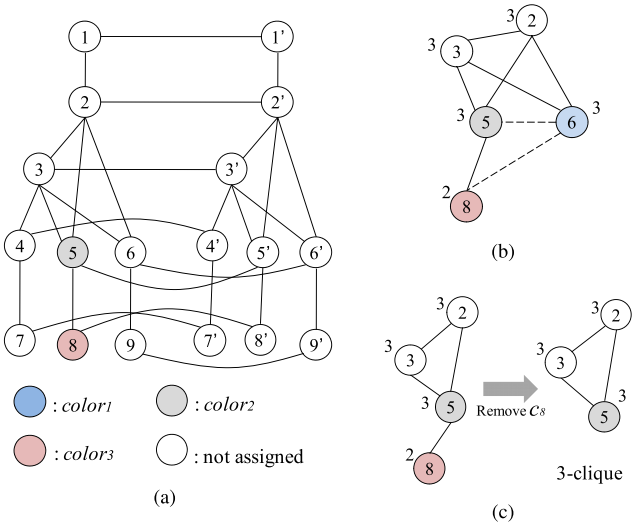


FIGURE 6. Example showing the calculation of $clr(c_6)$. (a) Example of VVG. (b) $VVG_s(c_6)$. (c) $VVG'_s(c_6)$.

Algorithm 2 Candidate Color Set Determination: $clr\_dtm(VVG, c_i)$

```
1: λ: the vendor constraint;
2: G stores the current VVG;
3: for each color color_k do
4:    Load G;
5:    Determine all sensitive nodes and construct VVG'_s(c_i);
6:    for each node v ∈ VVG'_s(c_i) do
7:       if ω(v) < λ then
8:          VVG'_s(c_i).del_node(v);
9:       end if
10:   end for
11:   counter:=0;
12:   for each node v ∈ VVG'_s(c_i) do
13:      if VVG'_s(c_i).degree(v) ≥ λ − 1 then
14:         counter++;
15:      end if
16:   end for
17:   if counter ≥ λ then
18:      Remove color_k from clr(c_i);
19:   end if
20: end for
```

size of each node is indicated next to the node. The sensitive nodes are c_2, c_3, c_5, and c_8. Then, $VVG'_s(c_6)$ comprising all sensitive nodes are constructed, as shown in Fig. 6(c). In this example, there exists three nodes whose degrees are no smaller than 2 after removing c_{12} from $VVG'_s(c_6)$, and this means that $VVG'_s(c_6)$ contains a 3-clique, and $color_1 \notin clr(c_6)$.

C. CLIQUE SIZE UPDATING

After assigning a specific color $color_k$ to a cluster c_i, the clique sizes of all sensitive nodes must be updated. Let c_j be a sensitive node in $VVG'_s(c_i)$ and ϕ be the clique size of c_j before clique size updating. The calculation of $\omega(c_j)$ after clique size updating is expressed as follows.

$$\omega(c_j) = \begin{cases} \phi + 1, & \text{if a } \phi\text{−clique containing } c_j \\ & \text{exist in } VVG'_s(c_i); \\ \phi, & \text{otherwise.} \end{cases} \tag{5}$$

Each time after updating the clique size of a sensitive node c_j, we must check if the clique size of c_i need to be updated. Let φ be the clique size of c_i before clique size updating, and the clique size of c_i ($\omega(c_i)$) is updated as follows.

$$\omega(c_i) = \begin{cases} \omega(c_j), & \text{if } \omega(c_j) == \phi + 1 \&\& \varphi < \omega(c_j); \\ \varphi, & \text{otherwise.} \end{cases} \tag{6}$$

The clique size increment of c_j is caused by connecting c_i and c_j with a sensitive edge, and thus, the clique size of c_i must be no smaller than the clique size of c_j if the clique size of c_j is increased; otherwise, the clique size of c_i remains the same. The brief steps of updating the clique sizes of nodes in VVG after assigning a specific color $color_k$ to c_i are described in Algorithm 3.

Algorithm 3 Clique Size Updating After Assigning c_i With $color_k$ in VVG: $CQ\_Update(VVG, c_i, color_k)$

```
1: λ: the vendor constraint;
2: G: the VVG before assigning c_i with color_k;
3: G': the VVG after assigning c_i with color_k;
4: Construct the VVG_s(c_i) and VVG'_s(c_i) from G;
5: for each sensitive node c_j in VVG'_s(c_i) do
6:    φ: the clique size of c_j in G;
7:    if a φ−clique containing c_j exists in VVG'_s(c_i) then
8:       ω(c_j) := φ + 1;
9:       if ω(c_i) < ω(c_j) then
10:         ω(c_i) := ω(c_j);
11:      end if
12:   else
13:      ω(c_j) := φ;
14:   end if
15: end for
```

An example showing the clique size updating after assigning c_6 with $color_1$ is illustrated in Fig. 7. The vendor constraint is set as 4, and Fig. 6(b) shows the $VVG_s(c_6)$ before color assignment. The corresponding $VVG'_s(c_6)$ is shown in Fig. 7(a) and the clique size of c_2 is the first to be updated. There is a 3-clique that contains c_2 in $VVG'_s(c_6)$, so the clique size of c_2 is updated from 3 to 4. The updated value of $\omega(c_2)$ is larger than $\omega(c_6)$, so $\omega(c_6)$ is also updated to 4. The clique sizes of the remaining clusters in $VVG'_s(c_6)$ are updated in the same manner and the final result is shown in Fig. 7(b).

D. RESOURCE-CONSTRAINED TASK SCHEDULING

Area is always sensitivity to chip designers, and it is closely related to the number of cores integrated in

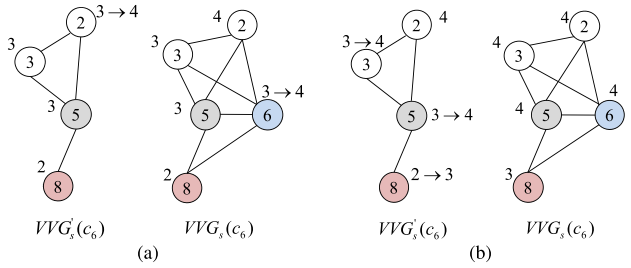


FIGURE 7. Example of updating the clique sizes of clusters in *VVG* after assigning c_6 with $color_1$. (a) $VVG_s(c_6)$ after updating the clique sizes of c_2 and c_6. (b) $VVG_s(c_6)$ after updating the clique sizes of the remaining clusters.

the MPSoCs. In this subsection, a list scheduling-based resource-constrained task scheduling method is proposed and described in Algorithm 4, where the numbers of available vendors and cores are set as constraints. In contrast to a previously proposed method [21] that conducts task scheduling and vendor assignment in different steps, we execute vendor assignment together with task scheduling to achieve better scheduling results.

Algorithm 4 List Scheduling-Based Resource-Constrained Task Scheduling: $TS_R(TG, \lambda, CORE)$

1: λ: vendor constraint; $CORE$: sets of available cores;
2: $Task\_Cluster_v(TG, TG', \lambda)$;
3: Each color is mapped to a vendor;
4: $clr(c_i)$: candidate color set of c_i, and it contains all colors at first;
5: $core(v_j)$: the set of cores that are available for task v_j;
6: **while** exist an unscheduled task v_j **do**
7: Let c_i be the cluster that $v_j \in c_i$;
8: $clr(c_i) := clr\_dtm(VVG, c_i)$;
9: Determine $core(v_j)$;
10: Schedule v_j to the core in $core(v_j)$ with the smallest $v_j.finish\_time$;
11: Schedule v'_j to the core in $core(v'_j)$ with the smallest $v'_j.finish\_time$;
12: Update the VVG;
13: $CQ\_Update(VVG, c_i, color_k)$;
14: **end while**

Vendor-constrained task clustering is first executed to enforce the vendor constraints satisfaction (*Line 1*). Then, each color is mapped to a vendor such that $clr(c_i)$ represents the set of vendors that can be assigned to the cluster c_i. $core(v_j)$ is the set of available cores from the vendors in $clr(c_i)$, where $v_j \in c_i$ (*Lines 3-5*). Each time we schedule tasks in a cluster c_i, we must first determine the candidate vendors that can assign to this cluster (*Line 8*). Then, the principle of scheduling a task v_j involves scheduling and binding it to the core in $core(v_j)$ with the smallest finish time $v_j.finish\_time$, and the duplicated task v'_j is scheduled by the same method as that employed for scheduling v_j (*Lines 7-11*). Finally, the clique sizes of all sensitive nodes in *VVG* will be updated each time after scheduling a task (*Lines 12-13*).

V. SECURITY-AWARE PERFORMANCE-CONSTRAINED TASK SCHEDULING

System performance is also one of the key considerations for chip designers, and a performance-constrained task scheduling approach is presented in this section, which minimizes the potential Trojan triggering risks under both the vendor and performance constraints. This approach solves ***Problem 2***, and it consists of four steps (see Fig. 8), which are: *performance-constrained task clustering*, *vendor-constrained task clustering*, *vendor assignment*, and *task scheduling*. As the *vendor-constrained task clustering* is identical to the resource-constrained task scheduling, we explain the other three steps next.

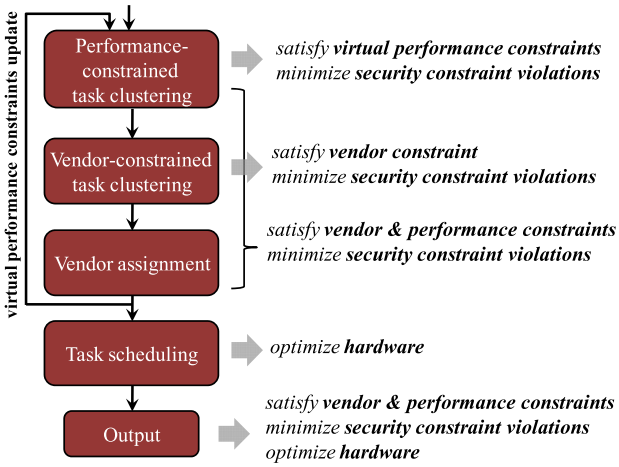


FIGURE 8. Flow of the performance-constrained task scheduling approach.

The delays of cores from different IP vendors may be different although they execute the same task, and therefore, it is impossible to determine the execution time for each task before vendor assignment. The virtual performance constraint is introduced in the performance-constrained task clustering, where we assume that each task is executed by the fastest core. Vendor-constrained task clustering is then conducted to ensure the vendor constraint satisfaction. After vendor assignment, the actual execution time of each task is determined, and so is the actual schedule length of this application. The virtual performance constraints are reduced if the actual schedule length violates the performance constraint, and these three steps iteratively repeated until the performance constraints are satisfied. Finally, the tasks are scheduled with a minimization of cores required.

A. PERFORMANCE-CONSTRAINED TASK CLUSTERING

Because the method of clustering tasks in TG can be also be applied to TG', we will discuss the performance-constrained task clustering in TG in this subsection. To minimize the schedule length of TG, several adjacent tasks must enter into the same core to hide the inter-core communication latency,

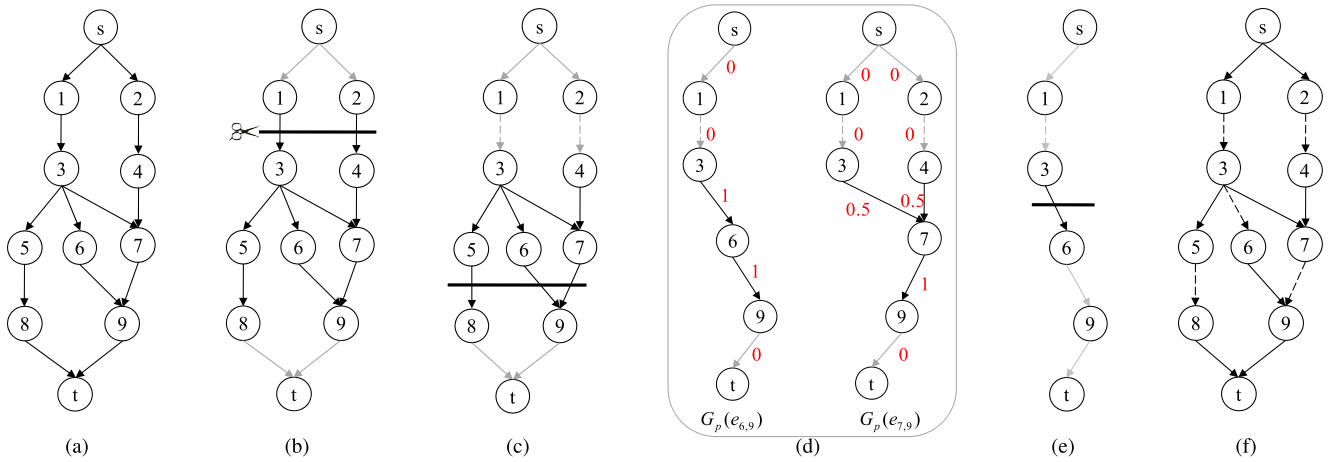


FIGURE 9. Example of performance-constrained task clustering. (a) Example of *TG* with source and sink nodes. (b) 1st iteration of min-cut. (c) 2nd iteration of min-cut. (d) 3rd iteration of min-cut. (e) Task clustering result.

and we first need to determine the timing-critical paths. The *slack time* of each task $slack(v)$ is calculated as follows.

$$slack(v) = t_{alap}(v) - t_{asap}(v) - v.exec\_time \tag{7}$$

where $t_{asap}(v)$ and $t_{alap}(v)$ are the as-soon-as-possible and as-late-as-possible schedules of v, respectively, under the given virtual performance constraint; $v.exec\_time$ is the execution time of v.

To minimize the number of vendor diversity constraints, we need to contract the least number of edges in *TG*, and a max-flow min-cut-based method is then proposed. Sink and source nodes (s, t) are added to *TG*, and two sets of edges, E_s and E_t, are also added. E_s is the set of edges pointing from s to the tasks with no input, and E_t is the set of edges pointing from the tasks with no output to t.

The **timing violation graph** (*TVG*) is an induced subgraph of *TG*, which comprises of all tasks with negative slack times. The aim of our performance-constrained task clustering is to obtain a set of contracted edges with the minimum number among all possible sets of edges that make the schedule length satisfy the performance constraint.

Definition 5 (Brother Edge): Edge e' is a *brother* of e_{ij} if it either starts from v_i or ends with v_j. $in\_edge(v)$ and $out\_edge(v)$ are the sets of all edges that end with v and start from v, respectively.

Only one edge in $in\_edge(v)$ or $out\_edge(v)$ can be contracted when optimizing the performance, and the reason is explained by the following example. Suppose e_{ij} and e_{ik} are brothers; v_j and v_k can be executed in parallel, but they must be executed sequentially if e_{ij} and e_{ik} are both contracted, which will increases the delays of the related paths.

Let edge e_{ij} point from v_i to v_j, and $\Delta dly(e_{ij})$ is the delay decrement between inter-core communication and intra-core communication of e_{ij}. The capacity of e_{ij} is denoted as $cap(e_{ij})$ and calculated as:

$$\begin{aligned} \Delta dly(e_{ij}) &= dly_{inter}(e_{ij}) - dly_{intra}(e_{ij}) \\ cap(e_{ij}) &= \begin{cases} \infty, & \text{if } e_{ij} \text{ cannot be contracted;} \\ \dfrac{1}{\Delta dly(e_{ij})}, & \text{otherwise;} \end{cases} \end{aligned} \tag{8}$$

Fig. 9 illustrates an example of max-flow min-cut-based performance-constrained task clustering, where the dashed lines are the contracted edges and the gray lines represent the edges whose capacities are ∞. Fig. 9(a) shows the *TG*, where the execution time of each task is the same, and the Δdly of all edges are 1 $u.t.$. Our aim is to minimize the total performance delay by 2 $u.t.$. In the first iteration of performance optimization, the max-flow min-cut algorithm selects $e_{1,3}$ and $e_{2,4}$, as shown in Fig. 9(b), and they are contracted in the first iteration. The *TVG* is then updated as shown in Fig. 9(c), where the max-flow min-cut algorithm selects $e_{5,8}$, $e_{6,9}$ and $e_{7,9}$ in the min-cut set, but $e_{6,9}$ and $e_{7,9}$ are brother edges, so only one of them can be contracted in this iteration.

If several edges in $in\_edge(v)$ or $out\_edge(v)$ are selected simultaneously by the max-flow min-cut algorithm (denoted as $bro\_edge(v)$), we must choose the most suitable edge to contract. Let $G_p(e_{ij})$ be the graph consists of all paths that passing though e_{ij} in *TVG*, and $dly_{opt}(G_p(e_{ij}))$ estimates the maximum delay that can be optimized in $G_p(e_{ij})$. The steps of calculating $dly_{opt}(G_p(e_{ij}))$ are summarized as follows.

1) For an edge e with $k-1$ brothers, we assume that the probability of selecting e to contract is $1/k$, and the delay that may be optimized in e is $\Delta dly(e)/k$;
2) Let $\Delta dly(e)/k$ be the *length* of e and we calculate the lengths of all edges in $G_p(e_{ij})$. Then, $dly_{opt}(G_p(e_{ij}))$ equals the shortest path length from s to t in $G_p(e_{ij})$.

The edge in $bro\_edge(v)$ with the smallest dly_{opt} is contracted because: 1) this edge has the least choices for optimizing the delays; and 2) this edge can no longer be contracted in the following iterations. Fig. 9(d) demonstrates $G_p(e_{6,9})$ and

$G_p(e_{7,9})$, and the length of each edge is denoted in red next to the edge. The values of $dly_{opt}(G_p(e_{6,9}))$ and $dly_{opt}(G_p(e_{7,9}))$ are 2 and 1.5, respectively, and thus, we contract $e_{5,8}$ and $e_{7,9}$ in the second iteration of min-cut-based task clustering. The *TVG* is then updated as shown in Fig. 9(e), and $e_{3,6}$ is finally contracted. The final task clustering result is given in Fig. 9(f) with five contracted edges. The details of performance-constrained task clustering method are provided in Algorithm 5.

Algorithm 5 Performance-Constrained Task Clustering: $Task\_Cluster_p(TG, p_{c\_v})$

1: p_{c_v}: virtual performance constraint;
2: **repeat**
3: Calculate $slack(v), \forall v \in TG$;
4: Construct *TVG*;
5: Calculate $cap(e), \forall e \in TVG$;
6: $Max\_Flow(TVG, cap, cut\_set)$; //$cut\_set$ is the set of edges selected by the max-flow min-cut algorithm;
7: For each v, calculate $bro\_edge(v)$ by $bro\_edge(v) := adj\_edge(v) \cap cut\_set$ // $adj\_edge(v)$ is the set consisting of all adjacent edges of v;
8: **while** exists v that $bro\_edge(v).num\_of\_edge > 1$ **do**
9: **for** each $e \in bro\_edge(v)$ **do**
10: Calculate the $dly_{opt}(e)$;
11: **end for**
12: Find $e_{min} \in bro\_edge(v)$, with the minimum dly_{opt};
13: **for** $e \in bro\_edge(v) \&\& e \neq e_{min}$ **do**
14: $cut\_set.del\_edge(e)$;
15: **end for**
16: **end while**
17: **for** each $e_{ij} \in cut\_set$ **do**
18: $Cluster(c_i, c_j)$;
19: **end for**
20: Calculate the schedule length sl;
21: **until** $sl \leq p_{c\_v}$

B. PERFORMANCE-CONSTRAINED TASK SCHEDULING

The methods proposed in this subsection schedule tasks under both vendor and performance constraints. The core speeds from different vendors may vary, and we must first assign tasks to vendors to determine the actual execution time of each task. Then, a modified force-directed scheduling-based algorithm is conducted to schedule all tasks into certain periods.

Algorithm 6 shows the steps followed to assign each task with the most appropriate vendor. Before vendor assignment, we cannot determine the actual execution time of each task, and we focus only on the inter-core communication delay when optimizing the schedule length. Thus, we ignore the core speed variation when executing the vendor assignment, and set the core speed to be the fastest among all vendors. If the cluster c_i contains timing-critical tasks, this cluster will be assigned to the color available with the maximum computational cost of timing-critical tasks; otherwise, this cluster will be assigned to the color available with the minimum computational cost (*Lines 6-19*). Finally, the colors with the maximum $color.cri$ are mapped to the vendors with the fastest core speed to boost the system performance (*Lines 20-24*).

Algorithm 6 Vendor Assignment: $Vendor(TG, TG', \lambda, p_{c\_v})$

1: λ: vendor constraint; p_{c_v}: virtual performance constraint;
2: **for** each $color_k$ **do**
3: $color_k.cri := 0$; $color_k.sum := 0$;
4: **end for**
5: Construct *VVG* from *TG* and *TG'*;
6: **while** exist a colorless c_i **do**
7: ρ: computational cost of all tasks in c_i;
8: τ: computational cost of all timing-critical tasks in c_i;
9: $clr(c_i) := clr\_dtm(VVG, c_i)$;
10: **if** $\tau > 0$ **then**
11: Find $color_k \in clr(c_i)$, with the maximum $color_k.cri$;
12: $c_i \leftarrow color_k$; $color_k.sum+ = \rho$; $color_k.cri+ = \tau$;
13: **else**
14: Find $color_k \in clr(c_i)$, with the minimum $color_k.sum$;
15: $c_i \leftarrow color_k$; $color_k.sum+ = \rho$;
16: **end if**
17: Update *VVG*;
18: Color the duplicated cluster c_i' in the same manner;
19: **end while**
20: Sort the vendors in descending order of core speed;
21: Sort the colors in descending order of $color.cri$;
22: **for** int i=1; $i \leq \lambda$; i++ **do**
23: Tasks with the i-th color are assigned with the i-th vendor;
24: **end for**

Then, the details of this performance-constrained task scheduling approach is presented in Algorithm 7. After executing performance-constrained task clustering, vendor-constrained task clustering, and vendor assignment under the vendor constraint λ and virtual performance constraint p_{c_v}, the actual execution time of each task is determined and the actual schedule length (sl) can also be calculated. These three steps are repeated with an iteratively reduced p_{c_v} until the final schedule length satisfies the performance constraint (*Lines 3-11*). Then, a force-directed scheduling-based method is implemented to schedule all tasks (*Lines 12-17*), and only a small number of cores is required because tasks are scheduled evenly in all time periods.

VI. EXPERIMENTAL RESULTS

A. EXPERIMENTAL SETUP

To demonstrate the effectiveness of our proposed methods, we tested a set of standard task graphs from two sources:

Algorithm 7 Performance-Constrained Task Scheduling: $TS_P(TG, \lambda, p_c)$

1: λ: vendor constraint; p_c: performance constraint; p_{c_v}: virtual performance constraint;
2: $p_{c\_v} := p_c, \sigma := 0$;
3: **repeat**
4: $G' := G := TG$;
5: $p_{c\_v} := p_{c\_v} - \sigma$;
6: $Task\_Cluster_p(G, p_{c\_v})$; $Task\_Cluster_p(G', p_{c\_v})$;
7: $Task\_Cluster_v(G, G', \lambda)$;
8: $Vendor(G, G', \lambda, p_{c\_v})$;
9: Calculate the actual schedule length of both G and G', which is denoted as sl;
10: The schedule length overhead is $\sigma := sl - p_c$;
11: **until** $\sigma \leq 0$;
12: Construct the distribution graph for each vendor [35].
13: **while** an unscheduled task v_i exists **do**
14: Schedule v_i and v_i' to the period with the smallest *force*;
15: Update the mobilities of their ancestors and successors;
16: Update the distribution graph of each vendor;
17: **end while**

task graphs that are modeled from actual application programs, including Robot control (robot), Sparse matrix solver (sparse), SPEC fpppp (fpppp); randomly generated task graphs with a large number of tasks, including rand0, rand1, rand2, rand3, and rand4. These task graphs are available from Kasahara Lab., Waseda University.[1] All of the experiments were implemented in C on a Linux Workstation with an E5 2.6-GHz CPU and 16-GB RAM.

TABLE 1. Details of the benchmarks.

task graph	tasks	edges	Para.	ACC (*u.t.*)	clique size	runtime (*s*)
robot	88	131	4.4	28.2	3	7.6
sparse	96	67	16.0	20.2	3	8.1
fpppp	334	1145	6.7	21.3	3	26.9
rand0	500	1910	27.7	10.6	3	42.7
rand1	1000	3005	60.2	7.8	3	85.7
rand2	2000	3930	151.9	10.6	3	174.7
rand3	3000	8866	157.4	8.2	3	277.2
rand4	5000	14929	246.9	5.5	3	486.5

The details of these task graphs are shown in Table 1, where column tasks and edges give the numbers of tasks and edges in each task graph, respectively. Column Para. shows the parallelism of each task graph, which is the ratio of the number of tasks to the minimum schedule length; column *ACC* gives the averaged computational cost of each task, column clique size shows the clique size of each task graph, and the column runtime presents the runtime of our proposed method required to calculate the clique size of each task graph. The results show that our method could calculate the clique size of task graph within a small computational time, and the runtime increases almost linearly with the scale of the input task graph. In the following experiments, the intra-core communication delay is ignored.

[1] Available at http://www.kasahara.elec.waseda.ac.jp/schedule/index.html.

B. RESULTS OF RESOURCE-CONSTRAINED TASK SCHEDULING

This subsection first demonstrates the comparison results between our resource-constrained task scheduling approach (TS_R) and a straight forward method [21] when all security constraints are strictly satisfied. The core speeds of different vendors may be different, and we set the step of speed differences equal to 10% of the fastest core speed in our experiments; i.e., k (*ms*) and $1.1k$ (*ms*) are required for the cores of the first and the second fastest vendors to execute a task with k (*u.t.*) computational cost, respectively. The Communication-to-Computation Ratio (*CCR*) is the ratio of the inter-core communication delay to the computational cost of the task, which also plays an important role in determining the schedule length, and the *CCR* was set to be 1.0 in this experiments. Because all security constraints are satisfied in this set of experiments, vendor-constrained task clustering method is not conducted in our TS_R.

TABLE 2. Comparisons of resource-constrained task scheduling results with all security constraints satisfied.

task graph	$core_c$	Straight forward [21]		TS_R		Δ SL (%)
		SL_A (*ms*)	runtime (*s*)	SL_B (*ms*)	runtime (*s*)	
robot	1	3194	0.2	2187	7.7	31.5
	2	1942	0.2	1484	7.7	23.6
	3	1516	0.2	1307	7.7	13.8
sparse	3	885	0.2	617	8.4	30.3
	4	710	0.2	512	8.4	27.9
	5	611	0.2	463	8.4	24.2
fpppp	1	8994	1.0	6023	29.0	33.0
	2	5106	0.9	3647	29.1	28.6
	3	3891	0.9	2880	29.0	26.0
rand0	5	1280	2.3	840	43.8	34.4
	7	941	2.3	647	43.6	31.2
	9	760	2.4	546	43.8	28.2
rand1	10	1011	5.9	688	88.4	31.5
	15	711	5.8	512	88.9	28.0
	20	565	5.9	428	89.1	24.2
rand2	25	1058	11.2	716	181.9	32.3
	37	752	11.2	533	179.8	29.1
	50	590	11.5	435	182.7	26.3
rand3	25	1245	24.9	846	310.6	32.0
	37	880	25.9	627	311.7	28.8
	50	691	25.4	506	315.2	26.8
rand4	40	854	70.8	591	571.3	30.8
	60	607	72.0	438	580.9	27.8
	80	486	71.9	359	584.5	26.1
avg.						28.2

Table 2 demonstrates the experimental results. The clique sizes of all task graphs are 3 (see Table 1), and we assume that 3 IP vendors are required for each task graph. In addition, we set the resource constraints of all vendors to be the same, and column $core_c$ gives the resource constraint, which is the number of available cores from each vendor. The SL_A and SL_B columns show the schedule lengths

generated by the straight forward method [21] and our TS_R, respectively. ΔSL equals $(SL_A - SL_B)/SL_A$, which is the ratio of the optimized schedule length to the schedule length of the straight forward method. The results demonstrate that our TS_R reduces the average schedule length by 28.2% if compared against the straight forward method. Although the runtimes of our approach are about 10 times larger than the straight forward method, the time complexities of these two approaches are equivalent. In contrast to the straight forward task scheduling method that performs the task coloring and task scheduling in different steps, our TS_R conducts task scheduling alongside task coloring, which allow us to select the core from a larger candidate core set $core(v)$ for each task v, thereby reducing the schedule length significantly.

TABLE 3. Vendor-constrained task clustering results.

task graph	clique size	λ	scy	Cluster-based		TC_v		
				scy_v	ratio (%)	scy_v	ratio (%)	runtime (s)
robot	3	2	350	7	2.00	4	1.14	15.7
sparse	3	2	230	13	5.65	8	3.48	23.7
fpppp	3	2	2624	37	1.41	22	0.84	120.9
rand0	3	2	4320	172	3.98	142	3.29	112.7
rand1	3	2	7010	105	1.50	78	1.11	181.9
rand2	3	2	9860	58	0.59	36	0.37	352.9
rand3	3	2	19732	93	0.47	70	0.35	561.3
rand4	3	2	34858	106	0.31	60	0.17	984.8
avg.					1.99		1.34	

A large number of vendors will be required if we strictly follow all security constraints, and the design cost may exceed the budget. Thus, the number of IP vendors available is always given as a constraint. Then, we will demonstrate the task clustering results between our proposed vendor-constrained task clustering method TC_v and the cluster-based method [21] if the vendor constraint λ is set to 2. The comparison results are presented in Table 3, where scy and scy_v are the numbers of total security constraints and security constraint violations, respectively. Column Ratio gives the ratio of scy_v to scy, and column runtime shows the runtime required for conducting our vendor-constrained task clustering. The results of our proposed method TC_v demonstrate that the IP vendor constraint can be satisfied at a cost of 1.34% security constraint violations, while clique-based method causes an average of 1.99% security constraint violations.

C. RESULTS OF PERFORMANCE-CONSTRAINED TASK SCHEDULING

In this subsection, the results of our performance- constrained task clustering and scheduling methods are presented. The schedule length of each task graph is determined by the core speeds, and we ignore the variations of core speeds in task clustering stage because tasks are not assigned to vendors yet. Unit of time ($u.t.$) is used to describe the schedule length of each task graph.

CCR is one of the key factors that determine the inter-core communication delay, and we tested all benchmarks with three different *CCR* values: 0.1, 0.5, and 1.0. The task clustering results between our performance-constrained task clustering method TC_p and cluster-based method [21] are illustrated in Table 4. SL is the schedule length of each task graph if all security constraints are satisfied. SL_{min} is the minimum schedule length of the task graph obtained by the

TABLE 4. Comparisons of performance-constrained task clustering results.

task graph	scy	CCR	SL ($u.t.$)	SL_{min} ($u.t.$)	Cluster-based		TC_p		
					scy_v	ratio (%)	scy_v	ratio (%)	runtime (s)
robot	350	0.1	612	574	46	13.14	40	11.43	1.2
		0.5	839	644	49	14.00	44	12.57	1.4
		1.0	1114	767	47	13.43	42	12.00	1.3
sparse	230	0.1	131	126	6	2.61	4	1.73	0.4
		0.5	179	147	7	3.04	6	2.61	0.5
		1.0	236	192	6	2.61	4	1.73	0.5
fpppp	2624	0.1	1162	1087	14	0.53	12	0.46	0.8
		0.5	1590	1097	17	0.65	12	0.46	0.9
		1.0	2119	1345	16	0.61	12	0.46	1.0
rand0	4320	0.1	204	202	3	0.07	2	0.05	0.5
		0.5	280	264	6	0.14	4	0.09	0.6
		1.0	373	340	6	0.14	4	0.09	0.6
rand1	7010	0.1	137	133	29	0.41	20	0.29	0.6
		0.5	190	164	43	0.61	38	0.54	0.9
		1.0	254	210	65	0.93	50	0.71	1.0
rand2	9860	0.1	147	142	13	0.13	8	0.08	0.8
		0.5	199	168	58	0.59	42	0.43	1.0
		1.0	268	219	51	0.52	38	0.39	1.0
rand3	19732	0.1	165	158	43	0.22	32	0.16	1.1
		0.5	229	190	71	0.36	58	0.29	1.5
		1.0	304	233	135	0.68	120	0.61	2.4
rand4	34858	0.1	116	112	11	0.03	8	0.02	1.0
		0.5	160	135	72	0.21	56	0.16	1.6
		1.0	214	171	89	0.26	74	0.21	2.3
avg.		0.1				2.14		1.78	
		0.5				2.45		2.14	
		1.0				2.40		2.03	

TABLE 5. Task clustering results of our proposed performance-constrained task clustering method under a variety of performance constraints.

task graph	scy	CCR	SL ($u.t.$)	δ	SL_{con} ($u.t.$)	scy_v	ratio (%)	runtime (s)
robot	350	1.0	1114	0.90	997	6	2.22	0.3
				0.80	892	16	5.93	0.5
				0.70	780	30	11.1	0.8
sparse	230	1.0	236	0.90	208	2	0.57	0.2
				0.80	192	4	1.13	0.2
				0.70	162	18	5.08	0.2
fpppp	2624	1.0	2119	0.90	1871	2	0.08	0.2
				0.80	1623	4	0.16	0.2
				0.70	1375	6	0.25	0.3
rand0	4320	1.0	373	0.95	354	2	0.05	0.2
				0.90	340	4	0.10	0.2
				0.85	313	14	0.34	0.4
rand1	7010	1.0	254	0.95	240	4	0.06	0.2
				0.90	226	12	0.18	0.3
				0.85	217	20	0.30	0.4
rand2	9860	1.0	268	0.95	251	2	0.02	0.2
				0.90	243	8	0.08	0.3
				0.85	229	18	0.18	0.4
rand3	19732	1.0	304	0.95	288	4	0.02	0.2
				0.90	274	12	0.06	0.4
				0.85	258	30	0.15	0.5
rand4	34858	1.0	214	0.95	204	2	0.01	0.3
				0.90	194	10	0.03	0.5
				0.85	183	38	0.12	0.8

TABLE 6. Comparisons of performance-constrained task scheduling results.

task graph	scy	λ	p_c (ms)	Cluster-based [21]			TS_P				Savings	
				$core_c$	scy_v	ratio (%)	$core_{req}$	scy_v	ratio (%)	runtime (s)	scy_v (%)	$core$ (%)
robot	350	2	848	7	53	15.14	12	45	12.86	18.5	2.28	14.3
sparse	230	2	218	12	19	8.26	21	12	5.22	24.7	3.04	16.7
fpppp	2624	2	1441	30	57	2.17	49	28	1.07	54.1	1.10	18.3
rand0	4320	2	378	30	176	4.07	52	146	3.38	118.5	0.69	13.3
rand1	7010	2	231	75	122	1.74	118	98	1.40	198.2	0.34	21.3
rand2	9860	2	242	150	92	0.93	229	69	0.70	697.3	0.23	23.7
rand3	19732	2	265	150	156	0.79	248	128	0.65	1716.3	0.14	17.3
rand4	34858	2	193	300	183	0.52	486	148	0.42	1982.3	0.10	19.0
avg.						4.20			3.21		1.01	18.0

cluster-based method [21], and it is set as the performance constraint of our TC_p. The results demonstrate that our TC_p reduces the security constraint violations if compared against the cluster-based method: our TC_p enhances the system security by satisfying another 0.36%, 0.31% and 0.37% security constraints when the *CCR* are set as 0.1, 0.5 and 1.0, respectively.

Then, our proposed TC_p is tested under a variety of performance constraints, and the clustering results are demonstrated in Table 5, where the performance constraint SL_{con} is set to be $SL_{con} = \delta * SL$, and *CCR* is set to be 1.0. The clustering results show that our method TC_p could optimize the performance significantly, and only a small portion of security constraints are violated. Furthermore, the runtime of our TC_p is small even if the scale of the input task graph is large, and it increases almost linearly with the decrease of performance constraint.

Finally, our proposed performance-constrained task scheduling (TS_p) method is tested against the cluster-based scheduling method [21], and the results are presented in Table 6. The vendor constraints of the two approaches are set to be 2, and *CCR* is set to be 1.0. $core_c$ is the number of available cores from each vendor, and thus, the number of total cores required by cluster-based method is $2 * core_c$. The performance constraint p_c is set to be the schedule length of the cluster-based scheduling method [21] under the resource constraints. The column ratio gives the ratio of scy_v to scy. The percentages of saved scy_v and cores by our TS_P are given in column Savings.

The results show that our TS_P only causes a small portion of security constraint violations (3.21% on average) to satisfy the given vendor and performance constraints, and meanwhile, an average of 18.0% cores are saved if compared against the cluster-based method. In our TS_P, a limited number of iterations of task clustering and vendor assignment are executed before the performance constraint is finally satisfied, so the runtime of this approach is acceptable, although the scale of the task graph is large.

VII. CONCLUSIONS

The widespread use of COTS electronic components represents a high security risk to the MPSoCs, and thus, the idea of vendor diversity and task duplication has been recently proposed to reduce the risks of triggering hardware Trojans in MPSoC systems. However, these security constraints also bring seriously side effects on the design cost, area and system performance. In this work, we explore the multi-dimension design space between performance, area, security, and two task scheduling approaches are developed for resource-security, performance-security trade-offs. Meanwhile, the number of security constraint violations is minimized

In resource-constrained task scheduling approach, the clique size of the vendor violation graph is accurately calculated, and this allows us to optimize the number of IP vendors with a minimum number of security constraint violations. Then, vendor assignment is performed alongside task scheduling, which significantly optimizes the schedule length. In performance-constrained task scheduling approach, the proposed max-flow min-cut-based method enables a trade-off between security-performance, and only a minimum number of security constraint violations is caused to meet the desired performance. Furthermore, this approach schedules tasks to appropriate cores and time periods, and only a small number of cores is required. The experimental results demonstrate the effectiveness of these two approaches.

REFERENCES

[1] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware Trojans," *Computer*, vol. 43, no. 10, pp. 39–46, Oct. 2010.
[2] S. Swapp, "Scanning electron microscopy (SEM)," Univ. Wyoming, Laramie, WY, USA, Tech. Rep. [Online]. Available: https://serc.carleton.edu/research_education/geochemsheets/techniques/SEM.html
[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] K. Xiao and M. Tehranipoor, "BISA: Built-in self-authentication for preventing hardware Trojan insertion," in *Proc. Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, 2013, pp. 45–50.
[5] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *Proc. Int. Conf. Comput.-Aided Design*, 2013, pp. 824–829.
[6] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proc. Design Autom. Conf.*, Jun. 2015, pp. 1–6.
[7] K. Tiri and I. Verbauwhede, "A digital design flow for secure integrated circuits," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 25, no. 7, pp. 1197–1208, Jul. 2006.
[8] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jun. 2008, pp. 15–19.

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

[10] X. Li and B. C. Schafer, "Temperature-triggered behavioral IPs HW Trojan detection method with FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl.*, Sep. 2015, pp. 1–4.

[11] R. JayashankaraShridevi, D. M. Ancajas, K. Chakraborty, and S. Roy, "Runtime detection of a bandwidth denial attack from a rogue network-on-chip," in *Proc. Int. Symp. Netw.-Chip*, Sep. 2015, pp. 1–8.

[12] A. Kulkarni, Y. Pino, and T. Mohsenin, "SVM-based real-time hardware Trojan detection for many-core platform," in *Proc. 17th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2016, pp. 362–367.

[13] A. Kulkarni, Y. Pino, M. French, and T. Mohsenin, "Real-time anomaly detection framework for many-core router through machine learning techniques," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 3, no. 1, pp. 1–22, 2016.

[14] 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. Conf.*, Mar. 2012, pp. 1000–1005.

[15] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, "Belling the CAD: Toward security-centric electronic system design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 34, no. 11, pp. 1756–1769, Nov. 2015.

[16] K. Jiang, P. Eles, and Z. Peng, "Optimization of secure embedded systems with dynamic task sets," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1765–1770.

[17] X. Cui, K. Ma, L. Shi, and K. Wu, "High-level synthesis for run-time hardware Trojan detection and recovery," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf.*, Jun. 2014, pp. 1–6.

[18] T. Reece, D. B. Limbrick, and W. H. Bobinson, "Design comparison to identify malicious hardware in external intellectual property," in *Proc. IEEE Int. Conf. Trust Secur. Privacy Comput. Commun.*, Changsha, China, Nov. 2011, pp. 639–646.

[19] J. Rajendren, H. Zhang, O. Sinanoglu, and R. Karri, "High-level synthesis for security and trust," in *Proc. Int. On-Line Testing Symp. (IOLTS)*, 2013, pp. 232–233.

[20] J. Rajendran, O. Sinanoglu, and R. Karri, "Building trustworthy systems using untrusted components: A high-level synthesis approach," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 9, pp. 2946–2959, Sep. 2016.

[21] C. Liu, J. Rajendran, C. Yang, and R. Karri, "Shielding heterogeneous MPSoCs from untrustworthy 3PIPs through security-driven task scheduling," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 4, pp. 461–472, Dec. 2014.

[22] A. Sengupta and S. Bhadauria, "Untrusted third party digital IP cores: Power-delay trade-off driven exploration of hardware Trojan secuired datapath during high level synthesis," in *Proc. Great Lakes Symp. VLSI*, 2015, pp. 167–172.

[23] X. Wang and R. Karri, "NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May 2013, pp. 1–7.

[24] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: Threat analysis and countermeasures," *Proc. IEEE*, vol. 102, no. 8, pp. 1229–1247, Aug. 2014.

[25] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust*, Jun. 2010, pp. 5–6.

[26] M. Banga and M. S. Hsiao, "Trusted RTL: Trojan detection methodlogy in pre-silicon designs," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust*, Jun. 2010, pp. 56–59.

[27] J.-Y. Jou and C.-N. J. Liu, "Coverage analysis techniques for HDL design validation," in *Proc. IEEE Asia–Pacific Conf. Chip Design Lang.*, Oct. 1999, pp. 48–55.

[28] 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, Jun. 2011.

[29] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *Proc. IEEE VLSI Test Symp.*, Apr. 2012, pp. 23–25.

[30] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *Proc. IEEE Int. Conf. VLSI Design*, Jan. 2016, pp. 547–552.

[31] D. Gizopoulos *et al.*, "Architectures for online error detection and recovery in multicore processors," in *Proc. Design, Autom. Test Eur. Conf.*, Apr. 2011, pp. 533–538.

[32] R. Kalayappan and S. R. Sarangi, "SecCheck: A trustworthy system with untrusted components," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2016, pp. 379–384.

[33] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, 2nd ed. Amsterdam, The Netherlands: North Holland, 2004.

[34] N. Wang, W. Zhong, C. Hao, S. Chen, T. Yoshimura, and Y. Zhu, "Leakage power-aware scheduling with dual-threshold voltage design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3067–3078, Oct. 2016.

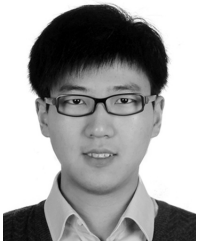
[35] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, no. 6, pp. 661–679, Jun. 1989.

NAN WANG received the B.S. degree in computer science from Nanjing University, Nanjing, China, in 2009, and the M.S. and Ph.D. degrees from the Graduate School of Information, Production and Systems, Waseda University, Japan, in 2011, and 2014, respectively.

He is currently a Lecturer of electronics and communication engineering with the East China University of Science and Technology, Shanghai, China. His current research interests include VLSI design automation, low-power design techniques, network-on-chip, reconfigurable architectures, and multiprocessor architectures.

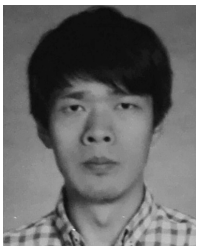
SONG CHEN received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 2000, and the M.S. and Ph.D. degrees from Tsinghua University, Beijing, China, in 2003 and 2005, respectively, all in computer science.

He served as a Research Associate with the Graduate School of Information, Production and Systems, Waseda University, Japan, from 2005 to 2009, where he also served as an Assistant Professor from 2009 to 2012. He is currently an Associate Professor with the Department of Electronic Science and Technology, University of Science and Technology of China. His current research interests include several aspects of VLSI physical design automation, on-chip communication system, and computer-aided design for emerging technologies.



JIANMO NI received the B.S. degree from Shanghai Jiao Tong University, China, in 2013, and the M.E. degrees from Waseda University and Shanghai Jiao Tong University in 2014 and 2016, respectively.

He is currently pursuing the Ph.D. degree with the University of California at San Diego. His research interests include VLSI computer-aided design, Internet of Things, and smart grid.



XIAOFENG LING received the B.S. and Ph.D. degrees from Shanghai Jiao Tong University, China, in 2006 and 2012, respectively. From 2013 to 2015, he served as the Director of research and development in a start-up company, and committed to the research and development of 5G wireless communication technology.

He is currently a Post-Doctoral Fellow with the School of Information Science and Engineering, East China University of Science and Technology. His research interests include wideband array signal processing, wireless communications, and MIMO technique.

YU ZHU received the B.S. and Ph.D. degrees in electronics and communication engineering from the Nanjing University of Science and Technology, Nanjing, China, in 1995 and 1999, respectively.

She was a Research Scholar with the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2005. She is currently a Professor of electronics and communication engineering with the East China University of Science and Technology, Shanghai, China. Her current research interests include real-time image and video analysis, computer vision, pattern recognition, and machine learning.