Security-Driven Task Scheduling for Multiprocessor System-on-Chips with Performance Constraints

Abstract—**The high penetration of third-party intellectual property (3PIP) brings a high risk of malicious inclusions and data leakage in products due to the planted hardware Trojans, and system level security constraints have recently been proposed for MPSoCs protection against hardware Trojans. However, secret communication still can be established in the context of the proposed security constraints, and thus, another type of security constraints is also introduced to fully prevent such malicious inclusions. In addition, fulfilling the security constraints incurs serious overhead of schedule length, and a two-stage performance-constrained task scheduling algorithm is then proposed to maintain most of the security constraints. In the first stage, the schedule length is iteratively reduced by assigning sets of adjacent tasks into the same core after calculating the maximum weight independent set of a graph consisting of all timing critical paths. In the second stage, tasks are assigned to proper IP vendors and scheduled to time periods with a minimization of cores required. The experimental results show that our work reduces the schedule length of a task graph, while only a small number of security constraints are violated yes.**

Index Terms—**MPSoC, hardware Trojan, security, task scheduling, system performance.**

I. Introduction

The increased design productivity requirements for heterogeneous Multiprocessor System-on-Chip (MPSoC) require the industry to procure and use the latest Commercial-Off-The-Shelf (COTS) electronic components to track the most cutting edge technology while reducing the manufacturing costs. However, the hardware Trojans in these COTS components present high risks of malicious inclusions and data leakage in products [**?**]. Particularly, the growing number of mission-critical applications (e.g., finance and military) that use MPSoCs means that security is the highest priority issue, whereas the increasing integration of third-party Intellectual Property (3PIP) and the outsourcing of fabrication lead to the fact that most of the MPSoCs are not 100% trustworthy.

Emerging security problems bring an urgent need for detecting possible hardware Trojan attacks or for muting the effects. Researchers have proposed several methods for detecting hardware Trojans, which can mainly be classified into four groups: physical inspection [**?**], functional testing [**?**], built-in tests [**?**], and side-channel analyses [**?**]. However, it is impossible to detect all of the hardware Trojans in 3PIP cores, even with the most advanced technology.

System-level security-aware design methods have also been proposed to guard systems. Beaumont *et al.* [**?**] developed an online Trojan detection architecture that implements fragmentation, replication, and voting. Rajendran *et al.* [**?**] 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.* [**?**] proposed a novel, secure-embedded systems design framework for optimizing the runtime quality. Cui *et al.* [**?**] implemented both Trojan detection and recovery at run-time, which are essential for mission-critical applications.

Recently, a set of researchers have developed the design-for-security techniques in the context of MPSoCs [**?**] [**?**]. Rajendren *et al.* [**?**] incorporated security constraints into the higher level design of SoCs and proposed a scheduling method for avoiding malicious output and collusion between vendors. Building on the security constraints, researchers also began to reduce the power/area/delay overhead of the technique [**?**] [**?**].

However, secret communications between tasks may still be established in the context of the security constraints proposed in [**?**]- [**?**], and another type of security constraints is introduced in this study to guard the MPSoC systems. Furthermore, fulfilling the security constraints always incurs a significant overhead of performance delay, which is sensitive to the designers, and thus, a security-driven performance-constrained task scheduling method is also developed. Firstly, an edge contraction conflict graph (*ECCG*) is constructed from a timing violated graph, which consists of all paths whose delays exceed the performance constraint, and sets of edges are contracted by iteratively calculating the maximum weight independent set of *ECCG*. Then, tasks are assigned to vendors with respect to the core minimization, and they are scheduled evenly in time periods by force-directed scheduling method [**?**]. The experimental results demonstrate the high quality of the task scheduling result in reducing both the schedule length and the number of cores integrated, while most security constraints are satisfied.

II. Security Constraints and Problem Description

A. Threat Model

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 information.

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 source code, which may contain Trojans, and when a Trojan exists in an IP core, all the fabricated ICs 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 Trojan in an IP core to be an extremely challenging task.

However, many applications such as banking and military systems have high security requirements, and detecting all hardware Trojans is impossible even though with the most cutting-edge technologies; therefore, hardware Trojan protection strategy during high level synthesis requires attention, and several system-level design-for-security methodologies have been proposed.

B. Security-Driven Constraints

The recently proposed security constraints [?]- [?] enable a trustworthy design, and the IPs can be purchased from different vendors without worrying about their individual secruity problems. All tasks are scheduled and bound to cores under the following two security constraints.

*1) **Security Constraint 1:** Task duplication:* The probability that the Trojans implanted by different attackers have the same trigger input is quite low, and it is virtually impossible that two cores from different vendors will output the same tampered results after the same trigger input. Thus, each task is duplicately executed on the cores from different vendors, and the outputs of these cores will be compared by a trusted component (not designed by the third party) to ensure the trustworthiness of the comparison step [?]. If the comparison fails, all the dependent tasks are terminated and a security flag is raised.

*2) **Security Constraint 2**: Vendor diversity between parent and its children:* To mute the Trojan footprint, attackers always distribute Trojans in multiple IP cores and construct secret communications between IP cores to leak information, or to trigger the hibernating Trojans [?]. 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. Therefore, data-dependent tasks are executed on the cores from different vendors to isolate the triggered hardware Trojans.

III. Motivations and Problem Description

A. Motivation 1: Security Constraint Enhancement

An example of task scheduling with security constraints is illustrated in Fig. **??**, where the solid lines represent inter-core communications, 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; $m-n$ are the starting and finishing times. In this example, we assume that $C1$ and $C2$ are from the first and second IP vendors, respectively, and the execution time of each task is 10 units of time (*u.t.*).

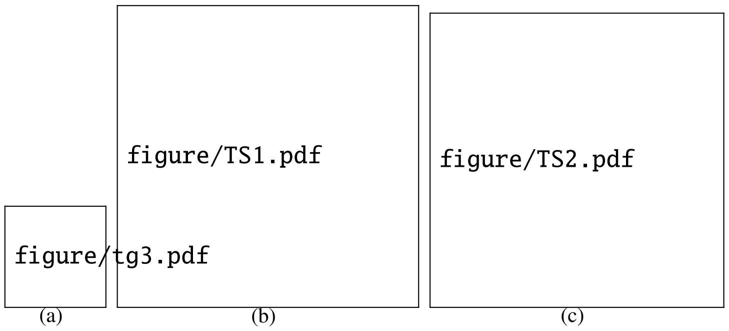


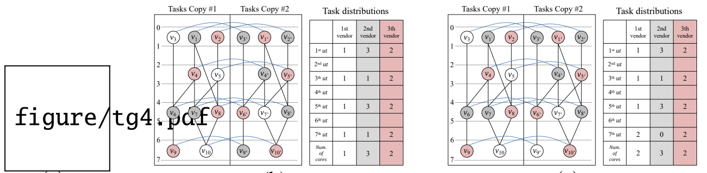
Fig. 1. Collusion between parent and its duplicated children. **??** Task graph. **??** Schedule with task duplication and vendor diversity constraints. **??** Secret communications may be established between parent and its duplicated children.

With all task duplication and vendor diversity constraints satisfied, the scheduling result of Liu *et al.* [?] is given in Fig. **??**. v_1 and its duplicated children v'_2, v'_3 are bound to the same core, and secret communications between (v_1, v'_2) and (v_1, v'_3) may be established because v'_2 and v'_3 are the duplicated tasks of v_2 and v_3; thus, Trojans in $C1$ may be triggered via (v_1, v'_2) or (v_1, v'_3). Due to the same reason, Trojans in $C2$ can also be triggered via (v'_1, v_2) or (v'_1, v_3). Fig. **??** gives all possible secret communications that may be established between tasks and their duplicated children, which are represented by the dash lines. For this reason, the following security constraint is also introduced.

***Security Constraint 3**: Vendor diversity between parent and its duplicated children:* A parent task and its duplicated children are bound to the cores from different IP vendors.

B. Motivation 2: Trade-off Between Schedule Length and Security

With the task graph given in Fig. **??**, Fig. **??** shows all of the security constraints: black lines, blue lines and red lines represent the first, second, and third types of security constraints, respectively. Suppose a task graph has n nodes and m edges, and the number of all security constraints (denoted as scy) is $n+4m$.



Task distributions (b)

	1st vendor	2nd vendor	3th vendor
1st *ut*	1	3	2
2nd *ut*			
3th *ut*	1	1	2
4th *ut*			
5th *ut*	1	3	2
6th *ut*			
7th *ut*	1	1	2
Num. of cores	1	3	2

Task distributions (c)

	1st vendor	2nd vendor	3th vendor
1st *ut*	1	3	2
2nd *ut*			
3th *ut*	1	1	2
4th *ut*			
5th *ut*	1	3	2
6th *ut*			
7th *ut*	2	0	2
Num. of cores	2	3	2

Fig. 2. Security constraints between tasks. **??** Task graph. **??** All security constraints are satisfied. **??** Security constraints after performance optimization.

Fulfills the security constraints at the finest granularity, but this incurs significant overhead of system performance. Therefore, researchers also explore the possibility of grouping dependent tasks into a cluster and scheduling the entire cluster to a single core to hide the inter-core communication latency [?] [?]. However, Liu *et al.* [?] forget to minimize the Trojan triggering risk, and Wang *et al.* [?] ignore the task criticality variation, which is also essential in evaluating the Trojan triggering risk. The edge in task graph that connects two clustered tasks is called ***contracted edge***. Task clustering violates the *security constraint 2*, and the number of ***security constraint violations*** is denoted as scy_v

The schedule length can be significantly optimized with a small number of security constraint violations. Let the inter-core communication delay be 1 *u.t.*, and the intra-core communication delay is ignored. The target is to optimize the schedule length of the task graph in Fig. **??** by 1 *u.t.*, and the optimized result shows that only 2 out of 26 security constraints are violated if we cluster (v_5, v_6) and (v'_5, v'_6) (see Fig. **??**).

C. Problem Description

Let the task graph be $TG = (V, E)$, where V is the set of all tasks and E represents the data dependencies between tasks. The optimization problem we focused in this study is named as the security-driven performance-constrained task scheduling problem, and the performance constraint is modeled as the maximum delay that the schedule length must not exceed.

***Problem** 1:* Inputs: task graph TG, performance constraint, and three types of security constraints. The target is to find a schedule with a minimized number of security constraint violations, and the number of cores required is also optimized.

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

$$min: \ \alpha * scy_v + core \tag{1}$$

where scy_v is the number of security constraint violations, $core$ is the number of cores required by the schedule, and α is a parameter large enough to keep the minimization of scy_v as the first priority.

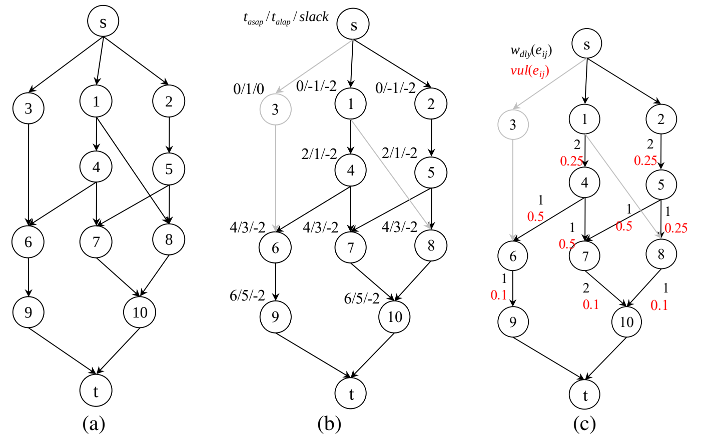


Fig. 3. Example of evaluating the timing violated graph. **??** Task graph with s and t. **??** TVG with timing constraint to be $5u.t.$ **??** The evaluation of $w_{dly}(e)$.

IV. Performance-Constrained Task Clustering

System performance is one of the key considerations for designers, and they always put several timing-critical tasks into the same core to minimize the schedule length [**?**]. However, this brings potential risk to systems security, and thus, we must minimize the potential Trojan triggering risk when optimizing the performance.

Because TG and its duplicated TG' contain the same information, and contracting the data-dependent tasks into the same cluster only violates the *security constraint 2*. Therefore, we only discuss the methods of contracting edges in TG in the following description. Let $slack(v)$ be the slack time of v under the performance constraint, and it is calculated as follows.

$$slack(v) = t_{alap}(v) - t_{asap}(v) - exec(v) \quad (2)$$

where $exec(v)$ is the execution time of task v, and $t_{asap}(v)$ and $t_{alap}(v)$ are the as-soon-as-possible and as-late-as-possible schedules, respectively.

Source and sink nodes s, t are added to TG, and directed edges that pointing from s to 0-indegree nodes, and from 0-outdegree nodes to t, are also added. An example of task graph with s and t is given in Fig. **??**. Then, a **timing violated graph** $TVG = (V_T, E_T)$ is constructed, and it is an induced subgraph of TG. V_T consists of all tasks with negative slacks, and $E_T = \{(v_i, v_j) \in E, v_i \in V_T \text{ and } v_j \in V_T\}$. Let $dly(e_{ij})$ be the inter-core communication delay of e_{ij}, and its intra-core communication delay is ignored. Fig. **??** gives an example of TVG, where the performance constraint is 5 $u.t.$ and $dly(e)$ is 1 $u.t.$ for each edge.

Several edges in TVG will be contracted until the performance constraint is satisfied. However, not all edges can be contracted with respect to the multi-core parallel execution. Regarding the tasks: 1) consuming the same data, or 2) feeding their computed data to the same task, they must not be assigned to the same core. This is because assigning the tasks that they once could be parallel executed to the same core forces them to be sequentially executed, which increases the schedule length of related paths. Let $in\_edge(v)$ be the set of edges that end with v, and $out\_edge(v)$ be the set of edges that start from v. Only one of the edges in $in\_edge(v)$ or $out\_edge(v)$ can be contracted.

Contracting an edge (e_{ij}) with $k(u.t.)$ minimizes the lengths of all paths that passing though e_{ij} by $k(u.t.)$. Let $w_{dly}(e_{ij})$ be the schedule length of TVG that can be optimized after contracting e_{ij}, and it is estimated by the following equations.

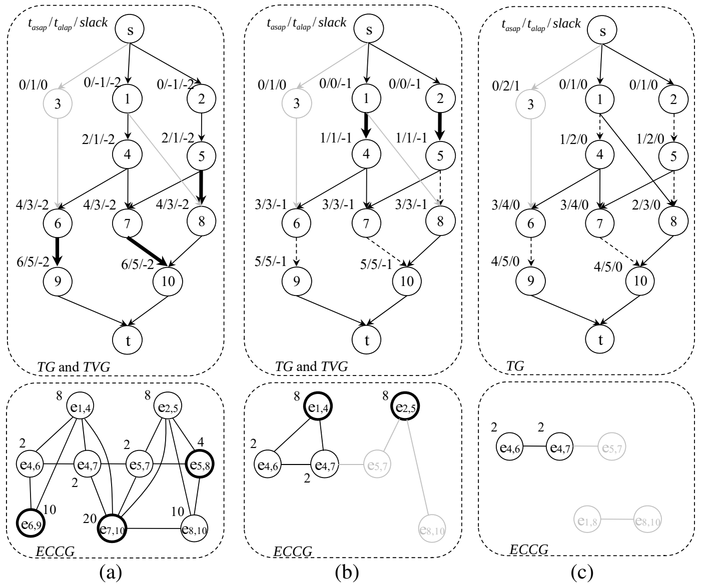


Fig. 4. Example of performance-constrained task clustering process. **??** The TVG and its corresponding $ECCG$ before task clustering. **??** The TVG and its corresponding $ECCG$ after the 1st iteration of task clustering. **??** The TVG and the corresponding VCG after the 2nd iteration of task clustering.

$$w_{dly}(e_{ij}) = \frac{path_{tvg}(e_{ij})}{path_{tvg}} * dly(e_{ij}) \quad (3)$$

where $path_{tvg}(e_{ij})$ is the number of paths in TVG that pass through e_{ij}, and $path_{tvg}$ is the number of all paths in TVG. Fig. **??** illustrates the w_{dly} in TVG, which is indicated next to the edge.

Clustering of timing critical tasks also necessitates information of task criticality. The total weight that evaluates these effects of contracting an edge e_{ij} in TVG is denoted as $w(e_{ij})$, which is calculated as follows.

$$w(e_{ij}) = \frac{w_{dly}(e_{ij})}{\beta * cri(v_i)} \quad (4)$$

where β is the user-determined factor, and $cri(v_i)$ is the task criticality of v_i.

Then, an **edge contraction conflict graph** ($ECCG$) is constructed to represent if every pair of edges can be contracted simultaneously. Set $ECCG = (V_E, E_E)$, and each vertex in V_E represents an edge in TVG that can be contracted. Two vertexes in V_E are connected when their corresponding edges cannot be contracted simultaneously, under one of the following two situations: 1) these two edges are in the same $in\_edge(v)$ or $out\_edge(v)$ (respect to the multi-core parallel execution); 2) these two edges are in the same path (for each path in TVG, only one edge can be contracted in each iteration, such that the path length will not be over optimized).

Clustering tasks may increase the number of IP vendors required, and if contracting an edge violate the IP vendor constraints (the method in [**?**] is applied to calculate the number IP vendors), such edge will be removed from $ECCG$. The maximum weight independent set (MWIS) of the weighted $ECCG$ is calculated by the method proposed in [**?**], and the set of edges in MWIS will be contracted with the maximum benefits among all possible optimization results.

An example of task clustering derived from the TG in Fig. **??** is given in Fig. **??**, where we are about to optimize the schedule length by 2 *u.t.*, and the criticalities of all tasks are assumed to be the same. The TVG consists of the nodes and edges with black color, and the dashed lines are the contracted edges. The TVG and its corresponding $ECCG$ are first constructed (Fig. **??**), and its MWIS is $\{e_{1,4},\ e_{2,5}\}$ which will be contracted to minimize the schedule length in the first iteration. Then, TVG is updated and $ECCG$ is re-constructed as shown in Fig. **??**, whose MWIS is $\{e_{5,8}, e_{6,9}, e_{7,10}\}$. After contracting these edges, Fig. **??** gives the final clustering result.

V. Performance-Constrained Task Scheduling

With the task clustering results, we will decide the IP vendor assignment for each cluster, and schedule tasks to time periods to minimize the number of IP cores required. The **vendor conflict graph** (VCG) is constructed from the clustering results: $VCG = (V_V, E_V)$, 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. Fig. **??** gives an example of VCG derived from the task clustering result.

***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 clique size of VCG equals the minimum number of vendors required, and it is calculated by the method proposed in [**?**]. The *candidate vendor set* of c_i comprises all of the IP vendors that can be assigned to cluster c_i, which is denoted as $cvs(c_i)$. The c_i that can be assigned with ipv_k only when

1) $\forall c_j \in VCG.adj\_node(c_i)$, $cvs(c_j) - ipv_k \neq \emptyset$;
2) If v_i is assigned with ipv_k, the clique size of the new VCG will not violate the vendor constraint.

At the very beginning of vendor assignment, we assume that each cluster can be assigned with all IP vendors, and each IP vendor in *candidate vendor set* has the same possibility to be assigned to the tasks in this cluster. E.g., if four IP vendors are in the $cvs(c_i)$, and the possibility of assigning c_i to each IP vendor is 0.25. Then, the *mobilities* of all tasks are calculated and the *distribution graph* [**?**] for each IP vendor is constructed to estimate the number of cores required during vendor assignment. Let ipv_k be one of the p candidate vendors in $cvs(c_i)$, and $prob(v_i, t_j)$ is the probability that v_i is in t_j. The probability that all tasks $v_m \in c_i$ are in t_j and assigned to ipv_k is denoted as $prob(c_i, t_j, ipv_k)$, which is calculated as follows.

$$prob(c_i, t_j, ipv_k) = \frac{1}{p} \sum_{v_m \in c_i} prob(v_m, t_j) \tag{5}$$

The next step is to take the summation of the probabilities of tasks assigned with the same vendor for each time period. The resulting distribution graph (*DG*) indicates the concurrency of tasks assigned to ipv_k in t_j, which is calculated as follows.

$$DG(t_j, ipv_k) = \sum_{all\ clusters} prob(c_i, t_j, ipv_k) \tag{6}$$

The maximum of all $DG(t_j, ipv_k)$, $\forall t_j \in [1, p_c]$ is denoted as $DG_m(ipv_k)$, which is used to estimate the number of cores coming from the IP vendor ipv_k. The total number of cores required is denoted as *core*, and it is the sum of all estimated cores $core = \sum_{\forall ipv_k} DG_m(ipv_k)$. An example of estimating the total number of cores is given in Fig. **??**, with the clustering result demonstrated in Fig. **??**. Fig. **??** gives the VCG derived from both TG and TG', and c_3 and c_6 are assumed to be assigned to ipv_1 and ipv_3, respectively. The mobility of each task is presented in Fig. **??**, and the candidate vendor set of each cluster is given in Fig. **??**, where the number of cores required is estimated to be 7.

To assign c_i with the best IP vendor, we first estimate the number of cores required if c_i is assigned to ipv_k, $\forall ipv_k \in cvs(c_i)$, and then assign c_i to the IP vendor with the smallest number of cores required. Fig. **??** illustrates an example of assigning c_1 with proper IP vendor, where $cvs(c_1) = \{ipv_1, ipv_2, ipv_4\}$. The numbers of cores required if c_1 is assigned to ipv_1, ipv_2 and ipv_4 are 7.67, 7.33, and 7.33, respectively (see Figs. **?? ?? ??**). Thus, c_1 will be assigned to either ipv_2 or ipv_4, because their vendor assignments are equally evaluated.

Each time after assigning a cluster with a proper IP vendor, we schedule all of the tasks in this cluster by force-directed scheduling method [**?**]. Force-directed scheduling method schedules tasks evenly in time periods, resulting in a small number of cores required. Algorithm **??** illustrates the details of our security-Driven performance-constrained task scheduling algorithm.

Algorithm 1 Security-Driven Performance-Constrained Task Scheduling Algorithm, $TS(TG, p_c)$.

```
1:  $p_c$ is the performance constraint;
2:  Construct $TVG$;
3:  while $TVG$ is not empty. do
4:      Calculate the weight of each edge in $TVG$;
5:      Construct $ECCG$, and calculate its MWIS;
6:      for each $v_e \in$ MWIS do
7:          Contract $e$ in $TG$;
8:      end for
9:      Calculate $slack(v), \forall v \in TG$, and construct $TVG$;
10: end while
11: Contract the edges in $TG'$ in the same manner;
12: Construct $VCG$, calculate the mobilities of all tasks, and
    initialize the candidate vendor sets of all clusters.
13: for each un-assigned cluster $c_i$ do
14:     Assign $c_i$ with the most suitable IP vendor $ipv_k \in cvs(c_i)$;
15:     Schedule all tasks in $c_i$ by force-directed scheduling [?],
        and update the mobilities;
16:     Update $cvs(c_j), \forall c_j \in VCG.adj\_nodes(c_i)$;
17: end for
```

VI. Experimental Results

A. Experimental Setups

All of the experiments were implemented in C on a Linux Workstation with an E5 2.6-GHz CPU and 32-GB RAM. To demonstrate the effectiveness of our proposed algorithms, we tested eight benchmarks from two sources[1]: task graphs that are modeled from actual application programs, including Robot control (robot), Sparse matrix solver (sparse), SPEC fpppp (fpppp); task graphs that are randomly generagted (rnc500, rnc1000, rnc2000, rnc3000, and rnc5000). The *communication-to-computation ratio* (CCR) is the ratio of the inter-core

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

communication delay to the computational cost of the task, and the intra-core communication is ignored in the experiments.

B. Number of Vendors Required

In this study, the third type of security constraints is introduced to enhance the system protection, and this impacts the number of vendors required. In this subsection, the numbers of vendors required are compared between the straight forward method [?], and our proposed method. In the straight forward method, only the first two types of security constraints described in Sect ?? are counted, while our method uses all three types of security constraints. The number of IP vendors required is calculated by the method proposed in [?].

We tested 100 task graphs that are randomly generated, and the numbers of tasks in these task graphs range from 100 to 200000. The comparison results are illustrated in Fig. ??, where $\omega(TG)$ is the clique size of task graph. The results indicate that the number of IP vendors required equals the clique size of task graph if we only follow the first two security constraints. However, if our proposed *security constraint 3* is also followed, the number of IP vendors increases to four if $\omega(TG) < 4$. As to the task graphs whose clique sizes are no smaller than 4, our proposed security constraints will not increase the number of vendors required.

C. Performance-Constrained Task Clustering Results

The IP vendor constraint is set to be 4 for all benchmarks, and all three types of security constraints are counted in the following experiments. Table I gives the task clustering results. The cluster-based method (*Cluster*) proposed in [?] clusters the tasks in critical paths to optimize the schedule length, however, this method does not minimize the number of security constraint violations. Our proposed task clustering method tries to maintain a high security level when optimizing the system performance. Task criticalities for all tasks are first assumed to be the same, and thus, maximizing the security is equivalent to minimizing the scy_v; its clustering results are given in column TC_1. Then, the task criticality of v_i is assumed to be the distance between s and v_i, because the computational results may contain more confidential information when the application proceeds, and the damage to the system is also much more serious. Its corresponding clustering results are given in column TC_2.

CCR is set to be 1.0, and the performance constraint P_c is set as $P_c = \delta * \text{SL}$, where SL is the schedule length with all security constraints satisfied. Two performance constraints are tested for each benchmark, with $\delta \in \{0.9,\ 0.8\}$. The results show that TC_1 violates the minimum number of security constraints, and its scy_v is reduced by 0.24% if compared against the cluster-based method. TC_2 considers the task criticality variations while clustering, and its average scy_v is 0.20% less than that of cluster-based method.

D. Performance-Constrained Task Scheduling Results

The scheduling results of the clustered-based task scheduling method [?] and our proposed task scheduling method are compared in Table ??, and columns *Cluster* and *TS* demonstrate their scheduling results, respectively. Two *CCRs* (0.5 and 1.0) are tested, and the task criticality variations are ignored. The performance constraint is set to be $P_c = 0.8 * \text{SL}$, *ratio* is the ratio of scy_v to scy, and *core* is the total number of cores required by the scheduling result.

TABLE I
COMPARISONS OF PERFORMANCE-CONSTRAINED TASK CLUSTERING RESULTS.

task graph	scy	SL (*u.t.*)	P_c (*u.t.*)	*Cluster* [?]		TC_1		TC_2	
				scy_v	%	scy_v	%	scy_v	%
robot	612	1114	997	10	1.63	6	0.98	6	0.98
			892	24	3.92	16	2.61	18	2.94
sparse	364	236	208	4	1.10	2	0.55	2	0.55
			192	6	1.65	4	1.10	4	1.10
fpppp	4914	2119	1871	6	0.12	2	0.04	2	0.04
			1623	8	0.16	4	0.08	4	0.08
rnc500	8140	373	340	6	0.07	4	0.05	4	0.05
			300	26	0.32	18	0.22	22	0.27
rnc1000	13020	254	226	16	0.12	12	0.09	14	0.11
			203	86	0.66	62	0.48	68	0.52
rnc2000	17720	268	243	14	0.08	8	0.05	10	0.06
			219	52	0.29	38	0.21	42	0.24
rnc3000	38464	304	274	14	0.04	12	0.03	14	0.04
			243	98	0.26	82	0.21	88	0.23
rnc5000	64716	214	194	16	0.03	10	0.02	12	0.02
			171	92	0.14	74	0.11	80	0.12
avg.					0.66		0.42		0.46

The comparison results show that, when $CCR = 0.5$, our TS reduces scy_v by 0.35% if compared against the cluster-based method; In addition, the number of cores required by our TS is 19.18% less than the cluster-based method. When $CCR = 1.0$, our TS saves scy_v and *core* by 0.30% and 18.59%, respectively. The runtime of our TS is about 10 times larger than the cluster-based method, but the time complexities of these two methods are both $O(n^3)$, where n is the number of tasks in TG.

VII. CONCLUSIONS

The security constraints introduced in this paper better protect the system from the malicious inclusions and data leakage due to the planted hardware Trojans. A design-for-security task scheduling approach is also proposed to improve the task scheduling result by reducing both the schedule length and the number of cores integrated, with only a small increase in the risk of Trojan triggering. In the first stage, the data-dependent tasks with negative slacks are iteratively clustered to reduce the schedule length until the given performance constraint is met. In the second stage, each cluster is assigned with a most proper IP vendor and tasks are scheduled by force-directed scheduling method, such that the number of cores required is minimized. The experimental results demonstrate that our proposed approach significantly minimizes the number of cores integrated in the MPSoC with performance constraint, while most of the security constraints are satisfied.

REFERENCES

[1] X. Wang and R. Karri, “NumChecker: detecting kernel control-flow modifying rootkits by using hardware performance counters,” *Proc. Design Automation Conference*, pp. 1-7, May 2013.

[2] S. Swapp, *Scanning Electron Microscopy (SEM)*, University of Wyoming.

[3] M. Banga and M.S. Hsiao, “A novel sustained vector technique for the detection of hardware trojans,” *Proc. International Conference of VLSI Design*, pp. 327-332, Jan. 2009.

[4] K. Xiao and M. Tehranipoor, “BISA: Built-in self-authentication for preventing hardware Trojan insertion,” *Proc. International Symposium on Hardware-Oriented Security and Trust*, pp. 45-50, 2013.

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

TABLE II
COMPARISONS OF PERFORMANCE-CONSTRAINED TASK SCHEDULING RESULTS.

task graph	tasks	*scy*	*CCR*	SL ($u.t.$)	P_c ($u.t.$)	*Cluster* [?]				*TS*				Savings	
						scy_v	ratio (%)	*core*	runtime (s)	scy_v	ratio (%)	*core*	runtime (s)	scy_v (%)	*core* (%)
robot	88	612	0.5	839	671	28	4.58	14	3.4	18	2.94	11	23.5	1.64	21.43
			1.0	1114	892	24	3.92	14	3.2	16	2.61	10	21.6	1.31	28.57
sparse	96	364	0.5	179	143	6	1.65	21	5.1	4	1.10	16	33.8	0.55	23.81
			1.0	236	189	6	1.65	19	4.8	4	1.10	15	34.7	0.55	21.05
fpppp	334	4914	0.5	1590	1272	10	0.20	13	7.5	4	0.08	10	57.8	0.12	23.08
			1.0	2119	1695	8	0.16	12	7.2	4	0.08	10	58.4	0.08	16.67
rnc500	500	8140	0.5	280	224	32	0.39	68	18.5	20	0.25	61	112.3	0.14	10.29
			1.0	373	300	26	0.32	67	17.2	18	0.22	58	108.9	0.10	13.43
rnc1000	1000	13020	0.5	190	152	96	0.74	95	34.7	68	0.52	81	207.4	0.22	14.74
			1.0	254	203	86	0.66	87	36.3	62	0.48	76	203.8	0.18	12.64
rnc2000	2000	17720	0.5	199	159	54	0.31	217	57.3	42	0.24	168	667.5	0.07	22.58
			1.0	268	214	52	0.29	206	53.8	38	0.21	171	673.5	0.08	16.99
rnc3000	3000	38464	0.5	229	183	106	0.28	238	158.6	86	0.22	189	1873.2	0.06	20.59
			1.0	304	243	98	0.26	226	154.5	82	0.21	173	1923.3	0.05	23.45
rnc5000	5000	64716	0.5	160	128	102	0.16	448	278.5	82	0.13	372	2289.6	0.03	16.96
			1.0	214	171	92	0.14	427	259.6	74	0.11	359	2305.7	0.03	15.93
avg.			0.5											0.35	19.18
			1.0											0.30	18.59

[6] M. Beaumont, B. Hopkins, and T. Newby, “SAFER PATH: security architecture using fragmented execution and replication for protection against Trojaned hardware,” *Proc. Design, Automation & Test in Europe Conference*, pp. 1000-1005, Mar. 2012.

[7] J. Rajendran, et al., “Belling the CAD: toward security-centric electronic system design,” *IEEE Trans. Comput.-Aided Design of Integr. Circuits and Syst.*, vol. 34, no. 11, pp. 1756-1769, Nov. 2015.

[8] K. Jiang, P. Eles, and Z. Peng, “Optimization of secure embedded systems with dynamic task sets,” *Proc. Design, Automation & Test in Europe Conference*, pp. 1765-1770, Mar. 2013.

[9] X. Cui et al., “High-level synthesis for run-time hardware Trojan detection and recovery,” *Proc. Design Automation Conference*, pp. 1-6, Jun. 2014.

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

[11] C. Liu, J. Rajendran, C. Yang, and R. Karri, “Shielding heterogeneous MPSoCs from untrustworthy 3PIPs through security-driven task scheduling,” *IEEE Trans. Emerging Topics in Computing*, vol. 2, no. 4, pp. 461-472, 2014.

[12] J. Rajendren, H. Zhang, O. Sinanoglu, and R. Karri, “High-level Synthesis for Security and Trust”, *Proc. International On-Line Testing Symposium*, pp. 232-233, 2013.

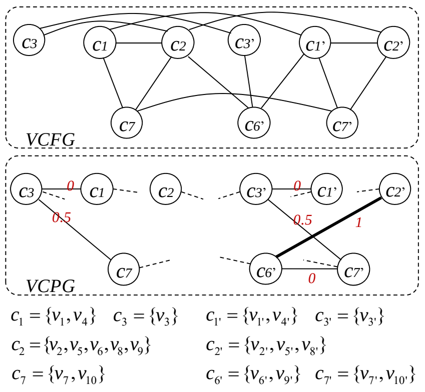
[13] N. Wang, S. Chen, J. Ni, X. Ling, and Y. Zhu, “Security-aware task scheduling using untrusted components in high-level synthesis,” *IEEE Access*, 2018, in press.

[14] 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,” *Proc. Great Lakes Symposium on VLSI*, pp. 167-172, 2015.

[15] P.G. Paulin and J.P. Knight, “Force-directed scheduling for the behavioral synthesis of ASIC’s,” *IEEE Trans. Comput. Aided-Design of Integr. Circuits and Syst.*, vol. 8, no. 6, pp. 661-679, Jun. 1989.

[16] D. Gizopoulos *et al.*, “Architectures for online error detection and recovery in multicore processors,” *Proc. Design, Automation and Test in Europe Conference*, pp. 533-538, Apr. 2011.

[17] X. Tang, H. Zhou, and P. Banerjee, “Leakage power optimization with dual-V_{th} library in high-level synthesis,” *Proc. Design Automation Conference*, pp. 202-207, 2005.


VCFG
VCPG
c3 c1 c2 c3' c1' c2' c7 c6' c7'
0
0.5
1
$c_1 = \{v_1, v_4\}$ $c_3 = \{v_3\}$ $c_{1'} = \{v_{1'}, v_{4'}\}$ $c_{3'} = \{v_{3'}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{8'}\}$
$c_7 = \{v_7, v_{10}\}$ $c_{6'} = \{v_{6'}, v_{9'}\}$ $c_{7'} = \{v_{7'}, v_{10'}\}$

(a)

figure/mobility1.pdf

(b)

figure/dg1.pdf

(c)

figure/dg2.pdf

(d)

figure/dg3.pdf

(e)

figure/dg4.pdf

(f)

Fig. 5. Example of vendor assignment. **??** Vendor conflict graph. **??** The mobilities of tasks. **??** The candidate vendor set of each cluster and the number of cores required. **??** 7.67 cores are required if c_1 is assigned with ipv_1. **??** 7.33 cores are required if c_1 is assigned with ipv_2. **??** 7.33 cores are required if c_1 is assigned with ipv_4.

figure/1-25-eps-converted-to.pdf

Fig. 6. Number of IP vendors required.