Security-Driven Task Scheduling under Performance Constraints for MPSoCs with Untrusted 3PIP Cores

Nan Wang, *Member, IEEE,* Songping Liu, Song Chen, *Member, IEEE,* and Yu Zhu, *Member, IEEE,*

Abstract—The high penetration of third-party intellectual property in MPSoCs gives rise to security concerns, and a set of security-driven constraints is imposed into task scheduling step of the design process to protect MPSoCs against hardware Trojan attacks. Due to the significant performance and area overheads incurred, designers start to selectively apply security-driven constraints to achieve the design targets, but they often ignore that parts of a design may be more vulnerable to hardware Trojan attacks. In this study, the differences in vulnerability to hardware Trojan attacks are also considered in the MPSoC design process, and a security-driven task scheduling method is proposed to minimize both the design's vulnerability and chip area under performance constraints. First, the schedule length is iteratively optimized by a maximum weight independent set-based method that minimizes the vulnerability increment. Second, tasks are assigned to IP vendors with a minimized number of cores required by maximizing the core sharing of tasks. Finally, tasks are scheduled to time periods using the force-directed scheduling method. Experimental results demonstrate the effectiveness of the proposed method in reducing the number of cores while maintaining system security under performance constraints.

Index Terms—MPSoC, third-party IP core, hardware Trojan, task scheduling, security.

1 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 manufacturing costs [1]. This has given rise to the trend of outsourcing the design and fabrication of third-party intellectual property (3PIP) components, which may not be trustworthy, and the hardware Trojans in these 3PIP components present high risks of malicious inclusions and data leakage in products [2]. This raises security concerns [3] because a small hardware modification by an adversary in the 3PIP cores can compromise the whole chip [4]. If such chips run safty-critical applications (e.g., in autonomous vehicles), the hardware Trojan attack may lead to catastrophic or life-threatening consequences [5]. Similarly, if these chips are used in information-critical systems (e.g., banking), the confidentiality and integrity of the user's data can be compromised [6].

Emerging security problems bring an urgent need to detect possible hardware Trojan attacks or mitigate their effects. Methods for detecting hardware Trojans can primarily be classified into the following groups: physical inspection [7], functional testing [8], built-in tests [9], and side-channel analyses [10]. However, it is impossible to detect advanced hardware Trojans, such as A2, due to its insertion stage and software triggered mechanism [11].

Design-for-trust techniques provide comprehensive protections to circuits and verify the correctness of system functionality at runtime. Incorporating security constraints in the MPSoC design process is one of the most popular design-for-trust techniques, which can mitigate the effects of the hardware Trojans and enable trustworthy computations using untrusted 3PIP cores [12]–[17]. This is achieved by duplicating tasks and mapping them on 3PIP cores from different vendors to detect Trojans that alter task outputs or mute potential Trojan effects by preventing collusion between malicious 3PIP cores from the same vendor. Designing MPSoCs with these security constraints brings significant design overheads (eg. approximately 200% area and 50% performance overheads [21]), and researchers have developed a number of solutions and created trusted designs with minimum resource overheads, performance degradation and energy consumption [18]–[21].

Some researchers have also started to consider security constraints as loose constraints (security constraints are not applied to some tasks and communications) to satisfy the design targets [22]–[24]. However, their work ignore that parts of a design are much more vulnerable to hardware Trojan attacks [25], and removing security constraints from the part of a circuit that are more susceptible to Trojan insertion may yield significant security losses [21]. Furthermore, these work only optimize the system performance in the context of security constraints, which might incurs a significant area overhead, but chip area is also one of the critical issues towards trusted design; therefore, performance and security along with chip area should be jointly considered for MPSoC design, especially for heterogeneous MPSoCs built from 3PIP cores which are untrustworthy.

In this study, we focus on the design of MPSoCs though security-driven task scheduling with performance constraints, and the goal is to minimize the design's vulnerability against hardware Trojan attacks and the number of cores required. A three-step design method that consists of task clustering, vendor assignment and task scheduling is proposed to enable MPSoC designers to

This work was supported by the National Key R&D Program of China under Grant 2022YFD2000400.

Nan Wang, Songping Liu, and Yu Zhu are with the School of Information Science and Engineering, East China University of Science and Technology, Shanghai, 200237, China.
Song Chen is with the School of Information Science and Technology, University of Science and Technology of China, Hefei, 230026, China.

achieve the desired performance, and obtain a high-security design with a small number of cores required. The contributions of the paper are summarized as follows:

1) This study treats the communications between tasks with different vulnerabilities against hardware Trojan attacks, and a maximum weight independent set-based method is proposed to minimize the design's vulnerability under performance constraints, by iteratively selecting a set of maximum weighted inter-core communications and assigning them to intra-core communications.
2) The numbers of cores are optimized in both the vendor assignment and task scheduling stages, by iteratively assigning tasks that share the most common cores to the same vendor and scheduling these tasks evenly in each time period. Furthermore, the proposed vendor assignment method evaluates the number of cores saved when clustering tasks rather than estimating the number of cores required, which speeds up the processing and provides better results.
3) This study considers core speed variation in the task scheduling process. All tasks are first assumed to be performed with the slowest speed, and after the exact core speeds are determined, the vendor assignment will be adjusted to assign the unprotected communications with security constraints in descending order of vulnerability, which further reduces the total design's vulnerability.

The remainder of this paper is organized as follows. Section II describes the related literature, and Section III demonstrates the motivations and describes the optimization problem. Section IV presents the details of the proposed task scheduling method. Section V illustrates the experimental results, and Section VI provides the conclusions.

2 Related Work

In general, the IPs procured from third-party 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 outsourced design and test services, as well as electronic design automation software tools supplied by different vendors, also make circuits vulnerable to malicious implants.

2.1 Security Countermeasures

Numerous and various functional and parametric tests are required to verify whether a 3PIP contains hardware Trojans. However, testing a black-box component is difficult and time-consuming, and it is impractical to perform such an exhaustive test for a large and complex design. Therefore, a number of countermeasures have been developed against hardware Trojans at the design stage [26]. Hardware security primitives provide built-in self-authentication against various threats and vulnerabilities arising at different phases [27]. System and architectural protection techniques prevent information leakage through hardware isolation and build trusted execution environments [28]. Side-channel protection techniques introduce noise or randomization in the software implementation to eliminate side-channel leakage [29]. IP protection techniques use hardware watermarking or steganography to protect an IP against threats [30]. Machine learning-assisted designs provide defenses against security threats or enhance system robustness [31].

Although hardware Trojan detection methods are implemented in different design stages, finding all hardware Trojans cannot be guaranteed even with the most cutting-edge technologies. However, many applications, such as banking and military systems, have high security requirements [32]. Therefore, Trojan-tolerant design methodologies are another way to protect designs from Trojan attacks [1].

2.2 Design-for-Trust

The design-for-trust techniques use strategies at design time to help detect hardware Trojans or mute the attack effects at runtime [12]. Many studies have attempted to detect malicious outputs by duplicating 3PIPs and to avoid collusion between parent and child tasks from the same vendor. Incorporating the above design constraints (i.e., security constraints) in the MPSoC design process has attracted the attention of researchers.

Reecee *et al.* [13] identified hardware Trojans through comparisons of two similar untrusted designs by testing functional differences for all possible input combinations. Beaumont *et al.* [14] developed an online Trojan detection architecture that implements fragmentation, replication, and voting. Cui *et al.* [15] implemented both Trojan detection and fast recovery at runtime for mission-critical applications, using recomputation with IP cores from different vendors. Shatta *et al.* [16] presented methodologies that detect the errors triggered by hardware Trojans in 3PIPs using voters, and recover the system by replacing the error.

Security constraints, including duplication and 3PIP vendor diversity, have been recently proposed for hardware Trojan protection, and high-level synthesis is the ideal level for incorporating security constraints [17]. However, fulfilling the security constraints in task scheduling may results significant overheads in system performance, chip area and power, and researchers have started to reduce these overheads. Rajmohan *et al.* [18] proposed a PSO-based hybrid evolutionary algorithm, and Sengupta *et al.* [19] proposed a bacterial foraging optimization-based design space exploration method to achieve a task schedule with higher security and less hardware overhead. Sun *et al.* [20] minimized the energy consumption while simultaneously protecting the MPSoC against the effects of hardware Trojans with security constraints. Cui *et al.* [21] solved the online hardware Trojan detection and recovery problem with graph-theory models that minimize the implementation cost of the design budget and area overhead. Liu *et al.* [22] proposed a set of task scheduling methods to reduce the increments of performance and hardware due to security constraints. Wang *et al.* [23], [24] optimized the design budget and system performance with a minimized number of unprotected communications.

The above techniques were developed to optimize design overheads when adopting security constraints, but the optimization space is limited [18]–[21]. Some researchers also treat security constraints as loose constraints (some constraints are not applied to tasks and communications) to achieve tight design targets, but they forget to minimize the induced design's security losses [22]–[24], and the induced chip area is also not optimized. Hardware Trojans intend to attack the targets with higher vulnerability to create larger damage to the systems or leakage the confidential information, and the vulnerabilities of tasks or communications can differ by $\times 10^3$ times in the same benchmark [25]. This indicates that the design's performance and area can be further reduced with a small penalty of vulnerability increment by removing some "proper" security constraints from tasks and communications. Therefore, we try to minimize both the circuit vulnerability against hardware Trojan attacks and the chip area under the performance constraints in this study.

3 Preliminaries and Problem Description

In this section, we present preliminaries and describe the problem considered in this study.

3.1 Threat Model

Hardware Trojan attacks are intended to affect normal circuit operation, potentially with catastrophic consequences in critical applications in the domains of banking, space and military [36]. They can also aim to leak secret information from inside a chip through covert channels or affect the reliability of a circuit through undesired process changes that cause device ware-out and long-term reliability issues [37]. From the perspective of the activation methods, hardware Trojans can be classified as either *always-on* or *conditionally triggered*. An always-on Trojan may be inserted in rarely accessed places and its footprint is kept small. Conditionally triggered Trojans hibernate initially, and are activated either by the Trojan implanter or by on-chip triggers [22].

In this study, we adopt the same threat model in [20], [22], which primarily focuses on detecting (or mitigating) malicious modifications. The Trojan may cause the task running on the malicious 3PIP to either produce incorrect output or collude with Trojans in another 3PIP core from the same vendor. As a result, the following two cases can occur at runtime: 1) *Malfunction:* due to the insertion of the malicious logic into a 3PIP core, the outputs of the infected cores will be altered at some undetectable points; 2) *Trojan collusion and Trojan triggering between Cores:* Trojans that are distributed on multiple cores to reduce the chance of being detected, and some malicious communication paths can also be established between cores by writing some illegal values to certain secret memory space. Therefore, with these secret communication channels, a malicious logic in one core can trigger the Trojans in another core, and active Trojans in different cores can collude to cause catastrophic consequences to the systems. Examples of the vulnerabilities caused by hardware Trojans in 3PIPs could be: information leakage, control flow violation, fault injection, side-channel leakage, and so on.

In this study, we target embedded platforms which execute application-specific tasks and have high security requirements, and such platforms are widely-used in auto-motive, safety-critical systems, etc [20]. The SoC used in these systems are vulnerable to various Trojans attacks when the untrusted 3PIPs get integrated into this SoC. Because the hardware Trojans in 3PIPs could be passed down the design cycle to post-silicon and all the fabricated chips contain such hardware Trojans. In such security-critical systems, designers always have prior knowledge of the application and its runtime constraints, and they can decide to purchase 3PIPs from different vendors, and bind tasks to IP cores.

3.2 Security Constraints

It is impossible to detect all hardware Trojans in 3PIPs even with the most advanced techniques, and therefore, runtime validation approaches provide a last line of defense against potentially undetected Trojans [3]. Integrating security constraints in the task scheduling process enables the runtime validation using untrusted 3PIPs, and the effectiveness of the security constraints in detecting the deliberate faults caused by Trojans and isolating the triggered Trojans are explained in [12]. The two types of security constraints which are also introduced in [17]–[24] handle two major types of Trojans: Those tampering program outputs and those leaking information through undesired communications. These security constraints are described as follows.

3.2.1 Duplication-With-Diversity

To detect hardware Trojans, each task is executed in duplicate on the cores from different vendors, and the outputs of these cores are compared by a trusted component (not designed by the third party). This type of security constraints is applied to tasks, and ensures the trustworthiness of task outputs [38].

Duplication-with-diversity is set based on the fact that the probability of Trojans implanted by different attackers having the same trigger input is quite low, and it is virtually impossible that two cores from different IP vendors will output the same tampered results after the same trigger input [39].

3.2.2 Isolation-With-Diversity

To mute undesired and potentially malicious communication paths and at the same time isolate a Trojan from the rest of the system, data-dependent tasks are executed on the cores fabricated from different IP vendors. This type of security constraint is applied to the communications between tasks, which ensures that all the valid communication paths are between 3PIPs from different vendors.

To mute Trojan footprints, attackers may distribute Trojans in multiple IP cores and construct secret communication paths between IP cores to leak information or to trigger the hibernating Trojans. These secret communication paths between IP cores from the same vendor cannot be acquired by other vendors [12]. Although redundant execution approaches, including voting architecture [14], dual/triple modular redundancy [38], and duplication-with-diversity, can detect hardware Trojans by comparing the outputs of cores from different vendors with the same input, they cannot cut off secret communications between multiple IP cores. Therefore, this isolation-with-diversity is also introduced to isolate the triggered hardware Trojans from the rest of the system.

3.3 Vulnerability Analysis

Analyzing a circuit's vulnerability against hardware Trojan attacks at different levels is a key step towards trusted design, because sections of a circuit with low controllability and observability are considered potential areas for hardware Trojans insertions [33]–[35]. The vulnerability analysis at behavioral level consists of *statement analysis* and *observerability analysis*, which quantifies the difficulty of activating each line of a code and observing internal signals [25]. The vulnerabilities of different signals vary a lot, and hardware Trojans may choose the target with low observability and high statement hardness to attack.

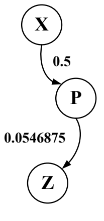
Adopted from the work presented in [25], an example of vulnerability analysis of a small behavior code (see Fig. 1) is also presented in Fig. 2. The *statement analysis* first measures the statement execution conditions. The notation $W[L, U]$ is developed for each signal where W represents the weight of the value range, and L and U show the lower and upper limits of the value range. The weight of a range is defined as $\frac{U-L+1}{U_O-L_O+1}$, where U_O and

```
1.      PORT(  CLK: IN BIT;
2.           X    : IN INTEGER RANGE 15 DOWNTO 0;
3.           Z    : OUT INTEGER RANGE 15 DOWNTO 0);
4.      PROC1 : PROCESS(CLK)
5.      VARIABLE P : INTEGER RANGE 15 DOWNTO 0;
6.      BEGIN
7.           FOR X IN 0 TO 10 LOOP
8.                IF (X<3) THEN
9.                     P := X;
10.          ELSEIF (X>5) THEN
11.               P := 14 - X;
12.          END IF;
13.     END LOOP;
14.     IF (P<7) THEN
15.          IF (X>7) THEN
16.               IF (P>2) THEN
17.                    Z <= P;
18.               END IF;
19.          END IF;
20.     END IF;
21.  END PROCESS PROC1;
```

Fig. 1. Example of code for vulnerability analysis.

Line number	Statement hardness (1/Tw)	Value range	
		X	P
6	1	{1[0,15]}	{1[0,15]}
7	1	{1[0,15]}	{1[0,15]}
8	1/0.6875	{0.6875[0,10]}	{1[0,15]}
9	1/0.1875	{0.1875[0,2]}	{0.1875[0,2]}
10	1/0.6875	{0.6875[0,10]}	{0.1875[0,2]}
11	1/0.3125	{0.3125[6,10]}	{0.3125[4,8]}
12	1/0.6875	{0.6875[0,10]}	{0.6875[0,2]∪[4,8]}
13	1	{1[0,15]}	{0.6875[0,2]∪[4,8]}
14	1	{1[0,15]}	{0.6875[0,2]∪[4,8]}
15	1/0.4375	{1[0,15]}	{0.4375[0,6]}
16	1/0.21875	{0.21875[8,15]}	{0.4375[0,6]}
17	1/0.0546875	{0.21875[8,15]}	{0.0546875[3,6]}
18	1/0.21875	{0.21875[8,15]}	{0.4375[0,6]}
19	1/0.4375	{1[0,15]}	{0.4375[0,6]}
20	1	{1[0,15]}	{0.6875[0,2]∪[4,8]}
21	1	{1[0,15]}	{0.6875[0,2]∪[4,8]}

(a)



(b)

Signal name	Target signal	Reachability	Observerability
X	Z	0.02734375	0.02734375
X	P	0.5	0
P	Z	0.0546875	0.0546875

(c)

Fig. 2. Example of vulnerability analysis. (a) Statement hardness of signals. (c) Observability of signals. (b) Data flow graph.

L_O are the declared upper and lower limits of the controlling signals. Because the hardware Trojans are always designed with small triggering probability to escape from the Trojan detection methods, and the *statement hardness* is defined as $1/W$. Fig. 2(a) gives the statement hardness of the signals X and P.

The *observerability analysis* evaluates reachability of signals and observability through circuit primary output, because a behavioral-level Trojan usually targets a signal with low observability to carry out an attack. A weight data graph of signals is constructed in Fig. 2(b), where the nodes are circuit signals and the directed edges are their dependency. For example, the signal X appears in Lines 8 and 11 where the target signal is P. The reachability of edge (X, P) is $0.1875 + 0.3125 = 0.5$, where 0.1875 and 0.3125 are shown in Fig. 2(a), but the observability of this edge is 0 as the signal P is not an output signal. The reachability and observability of each edge is presented in Fig. 2(c).

3.4 Motivations

Fulfilling security constraints may incur significant overheads of performance and area, and Fig. 3 shows an example, where 10 tasks are sorted into 4 different types. All intra-core communication delays are ignored, and the computational times of all tasks are assumed to be 1 unit of time (*ut*), which also equals their inter-core communication delays. Fig. 3(a) and Fig. 3(b) show the task graph and its as-soon-as-possible (ASAP) task schedule with security constraints, where tasks colored white, gray, and pink are assigned to the 1st, 2nd, and 3rd vendors, respectively. The duplicated task of v_i is denoted as $v_{i'}$. Satisfying all security constraints makes the schedule length 7 *ut*, and 6 cores from 3 IP vendors are required.

3.4.1 Vulnerability Increment in Performance Optimization

To reduce the performance overhead, researchers have imposed loose security constraints during task scheduling and explored the possibility of assigning data-dependent tasks into a single core to hide the inter-core communication delay [22]–[24]. In this study, clustering the connected data-dependent tasks into one core is denoted as **edge contraction**, and the edge that represents an intra-core communication is a **contracted edge**.

Edge contraction leaves the corresponding communication without protection, and this communication is vulnerable to hardware Trojans. Traditional methods that optimize system performance either ignore the consequence security loss [22] or treat every communication with the same security importance [23], [24]. However, communications have different vulnerabilities against hardware Trojan attacks, and the vulnerabilities can differ by $\times 10^3$ times in the same benchmark [25].

The example in Fig. 3(c) shows the necessity of optimizing the system performance with the consideration of vulnerability variation, where the target is to reduce the schedule length in Fig. 3(b) to 6 *ut*. Traditional methods [23], [24] contract the fewest edges, which are $e_{1,4}$ and $e_{2,5}$ (see the 1st schedule length optimization result in Fig. 3(c)). However, $e_{5,8}$, $e_{6,9}$, and $e_{7,10}$ might be less vulnerable to hardware Trojan attacks if compared to $e_{1,4}$ and $e_{2,5}$, and contracting these edges causes less vulnerability (see the 2nd schedule length optimization result in Fig. 3(c)), even though this contracts more edges.

3.4.2 Reducing Cores in Vender Assignment

In the vendor assignment stage, all tasks are partitioned into many groups such that all tasks in a group are executed by the cores from the same vendor. Traditional methods start to optimize the number of cores after the vendor assignment stage when the number of cores required can be evaluated [20]–[23]. However, the vendor assignment results also determine the number of cores required, and the example in Fig. 4 explains the reason. In this

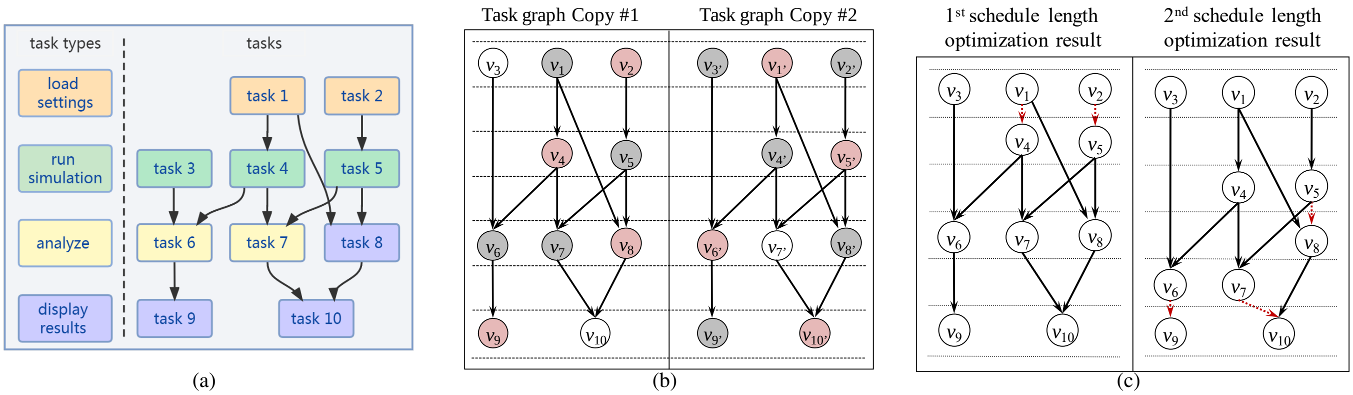
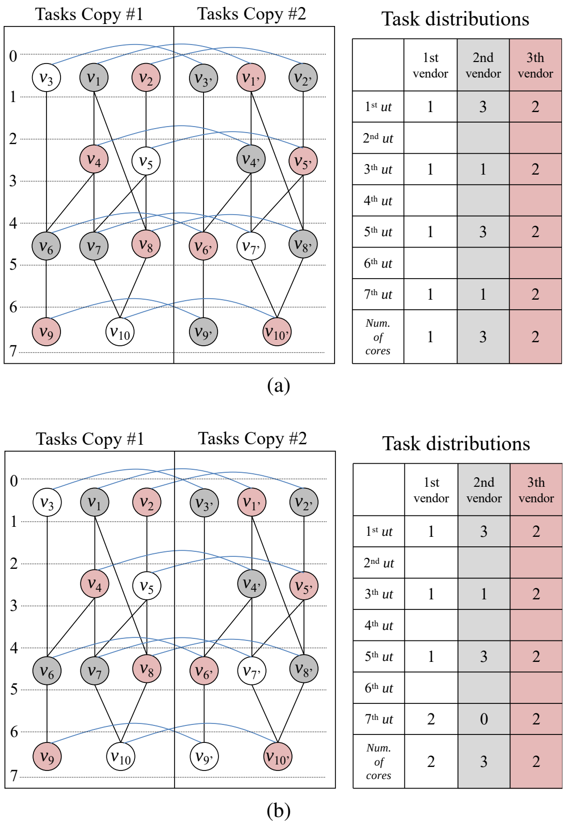

task types
tasks
load settings
run simulation
analyze
display results
task 1
task 2
task 3
task 4
task 5
task 6
task 7
task 8
task 9
task 10
(a)
Task graph Copy #1
Task graph Copy #2
v_3 v_1 v_2 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}
$v_{3'}$ $v_{1'}$ $v_{2'}$ $v_{4'}$ $v_{5'}$ $v_{6'}$ $v_{7'}$ $v_{8'}$ $v_{9'}$ $v_{10'}$
(b)
1st schedule length optimization result
2nd schedule length optimization result
(c)

Fig. 3. Example of task graph and its schedules. (a) Task graph. (b) ASAP schedule with security constraints. (c) Schedule length optimization results.


Tasks Copy #1
Tasks Copy #2
0
1
2
3
4
5
6
7
v_3 v_1 v_2 $v_{3'}$ $v_{1'}$ $v_{2'}$
v_4 v_5 $v_{4'}$ $v_{5'}$
v_6 v_7 v_8 $v_{6'}$ $v_{7'}$ $v_{8'}$
v_9 v_{10} $v_{9'}$ $v_{10'}$
(a)
(b)

Task distributions

	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

	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. 4. Example of vendor assignments. (a) Vendor assignment and its ASAP schedule, which requires 6 cores. (b) Vendor assignment and its ASAP schedule, which requires 7 cores.

example, the performance constraint is assumed to be 7 *ut*, and all security constraints are satisfied, which are represented by the blue (*duplication-with-diversity*) and black (*isolation-with-diversity*) lines between tasks. Fig. 4(a) and Fig. 4(b) give two different vendor assignments and their ASAP schedules. With the vendor assignment given in Fig. 4(a), the scheduling result requires 6 cores, but 7 cores are required with the vendor assignment shown in Fig. 4(b).

3.5 Problem Description

Clustering data-dependent tasks to reduce schedule length violates *isolation-with-diversity*, leaving the corresponding intra-core communications unprotected. Although incorporating security constraints in the design process cannot guarantee a full protection from all hardware Trojan attacks, the vulnerability against hardware Trojan attacks can be significantly reduced. In this study, the vulnerability of a communication is regarded as the reduced vulnerability after applying security constraints to this communication, and the vulnerability analysis [25] can be performed before our method to first determine the vulnerabilities of communications. The problem considered in this study can be described as follows.

Problem 1. The inputs of this problem are the task graph TG, vendor and performance constraints, core speeds of vendors, and vulnerability of each communication. The target is to find a schedule with the lowest design's vulnerability against hardware Trojan attacks, and the number of cores required is also optimized.

The design's vulnerability vul_s is regarded as the accumulated vulnerabilities of all unprotected communications, which can be calculated as follows:

$$vul_s = \sum_{e \in E_c} vul(e) \tag{1}$$

where E_c is the set of all unprotected communications, and $vul(e)$ is the vulnerability of e.

4 Security-Driven Task Scheduling with Performance Constraints

In this section, a three-step task scheduling method is proposed, and both the design's vulnerability and the number of cores required are optimized under performance constraints. The three steps of the proposed method are performance-constrained task clustering, vendor assignment with core minimization, and task scheduling.

4.1 Performance-Constrained Task Clustering

In this stage, we first apply security constraints to all tasks and communications, and then iteratively assign data-dependent tasks into the same core to meet the performance constraints with the design's vulnerability optimized. Typically, the cores produced by different vendors have different speeds, and the exact speed of each core is not yet determined; thus, we assume that tasks are performed with the slowest speed when optimizing the schedule

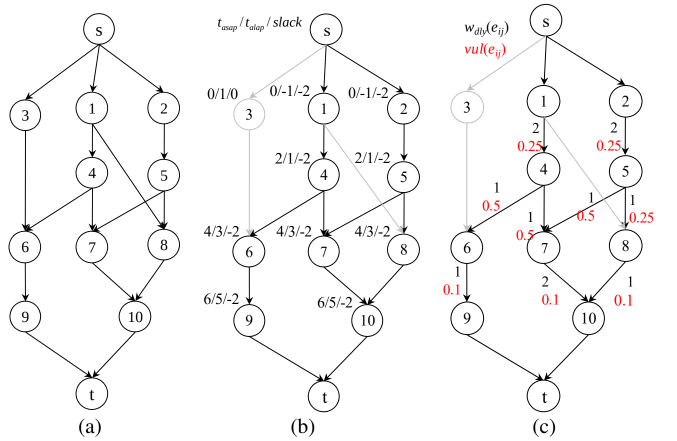


Fig. 5. Example of evaluating the timing violated graph. (a) Task graph with s and t. (b) TVG with a timing constraint of 5 ut. (c) The evaluation of $w_{dly}(e)$.

length. In addition, we discuss only the method of contracting edges in TG, and schedule length optimization of the duplicated task graph TG' can be performed in the same manner.

Source and sink nodes s and t are added to TG, and directed edges that point from s to 0-indegree nodes and from 0-outdegree nodes to t are also added. An example of the task graph from Fig. 3(a) with s and t added is given in Fig. 5(a). The **timing violated graph** ($TVG = (V_T, E_T)$) is then constructed by all paths from s to t whose lengths exceed the performance constraints, and it is an induced subgraph of TG. V_T consists of s, t and 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 $slack(v)$ be the slack time of v under the performance constraint, which is calculated as follows:

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

where $exec(v)$ is the execution time of task v, and $t_{asap}(v)$ and $t_{alap}(v)$ are the ASAP and as-late-as-possible (ALAP) schedules, respectively. Fig. 5(b) shows an example of TVG, where the performance constraint is 5 ut and the delay is 1 ut for each edge.

For application-specific IPs, data-dependent tasks might have to be executed by different IP cores, resulting in the corresponding edge being unable to be contracted. Therefore, we only focus on the edges that can be contracted to reduce the schedule length. In the following discussion, we assume that all edges can be contracted for simplicity.

Let $dly(e_{ij})$ be the inter-core communication delay of e_{ij}, and all intra-core communication delays are ignored in this study. Contracting an edge (e_{ij}) reduces the lengths of all paths that pass through e_{ij} by $dly(e_{ij})$. Let $w_{dly}(e_{ij})$ be the sum of the reduced schedule lengths of all paths (from s to t) in TVG after contracting e_{ij}, and it is calculated by the following equation:

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

where $path_{tvg}(e_{ij})$ is the number of paths in TVG that pass through e_{ij}.

Fig. 5(c) illustrates the $w_{dly}(e_{ij})$ and $vul(e_{ij})$ of all edges in TVG, which are indicated next to the edges. The target in the schedule length optimization stage is to contract the edges with larger schedule length reduction $w_{dly}(e_{ij})$ and smaller vulnerability $vul(e_{ij})$. Therefore, the total weight that evaluates an edge e_{ij} contraction, denoted as $w(e_{ij})$, can be calculated as follows:

$$w(e_{ij}) = \frac{w_{dly}(e_{ij})}{vul(e_{ij})} \tag{4}$$

However, not all edges can be contracted with respect to multicore parallel execution. Let $in\_edge(v)$ be the set of edges that end with v, and let $out\_edge(v)$ be the set of edges that start from v. Edges in TG that belong to the same $in\_edge(v)$ or $out\_edge(v)$ are called **brother edges**. If an edge is contracted during performance optimization, all its brother edges can no longer be contracted. The reason is that contracting brother edges means the tasks that once could be executed parallel in different cores now must be executed sequentially in the same core, and this may result in an increased schedule length. For example, contracting brother edges $e_{4,6}$ and $e_{4,7}$ in Fig. 5(b) makes v_6 and v_7 need to be conducted sequentially in the same core, but they can be computed once concurrently in different cores.

In addition, two edges belonging to the same path in the TVG should not be contracted simultaneously, and this avoids the over-optimization of the path length, which causes additional vulnerability against hardware Trojan attacks. The following example explains the reason, where the edges $e_{1,4}$ and $e_{4,7}$ belong to the same path in TVG (refer to Fig. 5(b)). Suppose that contracting either $e_{1,4}$ or $e_{4,7}$ will make the path length smaller than the performance constraint, and contracting $e_{1,4}$ and $e_{4,7}$ at the same time causes additional vulnerability.

Then, a weighted **edge contraction conflict graph** ($ECCG = (V_E, E_E)$) is constructed to represent whether every pair of edges in TVG can be contracted simultaneously. Each vertex in V_E represents an edge in TVG that can be contracted, and the weight of a vertex in V_E equals the weight of the corresponding edge in TVG. Two vertices 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 brother edges (with respect to the multicore parallel execution);
2) These two edges belong to the same path in TVG (to prevent the over-optimization of the path length).

The maximum weight independent set (MWIS) of $ECCG$ is calculated by the method proposed in [41], and the target is to find a set of edges with maximum weight that can be contracted simultaneously. Algorithm 1 shows details of the performance-constrained task clustering algorithm with the goal of minimizing the design's vulnerability. In the first step (*Lines 2-5*), TVG is constructed from TG, and the weights of all edges in TVG are evaluated. In the second step (*Lines 6-10*), the weighted $ECCG$ is built, and its MWIS is calculated. In the third step (*Lines 11-13*), the MWIS-selected edges in TG are contracted. These steps are iteratively repeated until the performance constraint is satisfied.

With the vulnerabilities of communications given in Fig. 5(c), an example of performance-constrained task clustering is shown in Fig. 6, where we are about to optimize the schedule length by 2 ut. The TVG consists of the nodes and edges with black color, and dashed lines are the contracted edges. $ECCG$ is given beneath the corresponding TVG, and the weight of contracting an edge is marked next to the node in $ECCG$. TVG and its corresponding $ECCG$ are constructed (see Fig. 6(a)), and its MWIS is $\{e_{5,8}, e_{6,9}, e_{7,10}\}$ which is contracted in the first iteration. Then, both TVG and $ECCG$ are updated as shown in Fig. 6(b),

Algorithm 1 Task clustering with performance constraint, $task\_cluster(TG, pc)$.

Input: task graph, TG;
performance constraint, pc.
Output: performance-constrained clustering result, TC.

```
while $TG.schedule\_length > pc$ do
    Construct $TVG$ from $TG$.
    for each $e$ in $TVG$ do
        Calculate $w(e)$;
    end for
    Construct $ECCG$ from $TVG$;
    for Each node $e$ in $ECCG$ do
        $ECCG.node\_weight(e) = w(e)$;
    end for
    Calculate $MWIS$ in $ECCG$;
    for each node $e$ in $MWIS$ do
        Contract the corresponding edge $e$ in $TG$;
    end for
end while
```

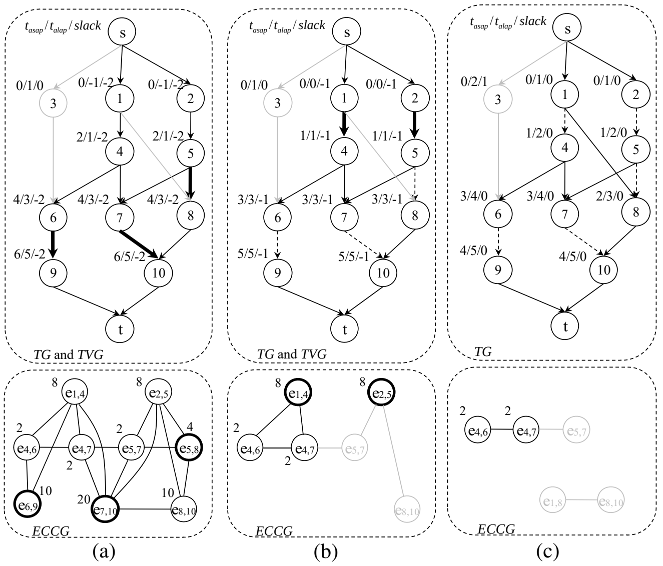


Fig. 6. Example of performance-constrained task clustering procedure. (a) TVG and its corresponding $ECCG$ before task clustering. (b) TVG and its corresponding $ECCG$ after 1st iteration of task clustering. (c) TVG and corresponding VCG after 2nd iteration of task clustering.

where $e_{5,7}$ and $e_{8,10}$ are not in $ECCG$ because their brother edges $e_{5,8}$ and $e_{7,10}$ are already contracted. The MWIS of the current $ECCG$ is $\{e_{1,4}, e_{2,5}\}$, and after contracting these edges, Fig. 6(c) yields the final clustering results, with the performance constraint satisfied.

4.2 Vendor Assignment with Core Minimization

The principle of vendor assignment is to iteratively cluster tasks into a number of v_c (vendor constraint) clusters, and assign each cluster with an IP vendor according to its core speed. Different from task clustering in the performance-constrained task clustering stage that violates isolation-with-diversity by clustering data-dependent tasks, clustering (also named as **cluster merging**) in vendor assignment follows security constraints.

The **vendor conflict graph** ($VCFG = (V_c, E_{cf})$) is constructed from the performance-constrained clustering results, and it represents whether two clusters must be assigned to different vendors. V_c is the set of all clusters from TG and TG', and a cluster is determined by the following two situations: 1) a task that is not connected by any contracted edge is regarded as a cluster; and 2) tasks that are connected to each other by contracted edges are in the same cluster, and the index of this cluster is decided by the minimum index of the tasks in this cluster. E_{cf} is the edge set in $VCFG$, and if two tasks must be assigned to different IP vendors due to security constraints, the two clusters that contain these two tasks will be connected in $VCFG$.

The **vendor compatible graph** ($VCPG = (V_c, E_{cp})$) is the complement graph of $VCFG$, and an edge in E_{cp} indicates that the connected clusters can be assigned to the same vendor. Fig. 7(a) gives examples of $VCFG$ and $VCPG$, which are constructed from the performance-constrained clustering result shown in Fig. 6(c). Because the edges in $VCPG$ are too numerous to demonstrate, we use dashed lines to represent the remaining edges that are connected to this cluster.

Then, a probability-based method is used to analyze the number of cores required. The accumulated probability of task concurrency is calculated, and it is denoted as *distribution graph* (DG). We assume that the probabilities of a task on all its possible scheduling results are the same [40], and the probability that v_i is executed in time t_j is denoted as $prob(v_i, t_j)$. The summation of the probabilities of all tasks in a cluster c for the time period t_j is denoted as $DG(c, t_j)$ and calculated as follows:

$$DG(c, t_j) = \sum_{v_i \in c} prob(v_i, t_j) \tag{5}$$

The maximum of all $DG(c, t_j)$, $\forall t_j \in [1, p_c]$ is denoted as $DG_{max}(c)$, which estimates the number of cores required for all tasks in cluster c. Fig. 7(b) presents the distribution graphs of all clusters, where the width of a task means the probability that this task will be conducted at the corresponding time period.

The number of cores required may be reduced by merging two clusters c_i and c_j, which is denoted as $Merge(c_i, c_j)$, and it can be calculated as follows:

$$Merge(c_i, c_j) = DG_{max}(c_i) + DG_{max}(c_j) - DG_{max}(c_i + c_j) \tag{6}$$

A larger $Merge(c_i, c_j)$ indicates a higher probability that tasks in c_i and c_j can share the same cores, and therefore, assigning these tasks to the same IP vendor reduces the number of cores. Examples of calculating $Merge(c_2, c_3)$ and $Merge(c_2, c_7)$ are shown in Fig. 7(b). $Merge(c_2, c_3) = 1 + 0.5 - 1.5 = 0$, which means that core reduction cannot be achieved by merging c_2 and c_3. $Merge(c_2, c_7) = 1 + 1 - 1 = 1$, indicating that merging c_2 and c_7 may reduce one IP core.

$Merge(c_i, c_j)$ is then set as the weight of edge (c_i, c_j) in $VCPG$. The edge with maximum weight is chosen, and the connected clusters are merged into one; this procedure continues until the number of clusters equals the number of vendors available. Because $VCPG$ with $O(n)$ nodes has nearly $O(n^2)$ edges, the maximum weight independent set of $VCPG$ is not introduced to determine the clusters to be merged due to its large time complexity.

An example of the cluster merging procedure is presented in Fig. 8, where the initial $VCFG$ and $VCPG$ are shown in Fig. 7(a), and the vendor constraint is 3. The maximum weight of all edges in $VCPG$ is 1, and c_2 and c_6 in Fig. 7(a) are merged into one cluster, named c_2. All edges that once connected to c_2 and c_6 in $VCFG$ now connect to c_2 in the updated $VCFG$,

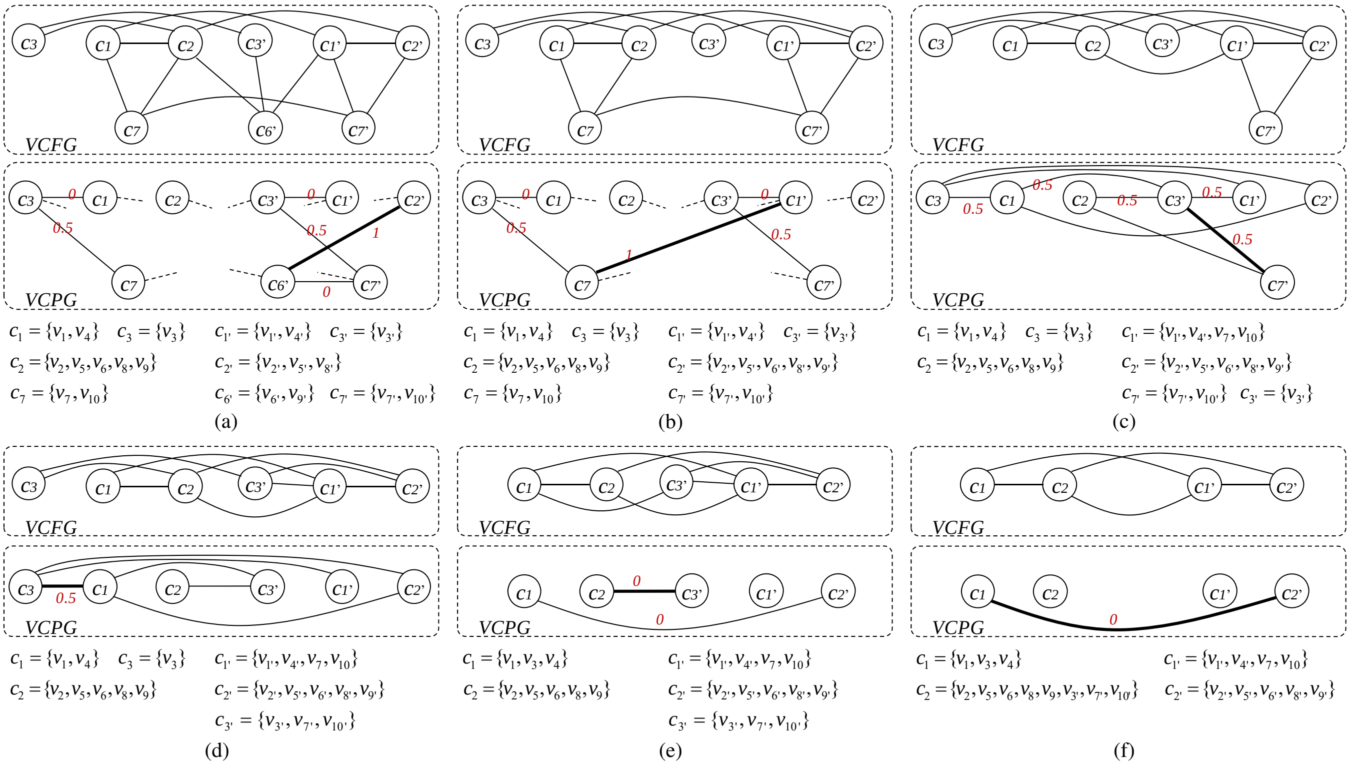

VCFG
VCPG
C1
C2
C3
C7
C1'
C2'
C3'
C6'
C7'
0
0.5
1
$c_1 = \{v_1, v_4\}$ $c_3 = \{v_3\}$ $c_{1'} = \{v_{1'}, v_{4'}\}$ $c_{3'} = \{v_{3'}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{8'}\}$
$c_7 = \{v_7, v_{10}\}$ $c_{6'} = \{v_{6'}, v_{9'}\}$ $c_{7'} = \{v_{7'}, v_{10'}\}$
(a)
$c_1 = \{v_1, v_4\}$ $c_3 = \{v_3\}$ $c_{1'} = \{v_{1'}, v_{4'}\}$ $c_{3'} = \{v_{3'}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{6'}, v_{8'}, v_{9'}\}$
$c_7 = \{v_7, v_{10}\}$ $c_{7'} = \{v_{7'}, v_{10'}\}$
(b)
$c_1 = \{v_1, v_4\}$ $c_3 = \{v_3\}$ $c_{1'} = \{v_{1'}, v_{4'}, v_7, v_{10}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{6'}, v_{8'}, v_{9'}\}$
$c_{7'} = \{v_{7'}, v_{10'}\}$ $c_{3'} = \{v_{3'}\}$
(c)
$c_1 = \{v_1, v_4\}$ $c_3 = \{v_3\}$ $c_{1'} = \{v_{1'}, v_{4'}, v_7, v_{10}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{6'}, v_{8'}, v_{9'}\}$
$c_{3'} = \{v_{3'}, v_{7'}, v_{10'}\}$
(d)
$c_1 = \{v_1, v_3, v_4\}$ $c_{1'} = \{v_{1'}, v_{4'}, v_7, v_{10}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{6'}, v_{8'}, v_{9'}\}$
$c_{3'} = \{v_{3'}, v_{7'}, v_{10'}\}$
(e)
$c_1 = \{v_1, v_3, v_4\}$ $c_{1'} = \{v_{1'}, v_{4'}, v_7, v_{10}\}$
$c_2 = \{v_2, v_5, v_6, v_8, v_9, v_{3'}, v_{7'}, v_{10'}\}$ $c_{2'} = \{v_{2'}, v_{5'}, v_{6'}, v_{8'}, v_{9'}\}$
(f)

Fig. 8. Example of cluster merging procedure. (a) $VCFG$ and $VCPG$ in 2nd iteration of cluster merging. (b) $VCFG$ and $VCPG$ in 3th iteration of cluster merging. (c) $VCFG$ and $VCPG$ in 4th iteration of cluster merging. (d) $VCFG$ and $VCPG$ in 5th iteration of cluster merging. (e) $VCFG$ and $VCPG$ in 6th iteration of cluster merging. (f) $VCFG$ and $VCPG$ in 7th iteration of cluster merging.

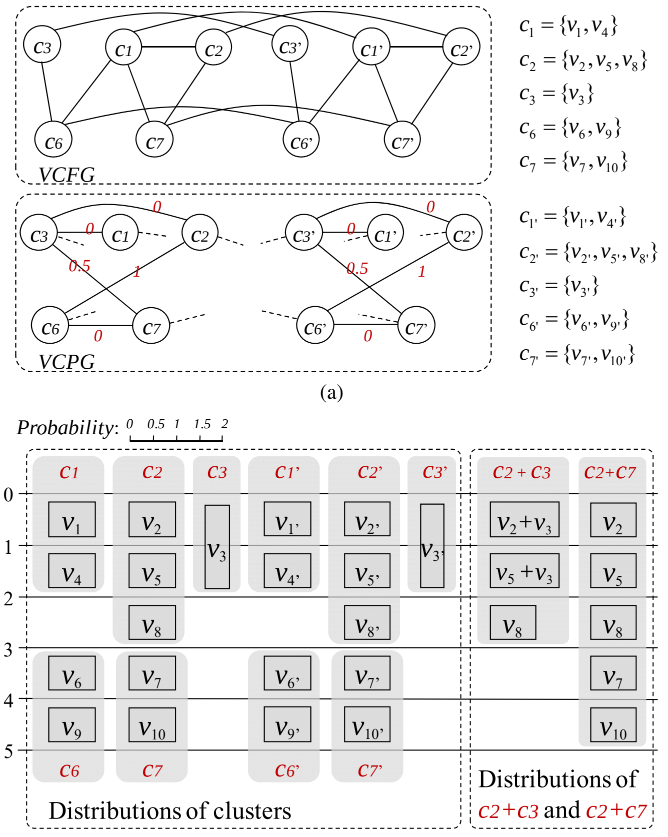
and the weights of edges that connect to c_2 in $VCPG$ are also updated. Fig. 8(a) shows $VCFG$ and $VCPG$ after the 1st iteration of merging clusters. Then, c'_2 and c'_6 are merged in the 2nd iteration because the weight of their connecting edge is 1. Fig. 8(b) shows the updated $VCFG$ and $VCPG$. With the corresponding $VCFGs$ and $VCPGs$ shown in Fig. 8(b)-Fig. 8(f), the pairs of clusters (c'_1, c_7), (c'_3, c'_7), (c_1, c_3), (c_2, c'_3) and (c_1, c'_2) are iteratively merged. This procedure terminates when the number of clusters equals the vendor constraint, and Fig. 9 gives the final results, where the total estimated number of cores is 5.5.

Then, the number of timing-critical tasks in each cluster is countered, and the clusters containing more timing-critical tasks are assigned to the vendor with faster core speeds. Some timing-critical paths may be over-optimized because all tasks are treated with the lowest core speed in the performance-constrained task clustering stage, and we need to adjust the vendor assignment to meet more security constraints. The slacks of tasks are updated with the assigned core speeds, and every intra-core communication is checked in descending order of vulnerabilities $vul(e)$ to determine whether this communication can be reassigned with security constraints. An intra-core communication (e) can be reassigned to inter-core communication to satisfy security constraints only when all tasks in the paths that pass through e have slack times no smaller than $dly(e)$, and one of its connected tasks will be assigned to the IP vendor with the least core increment.

Algorithm 2 describes the proposed vendor assignment algorithm. First (*Lines 1-2*), the number of cores required by each cluster is estimated by $DG_{max}(c)$, and $VCFG$ and weighted $VCPG$ are constructed. Second (*Lines 3-6*), the maximum weight edge in $VCPG$ is chosen, and the connected clusters are merged into one

Algorithm 2 Vendor-assignment with core minimization, $vendor\_assign(TC, TC', vc, pc)$.

```
Input: performance-constrained clustering results of task
       graph and duplicated task graph, $TC$, $TC'$.
       performance and vendor constraints, $pc$, $vc$.
Output: vendor assignment, $VA$.
 1: Calculate $DG_{max}(c)$ for each cluster $c$.
 2: Construct $VCFG$ and weighted $VCPG$;
 3: while $VCPG.node\_num > vc$ do
 4:     Choose the edge $e_{max} = (c_i, c_j)$ with the maximum weight
        in $VCPG$, and merge $c_i$ and $c_j$ into one cluster.
 5:     Update $VCFG$ and weighted $VCPG$,
 6: end while
 7: while Not all clusters are assigned with IP vendors do
 8:     assign $c_i$ to vendor $vendor_j$, where $c_i$ is an unassigned
        cluster with the most timing critical tasks and $vendor_j$ is
        the available vendor with the fastest core speed.
 9: end while
10: Update the distribution graphs with the determined task
    execution times and communication delays;
11: $E_{cv}$ is the set consists of all intra-core communications;
12: while $E_{cv} \neq \emptyset$ do
13:     Find $e \in E_{cv}$ with the largest $vul(e)$;
14:     if Assigning $e$ with security constraints still meets $pc$ then
15:         Assign either $source(e)$ or $target(e)$ with another vendor;
16:         Update the distribution graphs;
17:     end if
18:     Remove $e$ from $E_{cv}$;
19: end while
```


C3
C1
C2
C3'
C1'
C2'
C6
C7
C6'
C7'
VCFG
$c_1 = \{v_1, v_4\}$
$c_2 = \{v_2, v_5, v_8\}$
$c_3 = \{v_3\}$
$c_6 = \{v_6, v_9\}$
$c_7 = \{v_7, v_{10}\}$
0
0.5
1
VCPG
$c_{1'} = \{v_{1'}, v_{4'}\}$
$c_{2'} = \{v_{2'}, v_{5'}, v_{8'}\}$
$c_{3'} = \{v_{3'}\}$
$c_{6'} = \{v_{6'}, v_{9'}\}$
$c_{7'} = \{v_{7'}, v_{10'}\}$
Probability: 0 0.5 1 1.5 2
C1 C2 C3 C1' C2' C3' C2 + C3 C2+C7
0 1 2 3 4 5
v_1 v_4 v_2 v_5 v_8 v_3 $v_{1'}$ $v_{4'}$ $v_{2'}$ $v_{5'}$ $v_{8'}$ $v_{3'}$
v_2+v_3 v_5+v_3 v_8
v_6 v_9 v_7 v_{10} $v_{6'}$ $v_{9'}$ $v_{7'}$ $v_{10'}$
C6 C7 C6' C7'
Distributions of clusters
Distributions of $c2+c3$ and $c2+c7$

(a)

(b)

Fig. 7. Example of evaluating vendor compatible graph. (a) $VCFG$ and $VCPG$ derived from task clustering results. (b) Distributions of clusters.

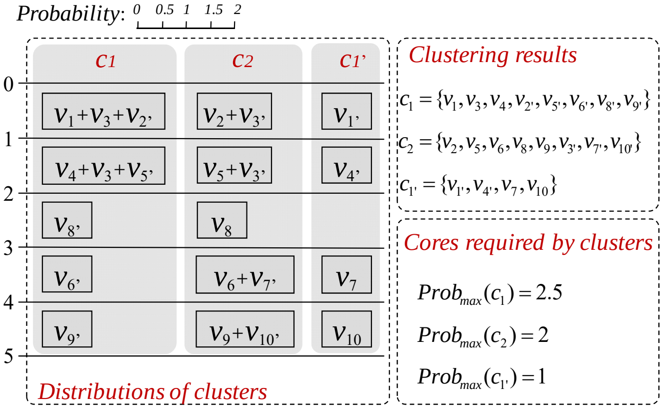


Fig. 9. Cluster merging results under vendor constraint.

cluster. Both $VCFG$ and weighted $VCPG$ are then updated, and this procedure continues until the number of clusters equals the number of IP vendors available. Third (*Lines 7-10*), the cluster containing more timing-critical tasks is assigned to the IP vendor with faster core speed. Finally (*Lines 11-19*), the unprotected edges are checked in descending order of vulnerabilities, and the vendor assignments are adjusted under performance constraints to further reduce the design's vulnerability.

4.3 Procedure of the Proposed Task Scheduling Method

With all security constraints satisfied, the number of IP vendors is always equal to the number of nodes in the maximum clique (denoted as *maximum clique size*) of $VCFG$. However, performance-constrained task clustering and vendor assignment may potentially increase the number of vendors required, and we must check every contracted edge if the resulting maximum clique size exceeds the vendor constraint. Computing the maximum clique size of a graph is NP-complete, and an efficient heuristic approach [22] is introduced. Each time after determining a contracted edge, the impact on the maximum clique size of the corresponding $VCFG$ is evaluated, and the edge is not contracted if the vendor constraint is violated. Instead, the algorithm chooses the second-best solutions.

Algorithm 3 Security-aware task scheduling with performance constraints, $task\_schedule(TG, pc, vc)$.

```
Input: task graph, $TG$
       performance and vendor constraints, $pc$, $vc$.
Output: scheduling results, $TS$.
1: $TC = task\_cluster(TG, pc)$;
2: $TC' = task\_cluster(TG', pc)$, where $TG'$ is the duplicate of $TG$;
3: $VA = vendor\_assign(TC, TC', vc, pc)$;
4: for each vendor $vendor_i$ do
5:     $V_{vendor_i}$ is the set of all tasks assigned to $vendor_i$;
6:     $FDS(V_{vendor_i}, pc)$;
7: end for
```

After vendor assignment, tasks with the same IP vendor are scheduled together using the force-directed scheduling (FDS) method [40]. FDS schedules tasks evenly across each time period, requiring only a small number of cores. Algorithm 3 gives the whole procedure of the proposed task scheduling algorithm. Tasks in the task graph and duplicated task graph are clustered under the performance constraint p_c (*Lines 1-2*) and then assigned to IP vendors with a minimized number of cores required (*Line 3*). Finally, tasks in each IP vendor are scheduled concurrently by the FDS (*Lines 4-7*).

Our proposed methods can also be easily adopted in other actual scenarios. In the first scenario, the number of vendors available might be less than the maximum clique size of the corresponding $VCFG$. In this situation, a vendor-constrained task clustering method [23] can be introduced before vendor assignment, so that the vendor constraint can be satisfied with a minimized number of contracted edges. In the second scenario, application-specific IPs might not be able to support every task. For each type of IP, we first build its $VCFG$ and $VCPG$ according to the clustering results of tasks that belong to this type, and then perform the vendor assignment along with task scheduling individually.

4.4 Time Complexity Analysis

The time complexity of the proposed method is analyzed as follows, and the input task graph has n nodes and m edges.

In each iteration of the performance-constrained task clustering stage, constructing $ECCG$ from TVG requires $O(m^2)$, and finding the MWIS in $ECCG$ also requires $O(m^2)$ [41]. Only a constant number of iterations are conducted before reaching the performance constraint, and finding all contracted edges to meet the performance constraint requires $O(m^2)$. In addition, each time before contracting an edge, updating $VCFG$ and evaluating its impact on the maximum clique size requires $O(n^2)$, and only a limited number of edges are contracted, making its computational cost remains at $O(n^2)$. The total time complexity of performance-constrained task clustering is $O(m^2)$ (because $O(n) \leq O(m)$).

In the vendor assignment and task scheduling stage, constructing $VCFG$ and $VCPG$ requires $O(n^2)$. In each iteration of merging clusters, $O(m)$ is required to estimate the maximum clique size, and $O(n)$ is required to update both $VCFG$ and $VCPG$. Vender assignment requires $O(n)$ iterations of merging clusters, and its time complexity is $O(mn)$. Performing the force-directed scheduling method to schedule all tasks requires $O(n^2)$, and the total time complexity of the vendor assignment and task scheduling stage is $O(mn)$.

The sum of $O(m^2)$ and $O(mn)$ is $O(m^2)$, which is the total time complexity of the proposed method.

5 Experimental Results

5.1 Experimental Setups

All experiments were implemented in C on a Linux Workstation with an E5 2.6-GHz CPU and 32-GB RAM. We tested eight benchmarks from two sources[1]: task graphs modeled from real application programs, including robot control (robot), sparse matrix solver (sparse), and SPEC fpppp (fpppp); and randomly generated task graphs (rnc500, rnc1000, rnc2000, rnc3000 and rnc5000). To simplify the experiments, all intra-core communication delays were ignored, and we set the step of core speed differences equal to 5% of the fastest core speed.

TABLE 1
Details of Benchmarks

task graph	n	m	Para.	ACC (ut)	$maxClique$
robot	88	131	4.4	28.2	3
sparse	96	67	16.0	20.2	3
fpppp	334	1145	6.7	21.3	3
rnc500	500	1910	27.7	10.6	3
rnc1000	1000	3005	60.2	7.8	3
rnc2000	2000	3930	151.9	10.6	3
rnc3000	3000	39034	34.2	13.0	4
rnc5000	5000	55432	90.8	11.0	4

Table 1 lists the details of these benchmarks. Columns n and m give the numbers of tasks and communications in each task graph, respectively. Column Para. shows the parallelism of each task graph, which is the ratio of the total processing time of all tasks to the ASAP schedule length (without communication delays). Column ACC shows the average computational cost of each task. Considering that the maximum clique sizes of most task graphs modeled from real application programs are no larger than 4 [22], the maximum clique sizes (see column *maxClique*) of all randomly generated TGs in Table 1 are 3 or 4, and the IP vendor constraint is set to be the maximum clique size of the benchmark.

Our proposed method is then compared with three other methods to demonstrate its effectiveness. The first approach is the "*cluster-based approach*" (**C-B** for short) [22]. This approach optimizes the schedule length by placing critical tasks on a single core, and then colors the performance-driven schedule to fulfill security constraints. The second approach is the "*min-cut-based approach*" (**MC-B** for short) [23]. This approach boosts performance by iteratively contracting the edges selected by the max-flow min-cut algorithm, then assigns tasks to the IP vendor with the traditional graph coloring method, and finally schedules tasks with the force-directed scheduling-based method. The third approach is the "*PSO-based approach*" (**PSO-B** for short) [18], which uses a PSO-based method to find a resource-optimized solution with all security constraints satisfied. In the experiment, the accumulated vulnerabilities of unprotected communications are also added to the cost function of PSO-B, so that PSO-B can be used to find solutions with tight performance constraints.

5.2 Performance-Constrained Task Scheduling Results

The performance-constrained task scheduling results are shown in Tables 2 and 3. The communication-to-computation ratio (CCR) is the ratio of the inter-core communication delay to the computational cost of the task, and two $CCRs$ (0.5 and 1.0) are tested. The performance constraint is set to pc=0.8SL, where SL is the ASAP schedule length with all security constraints satisfied. In this set of experiments, the communications have different vulnerabilities to hardware Trojan insertions, and to simplify our experiments, we set the vulnerability of e_{ij} in TG or TG' as follows:

$$vul(e_{ij}) = \frac{2 * dist(e_{ij}) * pte}{SL * m} \tag{7}$$

where $dist(e_{ij})$ is the distance from s to v_i. pte is the *possible Trojan entries* [18], and we set pte equal to 3 for all benchmarks.

Table 2 compares the task scheduling results of these methods on three real application graphs. The results show that our proposed method obtained the lowest vul_s for all benchmarks. MC-B optimized the design's vulnerability by reducing the number of unprotected communications but ignored the vulnerability variation of communications. MC-B might also choose brother edges to contract, resulting in a vul_s larger than that of our proposed method. PSO-B explores the design space to find the near-optimal solution, and the quality of its output also depends on the iteration of the algorithm. Furthermore, MC-B forgot to optimize the number of cores in the vendor assignment stage, and it required more cores than our method. Regarding the method C-B, it ignored both the design's vulnerability and the number of cores required during task scheduling, and both the vul_s and number of cores are the largest among the compared methods. PSO-B outputs the schedules with low vul_s and small numbers of cores required, but its CPU runtimes are much larger.

The comparison results of five randomly generated task graphs are shown in Table 3. The number of edges in these task graphs is much larger than the edges of *robot*, *sparse*, and *fpppp*, and therefore, the vulnerability of each communication is relatively much smaller (refer to Equ. (7)). The comparison results show that our proposed method also obtained the best results in reducing both the vul_s and the number of cores.

In terms of algorithm runtime, all of the algorithms can produce solutions within several minutes for the benchmarks modeled from real applications (see Table 2). For benchmarks that contain many nodes and edges, such as *rnc5000* which has 5000 nodes and 55432 edges (see Table 3), our proposed method can output a solution within approximately 10 minutes, which indicates that our proposed method is applicable for most benchmarks in real practice.

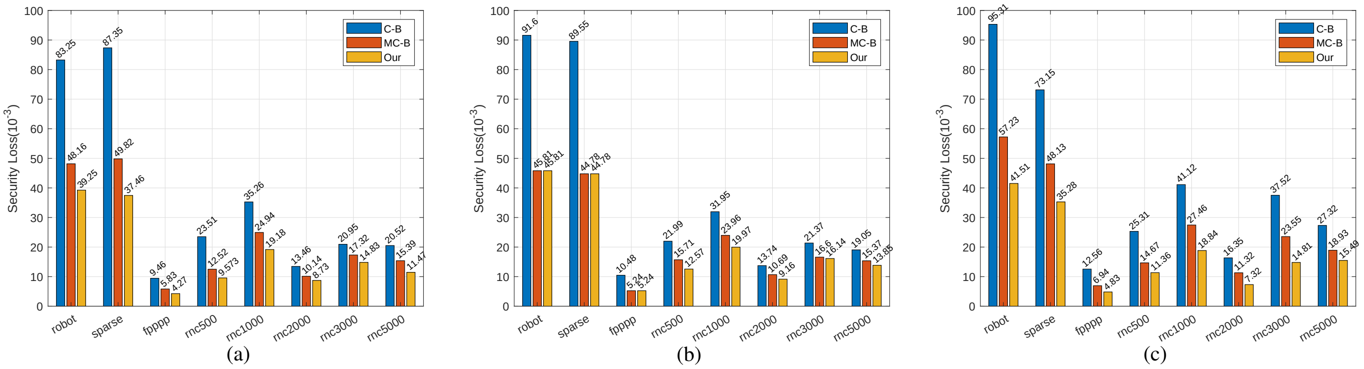
[1] https://www.kasahara.cs.waseda.ac.jp/schedule/index.html.

TABLE 2
Performance-Constrained Task Scheduling Results on Real Application Graphs.

task graph	CCR	SL (ut)	pc (ut)	vul_s (10^{-3})				Num. of cores				average runtime (s)			
				C-B	MC-B	PSO-B	Our	C-B	MC-B	PSO-B	Our	C-B	MC-B	PSO-B	Our
robot	0.5	839	671	353.68	206.57	163.82	155.35	14	12	11	11	3.9	17.5	221.6	18.7
	1.0	1114	892	274.91	185.62	158.19	147.83	14	12	10	10	3.9	17.6	227.5	18.9
sparse	0.5	179	143	146.75	93.48	81.48	76.07	21	18	16	16	5.3	41.2	285.6	33.8
	1.0	236	189	157.48	98.47	86.53	83.59	19	18	16	15	5.5	42.3	288.3	34.5
fpppp	0.5	1590	1272	15.23	6.37	5.15	4.52	13	12	10	10	7.5	50.2	369.2	47.6
	1.0	2119	1695	11.49	6.85	5.39	4.97	12	11	10	10	8.3	53.5	372.7	49.8
avg.	0.5			171.89	102.14	83.48	78.65								
	1.0			147.96	96.98	83.37	78.80								

TABLE 3
Performance-Constrained Task Scheduling Results on Randomly Generated Task Graphs.

task graph	CCR	SL (ut)	pc (ut)	vul_s (10^{-3})				Num. of cores				average runtime (s)			
				C-B	MC-B	PSO-B	Our	C-B	MC-B	PSO-B	Our	C-B	MC-B	PSO-B	Our
rnc500	0.5	280	224	25.13	15.70	13.25	12.46	68	65	58	58	18.9	113.6	823.5	95.4
	1.0	373	300	20.41	16.35	13.48	12.95	67	63	58	56	20.3	121.3	827.1	98.7
rnc1000	0.5	190	152	47.92	33.94	27.82	26.71	95	88	79	78	38.8	259.3	3271.3	183.5
	1.0	254	203	48.73	31.95	25.78	23.69	87	81	76	74	42.1	273.3	3282.9	189.7
rnc2000	0.5	199	159	20.61	16.53	12.76	12.37	217	184	170	167	57.5	715.7	8661.3	553.6
	1.0	268	214	21.47	14.51	11.34	10.62	206	180	168	164	59.1	748.2	8678.2	572.9
rnc3000	0.5	1336	1069	56.41	43.27	36.73	35.58	73	67	64	62	182.5	3582.6	33425.6	3014.9
	1.0	1779	1423	53.49	37.74	32.05	31.18	68	62	56	56	192.2	3715.3	33581.4	3253.4
rnc5000	0.5	850	680	48.13	43.17	35.21	34.52	142	132	125	122	384.9	9004.5	91423.9	6764.2
	1.0	1146	917	54.84	47.25	36.89	36.55	137	125	118	115	395.4	9528.3	91658.5	6816.1
avg.	0.5			39.64	30.52	25.15	24.33								
	1.0			39.79	29.56	23.91	23.01								



(a) (b) (c)

Fig. 10. Comparison of design's vulnerabilities against hardware Trojan attacks. (a) Design's vulnerability evaluations with vulnerabilities *randomly set*. (b) Design's vulnerability evaluations with vulnerabilities *equally set*. (c) Design's vulnerability evaluations with vulnerabilities *linearly set*.

5.3 Design's Vulnerability Optimization Results

Then, the effectiveness of our method in optimizing the design's vulnerability is tested. CCR is set to CCR=1.0 for all benchmarks, and the performance constraint is set to $pc = 0.9 * SL$. Fig. 10 shows the comparative results of design's vulnerabilities among C-B, MC-B, and our methods, with the following three sets of vulnerability configurations:

1) **Randomly Set**: the vulnerabilities of communications are randomly set with the value among [0, $\frac{2*pte}{m}$];
2) **Equally Set**: the vulnerabilities of all communications are the same, with the value of $\frac{pte}{m}$;
3) **Linearly Set**: the communication closer to the source has higher vulnerability, and the vulnerability of e_{ij} is set as

$$vul(e_{ij}) = \frac{2*(SL-dist(e_{ij}))*pte}{SL*m};$$

The pte is set to 3 in this set of experiments. Because both C-B and MC-B ignore the vulnerability variation, they produced the same task scheduling result with different vulnerability configurations. Our proposed method seeks the task schedule with the lowest design's vulnerability, and produces different task scheduling results for different vulnerability configurations.

Figs. 10(a), 10(b), and 10(c) show the design's vulnerability with the vulnerabilities of communications *randomly set*, *equally set*, and *linearly set*, respectively. For each tested benchmark, our proposed method outperforms both C-B and MC-B with different vulnerability configurations, and detailed comparisons are given as follows:

TABLE 4
Comparisons of Cores Required.

task graph	*SL* (*ut*)	Loose vendor constraints					Tight vendor constraints								
		vc	Num. of cores				*vc*	Num. of cores				Proportion of unprotected edges			
			C-B	MC-B	PSO-B	Our		C-B	MC-B	PSO-B	Our	C-B	MC-B	PSO-B	Our
robot	1114	3	10	9	8	8	2	10	8	7	7	2.29%	1.53%	1.53%	1.53%
sparse	236	3	14	12	12	12	2	14	11	10	10	8.96%	5.97%	5.97%	5.97%
fpppp	2119	3	10	9	8	8	2	9	7	6	6	1.57%	0.96%	0.96%	0.96%
rnc500	373	3	47	45	40	39	2	42	38	36	35	4.50%	3.72%	3.72%	3.72%
rnc1000	254	3	68	63	58	56	2	58	51	48	47	1.73%	1.30%	1.30%	1.30%
rnc2000	268	3	156	148	132	130	2	139	127	118	116	0.74%	0.46%	0.46%	0.46%
rnc3000	1779	4	56	51	48	48	2	52	48	45	45	58.52%	53.15%	53.15%	53.15%
rnc5000	1146	4	106	98	90	88	2	97	91	84	83	70.24%	67.89%	67.89%	67.89%

1) When the vulnerabilities are *randomly set*, the averaged design's vulnerabilities of C-B and MC-B are 36.72*10^{-3}, and 23.02*10^{-3}, respectively, and our proposed method obtains the lowest design's vulnerability, which is only 18.11*10^{-3}.
2) If the vulnerabilities are *equally set*, our method and MC-B obtain nearly equivalent results because MC-B treats each communication equally when optimizing system performance. The averaged design's vulnerabilities of C-B, MC-B and our methods are 37.47*10^{-3}, 22.27*10^{-3}, and 20.94*10^{-3}, respectively.
3) If the vulnerabilities are *linearly set*, our proposed method still demonstrate its advantage by obtaining the lowest design's vulnerability, which is 18.68*10^{-3}. However, the averaged design's vulnerabilities of C-B and MC-B are 41.08*10^{-3} and 26.03*10^{-3}, respectively.
4) For some benchmarks (*robot* and *sparse*), their design's vulnerabilities are relatively high because the vulnerabilities of communications are also determined by the number of edges in task graphs. These task graphs have much fewer edges, making the values of vulnerabilities in *robot* and *sparse* much larger than that in the other task graphs.

5.4 Comparison of Cores Required

To demonstrate the effectiveness of our method in reducing the number of cores required, the performance constraint is set to *SL* so that the performance-constrained task clustering stage can be skipped. Table 4 shows the numbers of cores required, where two sets of vendor constraints are tested. The loose vendor constraints are first set for all benchmarks, and the vendor constraint is set to be the maximum clique size of the corresponding TG. C-B ignores core optimization, MC-B only minimizes the number of cores when scheduling tasks, and PSO-B explores the design space with a limited number of iterations. In our proposed method, reducing the number of cores is considered in both vendor assignment and task scheduling, which enlarges the optimization space for saving cores. The results indicate that our proposed method needs the fewest numbers of cores in all benchmarks, and MC-B, PSO-B and our proposed method reduce the numbers of cores by 8.44%, 16.08% and 17.11%, respectively, compared to the C-B method.

Then, the tight vendor constraints are set because there might not be sufficient vendors for some specific IPs, and the vendor constraints of all benchmarks are set to 2. To meet the tight vendor constraints, the vendor-constrained task clustering method proposed in [23] is used to remove some security constraints from communications and allow the adjacent tasks to be executed on the cores from the same IP vendor. Therefore, some communications become unprotected, and the proportions of the number of unprotected edges to the number of all edges are given in the last column of Table 4. Because MC-B, PSO-B and our method use the same vendor-constrained task clustering, the proportions of their unprotected edges are the same. C-B ignored reducing the number of unprotected edges to meet vendor constraints, and its numbers of unprotected edges are larger. For the benchmarks *rnc3000* and *rnc5000*, whose task graphs contain many 4-cliques, many edges become unprotected to satisfy the vendor constraint. The number of cores required shows that our proposed method also obtained the fewest cores among these four methods, and the average numbers of cores required by MC-B, PSO-B and our proposed method are 13.47%, 20.68% and 21.49% less than that of C-B, respectively.

6 Conclusions

In this study, a security-driven task scheduling method is proposed to reduce the performance and area overheads of implementing security constraints in the design process, and the desired performance is set as a constraint. The communications between data-dependent tasks are treated with different vulnerabilities against hardware Trojan attacks, and a maximum weight independent set-based task clustering method is proposed to reduce the schedule length while maintaining a high security level. In addition, the numbers of cores required are optimized in both vendor assignment and task scheduling stages by assigning tasks that can share most cores to the same vendor and scheduling them evenly in each time period, which enlarges the optimization space for reducing cores. Experimental results demonstrate that our proposed method obtains the highest system security and the fewest cores among all compared methods.

References

[1] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware Trojans: lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 1, pp. 6-29, May 2016.

[2] 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.

[3] S. Bhunia, M.S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229-1247, Aug. 2014.

[4] M. Hussain, A. Malekpour, H. Guo, and S. Parameswaran, “EETD: an energy efficient design for runtime hardware Trojan detection in untrusted network-on-chip,” *Proc. IEEE Computer Society Annual Symposium on VLSI*, pp. 345-350, 2018.

[5] A. Malekpour, R. Ragel, A. Ignjatovic, and S. Parameswaran, “DosGuard: protecting pipelined MPSoCs against hardware Trojan based DoS attacks,” *Proc. International Conference on Applications-specific Systems, Architectures and Processors*, pp. 45-52, 2017

[6] F. Kounelis, N. Sklavos, and P. Kitsos, “Run-time effect by inserting hardware Trojans in combinational circuits,” *Euromicro Conference on Digital System Design*, pp. 287-290, 2017.

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

[8] B. Bilgic and S. Ozev, “Guaranteed activation of capacitive Trojan triggers during post production test via supply pulsing,” *Proc. Design, Automation & Test in Europe Conference*, pp. 993-998, 2022.

[9] D. Deng, Y. Wang, and Y. Guo, “Novel design strategy toward A2 Trojan detection based on built-in acceleration structure,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4496-4509, Feb. 2020.

[10] Y. Huang, S. Bhunia, and P. Mishra, “Scalable test generation for Trojan detection using side channel analysis,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 11, pp. 2746-2760, Nov. 2018.

[11] Y. Hou, H. He, K. Shamsi, Y. Jin, D. Wu, H. Wu, “R2D2: runtime reassurance and detection of A2 Trojan,” *Proc. International Symposium on Hardware-Oriented Security and Trust*, pp. 195-200, 2018.

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

[13] T. Reece and W. H. Robinson, “Detection of hardware Trojan in third-party intellectual property using untrusted modules,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 357-366, Jul. 2015.

[14] 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.

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

[16] M. Shatta, I. adly, H. Amer, G. Alkady, R. Daoud, S. Hamed, and S. Hatem, “FPGA-based architectures to recover from hardware Trojan horses, single event upsets and hard failures,” *Proc. International Conference on Microelectronics*, pp. 1-4, 2020.

[17] 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.

[18] S. Rajmohan, N. Ramasubramanian, and N. Naganathan, “Hybrid evolutionary design space exploration algorithm with defence against third party IP vulnerabilities,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2602-2614, May 2022.

[19] 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, May 2015.

[20] Y. Sun, G. Jiang, S.-K. Lam, and F. Ning, “Designing energy-efficient MPSoC with untrustworthy 3PIP cores,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 51-63, Jan. 2020.

[21] X. Cui, X. Zhang, H. Yan, L. Zhang, K. Cheng, Y. Wu, and K. Wu, “Toward building and optimizing trustworthy systems using untrusted components: a graph-theoretic perspective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1386-1399, Oct. 2020.

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

[23] N. Wang, S. Chen, J. Ni, X. Ling, and Y. Zhu, “Security-aware task scheduling using untrusted components in high-level synthesis,” *IEEE Access*, vol. 6, pp. 15663-15678, Jan. 2018.

[24] N. Wang, M. Yao, D. Jiang, S. Chen, and Y. Zhu, “Security-driven task scheduling for multiprocessor system-on-chips with performance constraints,” *Proc. IEEE Computer Society Annual Symposium on VLSI*, pp. 545-550, 2018.

[25] H. Salmani and M. Tehranipoor, “Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level,” *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 190-195, 2013.

[26] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, H. Li, “An overview of hardware security and trust: threats, countermeasures, and design tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1010-1038, Jun. 2021.

[27] D. Meng, R. Hou, G. Shi, B. Tu, A. Yu, Z. Zhu, X. Jia, Y. Wen, and Y. Yang, “Built-in security computer: deploying security-first architecture using active security processor,” *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1571-1583, Nov. 2020.

[28] N. Hu, M. Ye, and S. Wei, “Surviving information leakage hardware Trojan attacks using hardware isolation,” *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 2, pp. 253-261, Apr. 2019.

[29] H. Kim, S. Hong, B. Preneel, and I. Verbauwhede, “STBC: Side channel attack tolerant balanced circuit with reduced propagation delay, *Proc. IEEE Computer Society Annual Symposium on VLSI*, pp. 74-79, 2017.

[30] A. Sengupta and M.Rathor, “IP core steganography for protecting DSP kernels used in CE systems,” *IEEE Transactions on Consumer Electronics*, vol. 65, no. 4, pp. 506-515, Nov. 2019.

[31] S. Yu, C. Gu, W. Liu, and M. O’Neill “Deep learning-based hardware Trojan detection with block-based netlist information extraction,” *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1837-1853, Oct. 2022.

[32] X. Zhang and M. Tehranipoor, “Case study: detecting hardware Trojans in third-party deigital IP cores,” *International Symposium on Hardware-Oritented Security and Trust*, pp. 67-70, 2011.

[33] M. Tehranipoor and F. Koushanfar, “A survey of hardware Trojan taxonomy and detection,” *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10-25, Jan. 2010.

[34] J. Cruz, P. Slpsk, P. Gaikwad, and S. Bhunia, “TVF: a metric for quantifying vulnerability against hardware Trojan attacks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 7, pp. 969-979, Jul. 2026.

[35] Y. Dou, C. Gu, C. Wang, W. Liu, and F. Lombardi, “Security and approximation: vulnerabilities in approximation-aware testing,” *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 265-271, Jan. 2023.

[36] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M.S. Hsiao, J. Plusquellic, M. Tehranipoor, “Protection against hardware Trojan attacks: towards a comprehensive solution,” *IEEE Design & Test*, vol. 30, no. 3, pp. 6-17, Jun. 2013.

[37] R. S. Chakraborty, S. Pagliarini, J. Mathew, S. R. Rajendran, and M. N. Devi, “A flexible online checking technique to enhance hardware Trojan horse detectability by reliability analysis,” *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 260-270, Apr. 2017.

[38] 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, 2011.

[39] N. Veeranna and B.C. Schafer, “Hardware Trojan detection in behavioral intellectual properties (IP’s) using property checking techniques,” *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 576-585, Oct. 2017.

[40] P.G. Paulin and J.P. Knight, “Force-directed scheduling for the behavioral synthesis of ASIC’s,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661-679, Jun. 1989.

[41] L. Chang, W. Li, and W. Zhang, “Computing a near-maximum independent set in linear time by reducing-peeling,” *Proc. ACM International Conference on Management of Data*, pp. 1181-1196, 2017.

PLACE
PHOTO
HERE

Nan Wang received a B.E. degree in computer science from Nanjing University, Nanjing, China, in 2009, and M.S and Ph.D. degrees from the Graduate School of IPS, Waseda University, Japan, in 2011, and 2014, respectively. He is currently an associate professor in School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China. His current research interests include VLSI design automation, low power design techniques, network-on-chip and reconfigurable architectures. Dr. Wang is a member of IEEE and IEICE.

PLACE
PHOTO
HERE

Songping Liu received the B.E. degree in information engineering from East China University of Science and Technology, Shanghai, China, in 2021. He is currently working toward the M.S. degree in electronic information from East China University of Science and Technology, Shanghai, China. His current research interests include hardware Trojan detection and hardware security.

PLACE
PHOTO
HERE

Song Chen received a B.S. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2000, and M.S. and Ph.D. degrees in computer science from Tsinghua University, Beijing, China, in 2003 and 2005, respectively. From August 2005 to March 2009, he served as a research associate at the Graduate School of IPS, Waseda University, Japan, and from April 2009 to August 2012, he served the same university as an assistant professor. He is currently an associate professor at the Dept. of Electronic Sci. and Tech., University of Science and Technology of China (USTC). His current research interests include several aspects of VLSI physical design automation, on-chip communication system, and computer-aided design for emerging technologies. Dr. Chen is a member of IEEE and IEICE.

PLACE
PHOTO
HERE

Yu Zhu received the B.S. and Ph.D. degrees in electronics and communication engineering from Nanjing University of Science and Technology, Nanjing, China, in 1995 and 1999 respectively. She is currently a professor of electronics and communication engineering in East China University of Science and Technology, Shanghai, P.R. China. In 2005, she was a research scholar in UIUC. Her current research interests include computer design automation, pattern recognition and machine learning.

Authors' Response

We greatly appreciate the Editor's and the reviewers' insightful and scrupulous reviews of our paper. The comments provided have contributed substantially to the improvement of our manuscript. In what follows, we present the detailed explanations of how the manuscript has been revised to respond to the comments of the associate editor and the reviewers. In the previous pages, the sentences colored blue, red, and cyan are the modified parts.

Reviewer 1

Reviewer 2

Comment 1

The work has some similarity and overlap with the authors' prior work (ref. [23], [24]). But the authors simply mentioned prior work ignore the communication paths..." A better justification with proper comparison and novelty over prior work is needed.

Responses

Thanks very much for your valuable comments. Compared to our prior work, this work has made some improvements in the following two aspects.

1) Our prior work treat that all the communications and tasks are equally to the hardware Trojan insertions, and security constraints are removed from the communications with largest delay so as to meet the desired system performance. However, the vulnerabilities of different parts of a circuit can differ by $\times 10^3$ times [25], and removing security constraints away from the communications with larger vulnerabilities might cause significant security losses. In this work, the tasks and communications are treated with different vulnerabilities, and the proposed task scheduling algorithms maximize the security under the performance constraints.
2) Our prior work forgot to optimize the chip area in the task scheduling process, and the resulting chip area might be large. However, area is also one of the critical issues in the circuit design, and therefore, this work jointly optimizes performance and security along with area.

These explanations that show the novelties over our prior work is also presented in the manuscript, in the lower right of page 1 with blue color, and these explanations can also be found in the following.

However, their work ignore that parts of a design are much more vulnerable to hardware Trojan attacks [25], and removing security constraints from the part of a circuit that are more susceptible to Trojan insertion may yield significant security losses [21]. Furthermore, these work only optimize the system performance in the context of security constraints, which might incurs a significant area overhead, but chip area is also one of the critical issues towards trusted design; therefore, performance and security along with chip area should be jointly considered for MPSoC design, especially for heterogeneous MPSoCs built from 3PIP cores which are untrustworthy.

In this study, we focus on the design of MPSoCs though security-driven task scheduling with performance constraints, and the goal is to minimize the design's vulnerability against hardware Trojan attacks and the number of cores required.

Comment 2

The threat model needs a better justification. For example, how the adversary can establish a secure communication channel to trigger the Trojans?

Responses

Comment 3

The paper uses a lot of hypothetical terms without proper justification. For instance, (a) How can the vul() parameter be defined or generalized for any given system? (b) How to obtain the "pte" values? (c) How do we determine the probability parameters prob(v, t)? The problem seems superficially crafted without real-world relation.

Responses

Comment 4

One of the major issues of this paper is the lack of comparison with related work. The paper is tested with the author's own scheme with various parameter tuning. How is the work performed against state-of-the-art research, in particular those mentioned in Sec. 2? The paper needs a thorough redesign of the experimental section, including a comparison with other state-of-the-art work.

Responses

Comment 5

The simulation parameters are selected without proper justifications. For instance, why "pte"=3? Why the Randomly set (and other schemes) are chosen like this?

Responses

Comment 6

The intra-core communication delays are ignored in the simulation, which limits the practicality.

Responses

Comment 7

Some of the tasks are time-critical, but time or ”deadline” constraints are not modeled.

Responses

Reviewer 3

Comment 1

It appears that the key focus is on the vulnerability of communication, however, the authors do not clearly describe with a simple explanation of what sort of issues these could be. It would be good if they did provide that.

Responses

Q1. It would be good if they provided a clear and simple explanation of the vulnerabilities in communication.

A1. In this revised manuscript, we have given a simple example to illustrate the calculation of communication vulnerability. This example is presented in Section 2.3, which is also presented as follows.

Salmani et al. [25] introduced a method for analyzing hardware description languages at the behavioral level. This approach entails analyzing data and control flows at the behavioral level to determine the execution hardness of each statement and the observability of internal signals. Such analysis aids in assessing the detectability of Trojan insertions across various sections of the code.

To assess the vulnerability of circuits, we adopt this method. We illustrate this method using a simple example. Fig. 1 displays the sample code, with statement hardness determined by the ratio of execution control signal range to declared range. For instance, if line 7 confines X to 0-10 while its declared range is 0-15, the hardness for line 8, controlled by X, is 11/16=0.6875. Multiple signals multiply their weights for statement hardness. Fig. **??** details statement hardness and variable ranges. Fig. **??** shows control flow stacks. Fig. **??** exhibits the data flow graph, with node denoting signals and directed edges denoting dependencies and weights denoting the sum of weights of corresponding assignment statement. Fig. **??** assesses observability of signals from output Z; internal signal P’s influence by X yields 0 observability. Statement vulnerability equals weight multiplied by target signal’s observability to output.

Comment 2

The reason to minimize cores isn’t stated in the introduction. Is the reason so that an application is schedulable?

Responses

Comment 3

In the related work, under section 2.1, does the proposed work also suffer from the same issue that it is not possible to detect all possible modern Trojans that may lead to vulnerability? If so, then how does the proposed work differentiate itself from others in terms of that argument?

Responses

Comment 4

In section 2.2, could the authors explain the relationship between the use of HLS for security constraints [17] and the difficulties that arise from task scheduling? Are these application tasks or is some other HLS task that is being referred to?

Responses

Comment 5

In duplication-with-diversity, how does one ensure that a Trojan has truly not been detected? Could it have been that the Trojan simply wasnt activated during the duplicated execution?

Responses

Comment 6

It seems a little disconnected that the paper presents the security constraints of isolation-with-diversity and then later in 3.3.1 the paper says edge contraction promotes the colocation of dependent tasks on the same core. The presentation of the constraints, and how the presented work evolves beyond it should be made much clearer. For example, state that the paper will introduce some security constraints and the limitations that they impose. Then, say that these will be lifted in our work.

Responses

Comment 7

Since the work relies on vulnerability analysis [25], a quick and brief background or even basic intuition behind it would be very useful to make this paper self-contained. Now, there is very little explanation of the vulnerability of communication and most of it is referred to [25].

Responses

Q1. A quick and brief background or even basic intuition behind it would be very useful to make this paper self-contained.

A1. This comment has similarities to comment 1, and we have given an example in section 2.3 to describe the background, which is also presented as follows.

Salmani et al. [25] introduced a method for analyzing hardware description languages at the behavioral level. This approach entails analyzing data and control flows at the behavioral level to determine the execution hardness of each statement and the observability of internal signals. Such analysis aids in assessing the detectability of Trojan insertions across various sections of the code.

To assess the vulnerability of circuits, we adopt this method. We illustrate this method using a simple example. Fig. 1 displays the sample code, with statement hardness determined by the ratio of execution control signal range to declared range. For instance, if line 7 confines X to 0-10 while its declared range is 0-15, the hardness for line 8, controlled by X, is 11/16=0.6875. Multiple signals multiply their weights for statement hardness. Fig. **??** details statement hardness and variable ranges. Fig. **??** shows control flow stacks. Fig. **??** exhibits the data flow graph, with node denoting signals and directed edges denoting dependencies and weights denoting the sum of weights of corresponding assignment statement. Fig. **??** assesses observability of signals from output Z; internal signal P's influence by X yields 0 observability. Statement vulnerability equals weight multiplied by target signal's observability to output.

Comment 8

I find it somewhat confusing that the motivation includes using 3PIPs, but the paper makes a simplifying assumption with regards to data-dependent tasks being executed on separate cores if they require specific application-specific IP not being included. Isnt this an important part of addressing the complexity of developing a solution with 3PIPs?

Responses

Comment 9

The sections that describe the algorithms and the necessary graph transformations are explained well. However, I find that there needs to be a bit more rational for the choices. For example, why was it necessary for the authors to use a probabilistic approach to computing the number of cores required? Furthermore, the assumptions in the three algorithms are sprinkled within the description, which could be improved if they were stated up front, and explained why they make sense. For example, probabilities of a task on all its possible scheduling results are the same [40]. Please explain as this is an important assumption being made. There are several other examples of assumptions being made that need a bit better justification. Just because it simplifies a certain aspect of the work is less convincing of a reason. (Even in the experiments this is done).

Responses

Comment 10

With respect to the examples, I find that they are good. However, there is an opportunity to select a subset of the graphs to illustrate the key transformations. As it is the proposed work introduces several graphs, and finding a way to illustrate them with pictures is important. I find that an approach would be to use text to extrapolate some of the steps as in Fig 7, instead of having to have each of the worked-out iterations. That space could be used more effectively.

Responses

Comment 11

Why choose only these benchmarks? When I checked the website, there seems to be many more? Please include a description of why these 8 were chosen.

Responses

Comment 12

The paper claims that MC-B forgot to optimize for the vendor cores. Is this true? Or is it perhaps that it wasnt the focus of their work? Please clarify?

Responses
