Chapter 9
Validation of IP Security and Trust

Farimah Farahmandi and Prabhat Mishra

9.1 Introduction

Hardware Trojans are small malicious components added to the circuit. They are designed in a way that they are inactive most of the run-time, however, they change the functionality of the design or defeat the trustworthiness of it by leaking valuable information when they are triggered. Hardware Trojans can be categorized into two architectural types: combinational and sequential. A combinational Trojan is activated based on specific conditions and assignments on a portion of internal signals or flip-flops. On the other hand, sequential Trojans are small finite state machines that are activated based on a specific sequence. Each hardware Trojan consists of trigger and payload parts. Trigger is responsible for producing (activating) a set of specific conditions to activate the undesired functionality. Payload part is responsible for propagating the effect of the malicious activity to the observable outputs.

There are a wide variety of Trojans and they may be inserted during different phases such as specification time by changing timing characteristics or in design time by reusing untrusted available IPs offered by third parties or in fabrication time by altering the mask sets. However, it is more likely that Trojans are inserted during design time than manufacturing time as the attacker has better understanding of the design and can design trigger conditions as extremely rare events.

It is difficult to construct a fault model to characterize Trojan's behavior. Moreover, Trojans are designed in a way that they can be activated under very rare conditions and they are hard to detect. As a result, existing testing methods are impractical to detect hardware Trojans. There are several run-time remedies

F. Farahmandi (✉) • P. Mishra
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
e-mail: ffarahmandi@ufl.edu; prabhat@ufl.edu

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_9

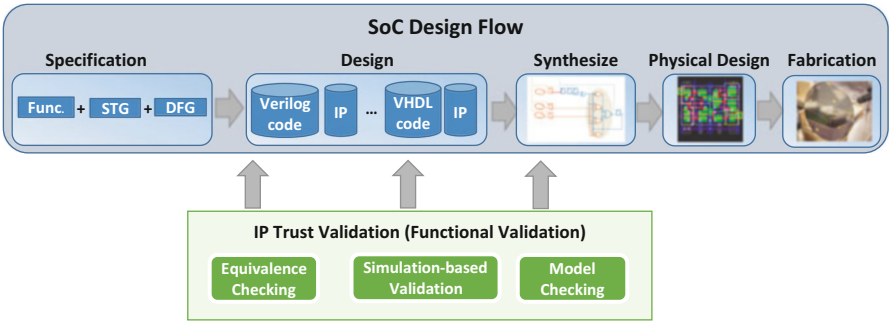

Fig. 9.1 Security validation flow. IPs used in SoC design should be validated against functional metrics using different methods such as equivalence checking, simulation-based validation, and model checking to insure security properties (nothing more, nothing less)

including disable backdoor triggers [27, 29], unused component identification [28], and run-time monitor of some design properties violence.

A major concern with the hardware IPs acquired from external sources (third party IP) is that they may come with deliberate malicious implants to incorporate undesired functionality, undocumented test/debug interface working as hidden backdoor, or other integrity issues. Therefore, security validation to identify untrusted IPs is critical part of the design process of digital circuits. Figure 9.1 shows that security validation should be done in different phases of SoC design to identify malicious functionality which may come from untrusted third party vendors.

There are several approaches focused on logic testing and pre-silicon validation efforts in order to detect and activate the potential hardware Trojans. These methods are described in Sect. 9.2. We review trust validation techniques based on equivalence checking in Sect. 9.3. Finally, model checking-based trust validation is outlined in Sect. 9.4.

9.2 Logic Testing for Trojan Detection

Several approaches are focused on generation of guided test vectors and compare the produced primary outputs with golden/expected outputs to detect and activate hardware Trojans. Traditional test generation techniques may not be beneficial as Trojans are designed in a way that they will be activated under very rare sequences of the inputs. In this section, we review simulation-based validation approaches including rare-node activation, redundant circuit detection, N-detect ATPG, and code coverage techniques as it is shown in Fig. 9.2.

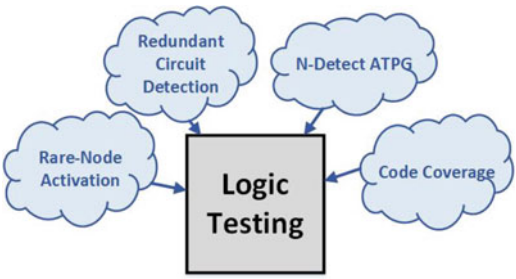

Fig. 9.2 Simulation-based validation for Trojan detection

9.2.1 Utilization Rarely Used Components for Trojan Detection

Unused Circuit Identification (UCI) algorithm is proposed by Hicks et al. [15] to report redundant components in pre-silicon verification time based on data flow dependencies of the design. The authors believed that these components can be used by an attacker to insert malicious functionality as they want to make sure that the inserted Trojan cannot be revealed during pre-silicon verification. To detect the hardware Trojans, the UCI algorithm identifies the signal pairs that they have the same source and sink first. Then, it simulates the HDL netlist with verification tests to find unused circuits that they do not affect on primary outputs and the UCI algorithm deactivates those unused pairs. However, the authors showed in their later work that there are many other types of malicious circuits that do not exhibit the hidden behavior and UCI algorithm cannot detect them [25].

Run-time solutions increase the circuit complexities and designers try to come up with efficient off-chip validation approaches to measure the trustworthiness of designs. Waksman et al. [28] proposed a pre-fabrication method to flag almost unused logic components based on their Boolean functions and their control values (FANCI). The authors believe that the almost-unused logic (rare nodes) will be the targets of attackers to insert hard-to-detect Trojans as the rare nodes have rarely impact on the functionality of the design outputs. They construct an approximate truth table (can also be mapped to vectors) for each internal signal to determine its effect on primary outputs. They use different heuristics to identify backdoors from control value vectors. However, this method reports 1–8 % of the total nodes of a design as potential malicious logic and it may incorrectly flag many as safe gates (false positives). Moreover, if the design is completely trustworthy, this method still reports many nodes as suspicious logic.

FANCI sets a probability threshold for trigger conditions and marks every signal that has lower activation probability than the pre-defined threshold as a suspicious signal. For example, suppose that the threshold is defined as 2^{-32}. If a Trojan trigger is a 32-bit specific value at a specific clock cycle, the probability of trigger activation is 2^{-32} and it is marked as malicious signal. DeTrust [32] has introduced a type of Trojans whose trigger happens across multiple clock cycles so the probability of their activation is computed higher and FANCI will report them as safe gates by mistake. For example, if the 32-bit trigger arrives in 8-bit chunks through 4

consecutive clock cycles, FANCI computes the trigger activation probability as 2^{-8} and because it is higher than the threshold (2^{-32}), it marks them as safe gates.

VeriTrust which is proposed by Zhang et al. [33] tires to find unused or nearly unused circuits which are not active during verification tests. The method classifies trigger conditions in two categories: bug-based and parasite-based hardware Trojans. Bug-based Trojan deviates the functionality of the circuit. On the other hand, parasite-based Trojans add extra functionality to the normal functionality of the design by introducing new inputs. In order to find the parasite-based trigger inputs, it first verifies the circuit by the existing verification tests. In the next step, it finds the entries of design functionality (like entries of Karnaugh-map) that are not covered during the verification. Then, it sets uncovered entries as don't cares to identify redundant inputs. DeTrust [32] introduced a type of Trojans where each of its gates is driven by a subset of primary inputs. Therefore, VeriTrust cannot detect this type of Trojan.

9.2.2 ATPG-Based Test Generation for Trojan Detection

In a recent case study [31], code coverage analysis and Automatic Test Pattern Generation (ATPG) are employed to identify Trojan-inserted circuits from Trojan-free circuits. The presented method utilizes test vectors to perform formal verification and code coverage analysis in the first step. If this step cannot detect existence of the hardware Trojan, some rules are checked to find unused and redundant circuits. In the next step, the ATPG tool is used to find some patterns to activate the redundant/dormant Trojans. Code coverage analysis is done over RTL (HDL) third party IPs to make sure that there are no hard-to-activate events or corner-case scenarios in the design which may serve as a backdoor of the design and leak the secret information [1, 31]. However, Trojans may exist in design that have 100 % code coverage.

Logic testing would be beneficial when it uses efficient test vectors that can satisfy the Trojan triggering conditions as well as propagate the activation effect to the observable points such as primary outputs. Therefore, the test can reveal the existence of the malicious functionality. These kinds of tests are hard to create since trigger conditions are satisfied after long hours of operation and they are usually designed with low probability. As a result, traditional use of existing test generation tools like ATPGs is impractical to produce patterns to activate trigger conditions. Chakraborty et al. proposed a technique (which is called MERO) to generate tests to activate rare nodes multiple times [6]. Their goal is to increase the probability of satisfying trigger conditions to activate Trojans. However, the effect of activation of trigger conditions caused by MERO tests can be masked as it does not consider payload when it generates the test. Saha et al. [23] proposed a technique based on genetic algorithm to guide ATPG tool in order to generate tests that not only activate Trojan triggers but also propagate the effect using payload nodes. Their goal is to observe the effect of the Trojan circuit from primary outputs.

A Trojan localization method was presented in [1]. The proposed method consists of four steps. In the first step, the design is simulated using random test vectors or tests generated by sequential ATPG tool to identify easy-to-detect signals. In the second step, full scan N-detect ATPG is employed to detect rare nodes. In the next step, an SAT solver is utilized to perform equivalence checking of suspicious netlist against the golden netlist to prune the search space of suspicious signals. Finally, uncovered gates are clustered using region isolation method to find the gates responsible for malicious functionality. However, as this approach uses SAT solvers, it fails for large and complex circuits. The method also requires a golden netlist which is usually hard to get.

Pre-defined templates are used to detect Trust-HUB benchmarks (https://www.trust-hub.org/) by Oya et al. [20]. First, a structural template library is constructed based on existing types of Trojans and their features. In the next step, the templates are scored based on their structures and whether these structures are observed only in Trojan circuits. Then a score threshold is defined to classify Trojan-inserted circuits from Trojan-free circuits. However, this approach may fail when an adversary inserts new Trojans.

9.3 Trojan Detection Using Equivalence Checking

Equivalence checking is used to formally prove two representations of a circuit design exhibit exactly the same behavior. From security point of view, verification of the correct functionality is not enough. The verification engineers have to make sure that no additional activity exists besides the normal behavior of the circuit. In other words, it is necessary to make sure that the IP is performing the expected functionality - nothing more and nothing less [13]. Equivalence checking is a promising approach to measure the level of trust for a design.

Figure 9.3 shows a traditional approach for performing equivalence checking using SAT solvers. The primary outputs of specification and implementation are fed to "xor" gates and the whole design (specification, implementation, and extra xor gate) are modeled as Boolean formulas (CNF clauses). If the specification and implementation are equivalent, the output of the xor gate should be always zero (false). If the output becomes true for any input sequence, it implies that the

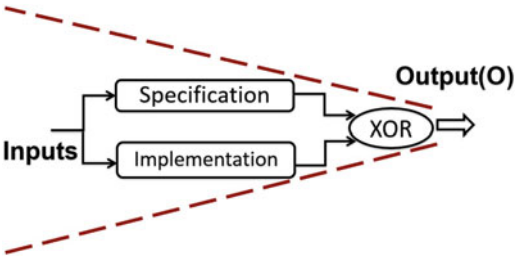

Fig. 9.3 Equivalence checking using SAT solvers

specification and the implementation are producing different outputs for the same input sequence. Therefore, the constructed CNF clauses of the input cone of "O" are used as an input of an SAT solver to perform equivalence checking. If the SAT solver finds a satisfiable assignment, the specification and implementation are not equivalent. SAT solver-based equivalence checking techniques can lead to state-space explosion when large IP blocks are involved with significantly different specification and implementation. Similarly, SAT solver-based equivalence checking approaches fail for complex arithmetic circuits with larger bit-widths because of bit blasting of arithmetic circuits.

9.3.1 Gröbner Basis Theory for Equivalence Checking of Arithmetic Circuits

A promising direction to address the state-space explosion problem in equivalence checking of hardware design is to use symbolic computer algebra. Symbolic algebraic computation refers to application of mathematical expressions and algorithmic manipulations methods to solve different problems. Symbolic algebra especially *Gröbner basis* theory can be used for equivalence checking and hardware Trojan identification as it formally checks two levels of a design and search for components that cause mismatch or change the functionality (hardware Trojans).

Computer symbolic algebra is employed for equivalence checking of arithmetic circuits. The primary goal is to check equivalence between the specification polynomial f_{spec} and gate-level implementation C to find potential malicious functionality. The specification of arithmetic circuit and implementation are formulated as polynomials. Arithmetic circuits constitute a significant portion of datapath in signal processing, cryptography, multimedia applications, error root causing codes, etc. In most of them, arithmetic circuits have a custom structure and can be very large so the chances of potential malfunction are high. These bugs may cause unwanted operations as well as security problems like leakage of secret key [3]. Thus, verification of arithmetic circuits is very important.

Using symbolic algebra, the verification problem is mapped as an ideal membership testing [10, 12, 24]. These methods can be applied on combinational [18] and sequential [26] Galois Filed $\mathbb{F}_{2^k}$ arithmetic circuits using Gröbner Basis theory [8] as well as signed/unsigned integer $\mathbb{Z}_{2^n}$ arithmetic circuits [10, 13, 30]. Another class of techniques are based on functional rewriting [7]. The specification of arithmetic circuit and implementation are converted to polynomials which constructs a multivariate ring with coefficients from $\mathbb{F}_{2^k}$. These methods use *Gröbner basis* and *Strong Nullstellent* over Galois field to formulate the verification problem as an ideal membership testing of specification polynomial f_{spec} in the ideal constructed by circuit polynomials (ideal I). Ideal I can have several generators, one of these generators is called Gröbner basis. First, Gröbner basis theory [8] is briefly described. Next, the application of Gröbner basis theory for security verification of integer arithmetic circuits is presented.

Let $M = {x_1}^{\alpha_1} {x_2}^{\alpha_2} \ldots {x_n}^{\alpha_n}$ be a monomial and $f = C_1M_1 + C_2M_2 + \cdots + C_tM_t$ be a polynomial with $\{c_1, c_2, \ldots, c_t\}$ as coefficients and $M_1 > M_2 > \cdots > M_t$. Monomial $\text{lm}(f) = M_1$ is called leading monomial and $\text{lt}(f) = C_1M_1$ is called leading term of polynomial f. Let $\mathbb{K}$ be a computable field and $\mathbb{K}[x_1, x_2, \ldots, x_n]$ be a polynomial ring in n variables. Then $< f_1, f_2, \ldots, f_s >= \{\sum_{i=1}^{n} h_i f_i : h_1, h_2, \ldots, h_s \in \mathbb{K}[x_1, x_2, \ldots, x_n]\}$ is an ideal I. The set $\{f_1, f_2, \ldots, f_s\}$ is called generator or basis of ideal I. If $V(I)$ shows the affine variety (set of all solution of $f_1 = F_2 = \cdots = f_s = 0$) of ideal I, $I(V) = \{f_i \in \mathbb{K}[x_1, x_2, \ldots, x_n] : \forall v \in V(I), f_i(v) = 0\}$. Polynomial f_i is a member of $I(V)$ if it vanishes on $V(I)$. Gröbner basis is one of the generators of every ideal I (when I is other than zero) that has a specific characteristic to answer membership problem of an arbitrary polynomial f in ideal I. The set $G = \{g_1, g_2, \ldots, g_t\}$ is called Gröbner basis of ideal I, if $\forall f_i \in I, \exists g_j \in G : \text{lm}(g_j) | \text{lm}(f_i)$.

The Gröbner basis solves the membership testing problem of an ideal using sequential divisions or reduction. The reduction operation can be formulated as follows. Polynomial f_i can be reducible by polynomial g_j if $\text{lt}(f_i) = C_1M_1$ (which is non-zero) is divisible by $\text{lt}(g_i)$ and r is the remainder ($r = f_i - \frac{\text{lt}(f_i)}{\text{lt}(g_j)}.g_j$). It can be denoted by $f_i \xrightarrow{g_j} r$. Similarly, f_i can be reducible with respect to set G and it can be represented by $f_i \xrightarrow{G}_+ r$.

The set G is Gröbner basis ideal I, if $\forall f \in I, f_i \xrightarrow{G}_+ 0$. Gröbner basis can be computed using Buchberger's algorithm [4]. Buchberger's algorithm is shown in Algorithm 9.1. It makes use of a polynomial reduction technique named S-polynomial as defined below.

Definition 1 (S-polynomial). Assume $f, g \in \mathbb{K}1, x_2, \ldots, x_n]$ are non-zero polynomials. The S-polynomial of f and g (a linear manipulation of f and g) is defined as: $\text{Spoly}(f, g) = \frac{\text{LCM}(\text{LM}(f), \text{LM}(g))}{\text{LT}(f) * f} - \frac{\text{LCM}(\text{LM}(f), \text{LM}(g))}{\text{LT}(g) * g}$, where $\text{LCM}(a, b)$ is a notation for the least common multiple of a and b.

Algorithm 9.1: Buchberger's algorithm [4]

```
1:  procedure GRÖBNER BASIS FINDER
2:      Input: ideal I =< f1, f2, ..., fs > ≠ {0}, initial basis F = {f1, f2, ..., fs}
3:      Output: Gröbner Basis G = {g1, g2, ..., gt} for ideal I
4:      G = F
5:      V = G × G
6:      while V ≠ 0 do
7:          for each pair (f, g) ∈ V do do
8:              V = V − (f, g)
9:              Spoly(f, g) →G r
10:             if r ≠ 0 then
11:                 G = G ∪ r
12:                 V = V ∪ (G × r)
        return G
```

Example 1. Let $f = 6*{x_1}^4*{x_2}^5+24*{x_1}^2-x_2$ and $g = 2*{x_1}^2*{x_2}^7+4*{x_2}^3+2*x_3$ and we have $x_1 > x_2 > x_3$. The S-polynomial of f and g is defined below:

$$\mathrm{LM}(f) = {x_1}^4 * {x_2}^5$$

$$\mathrm{LM}(g) = {x_1}^2 * {x_2}^7$$

$$\mathrm{LCM}({x_1}^4 * {x_2}^5, {x_1}^2 * {x_2}^7) = {x_1}^4 * {x_2}^7$$

$$\begin{aligned}\mathrm{Spoly}(f,g) &= \frac{{x_1}^4 * {x_2}^7}{6 * {x_1}^4 * {x_2}^5} * f - \frac{{x_1}^4 * {x_2}^7}{2 * {x_1}^2 * {x_2}^7} * g \\ &= 4{x_1}^2 * {x_2}^2 - \frac{1}{6} * {x_2}^3 - 2 * {x_1}^2 * {x_2}^3 - {x_1}^2 * x_3\end{aligned}$$

□

It is obvious that S-polynomial computation cancels leading terms of the polynomials. As shown in Algorithm 9.1, Buchberger's algorithm first calculates all S-polynomials (lines 7-9 of Algorithm 9.1) and then adds non-zero S-polynomials to the basis G (line 11). This process repeats until all of the computed S-polynomials become zero with respect to G. It is obvious that Gröebner basis can be extremely large so its computation may take a long time and it may need large storage memory as well. The time and space complexity of this algorithm are exponential in terms of the sum of the total degree of polynomials in F, plus the sum of the lengths of the polynomials in F [4]. When the size of F increases, the verification process may be very slow or in the worst-case may be infeasible.

Buchberger's algorithm is computationally intensive and it may affect the performance drastically. It has been shown in [5] that if every pair (f_i, f_j) that belongs to set $F = \{f_1, f_2, \ldots, f_s\}$ (generator of ideal I) has a relatively prime leading monomials ($\mathrm{lm}(f_i).\mathrm{lm}(f_j) = \mathrm{LCM}(\mathrm{lm}(f_i).\mathrm{lm}(f_j))$) with respect to order $>$, the set F is also Gröbner basis of ideal I.

Based on these observations, efficient equivalence checking between specification of an arithmetic circuit and its implementation can be performed as shown in Fig. 9.4. The major computation steps in Fig. 9.4 are outlined below:

- Assuming a computational field $\mathbb{K}$ and a polynomial ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$ (note that variables $\{x_1, x_2, \ldots, x_n\}$ are subset of signals in the gate-level implementation), a polynomial $f_{\mathrm{spec}} \in \mathbb{K}[x_1, x_2, \ldots, x_n]$ representing specification of the arithmetic circuit can be derived.
- Map the implementation of arithmetic circuit to a set of polynomials that belongs to $\mathbb{K}[x_1, x_2, \ldots, x_n]$. The set F generates an ideal I. Note that according to the field $\mathbb{K}$, some vanishing polynomials that construct ideal I_0 may be considered as well.
- Derive an order $>$ in a way that leading monomials of every pair (f_i, f_j) are relatively prime. Thus, the generator set F is also Gröbner basis $G = F$. As the combinational arithmetic circuits are acyclic, the topological order of the signals in the gate-level implementation can be used.

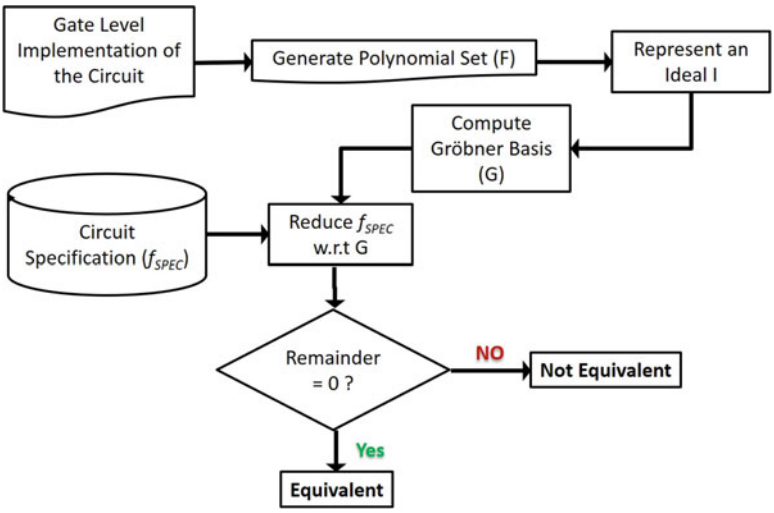

Fig. 9.4 Equivalence checking flow

- The final step is reduction of f_{spec} with respect to Gröbner basis G and order $>$. In other words, the verification problem is formulated as $f_{\text{spec}} \xrightarrow{G}_{+} r$. The gate-level circuit C has correctly implemented specification f_{spec}, if the remainder r is equal to 0. The non-zero remainder implies a bug or Trojan in the implementation.

Galois field arithmetic computation can be seen in Barrett reduction [16], Mastrovito multiplication, and Montgomery reduction [17] which are critical part of cryptosystems. In order to apply the method of Fig. 9.4 for verification of Galios field arithmetic circuits, Strong Nullstellensatz over Galois Fields is used. Galois field is not an algebraically closed field, so its closure should be used. Strong Nullstellensatz helps to construct a radical ideal in a way such that $I(V_{\mathbb{F}^{2^k}}) = I + I_0$. Ideal I_0 is constructed by using vanishing polynomials $x_i^{2^k} - x_i$ by considering the fact that $\forall x_i^{2^k} \in \mathbb{F}_{2^k} : x_i^{2^k} - x_i = 0$. As a result, the Gröbner basis theory can be applied on Galois field arithmetic circuits. The method in [18] has extracted circuit polynomials by converting each gate to a polynomial and SINGULAR [9] has been used to do the $f_{\text{spec}} \xrightarrow{G}_{+} r$ computations. Using this method, the verification of Galois field arithmetic circuits like Mastrovito multipliers with up to 163 bits can be done in few hours. Some extensions of this method have been proposed in [19]. The cost of $f_{\text{spec}} \xrightarrow{G}_{+} r$ computation has been improved by mapping the computation on a matrix representing the verification problem, and the computation is performed using Gaussian elimination.

The Gröbner basis theory has been used to verify arithmetic circuits over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ in [12]. Instead of mapping each gate to a polynomial, the

repetitive components of the circuit are extracted and the whole component is represented using one polynomial (since arithmetic circuit over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ contain carry chain, the number of polynomials can be very large). Therefore, the number of circuit polynomials is decreased. In order to expedite the $f_{\text{spec}} \xrightarrow{G}_{+} r$ computation, the polynomials are represented by Horner Expansion Diagrams. The reduction computation is implemented by sequential division. The verification of arithmetic circuit over ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]/2^N$ up to 128 bit can be efficiently performed using this method. An extension of this method has been presented in [10] that is able to significantly reduce the number of polynomials by finding fanout free regions and representing the whole region by one single polynomial. Similar to [19], the reduction of specification polynomial with respect to Gröbner basis polynomials is performed by Gaussian elimination resulting in verification time of few minutes. In all of these methods, when the remainder r is non-zero, it shows that the specification is not exactly equivalent with the gate-level implementation. Thus, the non-zero remainder can be analyzed to identify the hidden malfunctions or Trojans in the system. In this section, the use of one of these approaches for equivalence checking of integer arithmetic circuits over $\mathbb{Z}_{2^n}$ is explained. Although the details are different for Galios Field arithmetic circuits, the major steps are similar.

9.3.2 Automated Debugging of Functional Trojans Using Remainders

The previous section describes that a non-zero remainder would be generated if there is a potential Trojan or bug. This section describes how to debug a Trojan using the non-zero remainder. To perform verification, the algebraic model of the implementation is used. In other words, each gate in the implementation is modeled as a polynomial with integer coefficients and variables from $\mathbb{Z}_2$ ($x \in \mathbb{Z}_2 \rightarrow x^2 = x$). Variables can be selected from primary inputs/outputs as well as internal signals in the implementation. These polynomials are driven in a way that they describe the functionality of a logic gate. Equation (9.1) shows the corresponding polynomial of *NOT, AND, OR, XOR* gates. Note that any complex gate can be modeled as a combination of these gates and its polynomial can be computed by combining the equations shown in Eq. (9.1).

$$\begin{aligned} z_1 &= NOT(a) \rightarrow z_1 = 1 - a, \\ z_2 &= AND(a, b) \rightarrow z_2 = a.b, \\ z_3 &= OR(a, b) \rightarrow z_3 = a + b - a.b, \\ z_4 &= XOR(a, b) \rightarrow z_4 = a + b - 2.a.b \end{aligned} \tag{9.1}$$

Finding potential Trojans starts with functional verification. The verification method is based on transforming specification polynomial (f_{spec}) using information (polynomials) that are directly extracted from the gate-level implementation. Then, the transformed specification polynomial is checked to see if the equality to zero holds. To fulfill the term substitution, the topological order of the circuit is considered (primary outputs have the highest order and primary inputs have the lowest). By considering the derived variable ordering, each non-primary input variable which exists in the f_{spec} is replaced with its equivalent expression based on corresponding implementation polynomial. Then, the f_{spec_i} is achieved and the process is continued on the updated $f_{\text{spec}_{i+1}}$ until a zero polynomial or a polynomial that only contains primary inputs (remainder) is reached. Note that, using a fixed variable (term) ordering to substitute the terms in the f_{spec_i}s results in having a unique remainder [8]. The following example shows the verification process of a faulty 2-bit multiplier.

Example 2. Suppose, we want to verify a 2-bit multiplier with gate-level netlist shown in Fig. 9.5 to check there is no additional functionality beside the main functionality in the implementation. Suppose a functional hardware Trojan exists in the design by putting the OR gate with inputs (A_1, B_0) instead of an AND gate as in Fig. 9.5. The specification of a 2-bit multiplier is shown by f_{spec_0}. The verification process starts from f_{spec_0} and replaces its terms one by one using information derived from implementation polynomials as shown in Eq. (9.2). For instance, term $4.Z_2$ from f_{spec_0} is replaced with expression $(R + O - 2.R.O)$. The topological order $\{Z_3, Z_2\} > \{Z_1, R\} > \{Z_0, M, N, O\} > \{A_0, A_1, B_0, B_1\}$ is considered to perform term rewriting. The verification result is shown in Eq. (9.2). Clearly, the remainder is a non-zero polynomial and it reveals the fact that the implementation is buggy. □

$$
\begin{aligned}
&f_{\text{spec}_0} : 8.Z_3 + 4.Z_2 + 2.Z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_1} : 4.R + 4.O + 2.z_1 + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_2} : 4.O + 2.M + 2.N + Z_0 - 4.A_1.B_1 - 2.A_1B_0 - 2.A_0.B_1 - A_0.B_0 \\
&f_{\text{spec}_3}(\text{remainder}) : 2.A_1 + 2.B_0 - 4.A_1.B_0
\end{aligned}
\tag{9.2}
$$

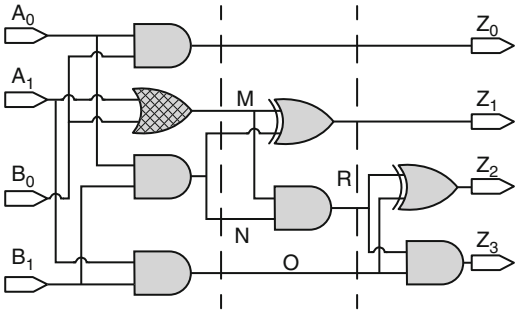

Fig. 9.5 A Trojan-inserted gate-level netlist of a 2-bit multiplier (an AND gate is erroneous and it is replaced by the shaded OR gate)

9.3.2.1 Test Generation for Trojan Detection

It has been shown that if the remainder is zero, there is no malicious functionality in the implementation and implementation is bug free [30]. Thus, when there is a non-zero polynomial as a remainder, any assignment to its variables that makes the decimal value of the remainder non-zero is a bug trace. The remainder is used to generate test cases to activate unknown faults or inserted Trojans [11]. The test is guaranteed to activate the existing malicious functionality in the design. Remainder is a polynomial with Boolean/integer coefficients. It contains a subset of primary inputs as its variables. The presented approach in [11] takes the remainder as an input and finds all of the assignments to its variables such that it makes the decimal value of the remainder non-zero. The remainder may not contain all of the primary inputs. As a result, the approach may use a subset of primary inputs (that appear in the remainder) to generate directed test cases with *don't cares*.

Such assignments can be found using an SMT solver by defining Boolean variables and considering signed/unsigned integer values as total value of the remainder polynomial ($i \neq 0 \in \mathbb{Z}, \text{check}(R = i)$). The problem of using SMT solver is that for each i, it finds at most one assignment to the remainder variables to produce value of i, if possible.

Algorithm 9.2 finds all possible assignments which produce non-zero decimal values of the remainder. It gets remainder R polynomial and primary inputs (PI) exists in the remainder as inputs and feeds binary values to PIs (s_i) and computes the total value of a term (T_j). The value of T_j is either one or zero as it is multiplication of some binary variables (line 4–5). The whole term value may be zero or equal to the term coefficient (C_{T_j}). Then, it computes sum of all terms value to find the corresponding value of the remainder polynomial. If the summation (value) of all the terms is non-zero, the corresponding primary input assignments are added to the set of Tests (lines 8–9).

Example 3. Consider the faulty circuit shown in Fig. 9.5 and the remainder polynomial $R = 2.(A_1 + B_0 - 2.A_1.B_0)$. The only assignments that make R to have a non-zero decimal value ($R = 2$) are ($A_1 = 1, B_0 = 0$) and ($A_1 = 0, B_0 = 1$).

Algorithm 9.2: Directed Test Generation Algorithm

```
1: procedure TEST–GENERATION
2:     Input: Remainder, R
3:     Output: Directed Tests, Tests
4:     for different assignments s_i of PIs in R do
5:         for each term T_j ∈ R do
6:             if (T_j(s_i)) then
7:                 Sum+ = C_Tj
8:         if ( Sum != 0 ) then
9:             Tests = Tests ∪ s_i
       return Tests
```

Table 9.1 Directed tests to activate functional Trojan shown in Fig. 9.5

A_1	A_0	B_1	B_0
1	X	X	0
0	X	X	1

These are the only scenarios that make difference between functionality of an AND gate and an OR gate. Otherwise, the fault will be masked. Compact directed test cases are shown in Table 9.1. □

The remainder generation is one time effort, multiple directed tests can be generated by using it. Moreover, if there are more than one bug in the implementation, the remainder will be affected by all of the bugs. So, each assignment that makes the remainder non-zero activates at least one of the existing faulty scenarios. Thus, the proposed test generation method can also be applied when there are more than one fault in the design.

Example 4. Suppose that in circuit shown in Fig. 9.5, the AND gate with inputs (A_0, B_1) is also replaced with an OR gate by mistake (therefore, there are two functional Trojans in the implementation of the 2 bit multiplier). The remainder polynomial will be equal to $R = 6.A_1 + 4.B_1 + 2.B_0 - 12.A_1.B_1 - 4.A_1.B_0$. It can be seen that assignment $(A_1 = 1, A_0 = 0, B_1 = 0, B_0 = 0)$ manifests the effect of the first fault in Z_1 and assignment $(A_1 = 1, A_0 = 0, B_0 = 0, B_0 = 1)$ activates the second fault in Z_2. □

9.3.2.2 Trojan Localization

So far, we know that the implementation is not correct and we have all the necessary test cases to activate the malicious scenarios. Next goal is to reduce the state space in order to localize the Trojan by using test cases generated in the previous section. The Trojan location can be traced by observing the fact that the outputs can possibly be affected by the existing Trojan. The proposed methodology is based on simulation of test cases.

The tests are simulated and the outputs are compared with the golden outputs (golden outputs can be found from the specification polynomials) to track the faulty (unmatched) outputs in set $E = \{e_1, e_2, .., e_n\}$. Each e_i denotes one of the erroneous outputs. To localize the bug/hardware Trojan, the gate-level netlist is partitioned such that fanout free cones (set of gates that are directly connected together) of the implementation are found. Algorithm 9.3 shows the procedure of partitioning of gate-level netlist of a circuit.

Each gate whose output goes to more than one gate is selected as a fanout. For generality, gates that produce primary outputs are also considered as fanouts. Algorithm 9.3 takes gate-level netlist (*Imp*) and fanout list (L_{fo}) of a circuit as inputs and returns fanout free cones as its output. Algorithm 9.3 chooses one fanout gate from L_{fo} and goes backward from that fanout until it reaches the gate g_i, whose input comes from one of the fanouts from L_{fo} or primary inputs; the algorithm marks all the visited gates as a cone.

Algorithm 9.3: Fanout free cone finder algorithm

```
1: procedure FANOUT–FREE–CONE–FINDER
2:     for Each fanout gate g_i ∈ L_fo do
3:         C.add(g)
4:         for All inputs g_j of g_i do
5:             if !(g_j ∈ L_fo ∪ PI) then
6:                 C.add(g_j)
7:                 Call recursive for all inputs of g_j over Imp
8:                 Add found gates to C
9:         Cones = Cones ∪ C
```

Algorithm 9.4: Bug Localization Algorithm

```
1: procedure BUG–LOCALIZATION
2:     Input: Partitioned Netlist, Faulty Outputs E
3:     Output: Suspected Regions C_S
4:     for each faulty output e_i ∈ E do
5:         find cones that construct e_i and put in C_ei
6:     C_S = C_e0
7:     for e_i ∈ E do
8:         C_S = C_S ∩ C_ei
       return C_S
```

Algorithm 9.4 shows the bug/Trojan localization procedure. Given a partitioned erroneous circuit and a set of faulty (unmatched) outputs E, the goal of the automatic bug localization is to identify all of the potentially responsible cones for the malicious functionality. First, sets of cones $C_{e_i} = \{c_1, c_2, \ldots, c_j\}$ that construct the value of each e_i from set E are found (line 4–5). These cones contain suspicious gates. All of the suspicious cones C_{e_i}s are intersected to prune the search space and improve the efficiency of Trojan localization algorithm. The intersection of these cones are stored in C_S (line 7–8).

If simulating all of the tests show the effect of the malicious behavior in just one of the outputs, it can be concluded that the location of the bug is in the cone that generates this output. Otherwise, the effect of bug might have been detected in other outputs. On the other hand, when the effect of the bug can be observed in all of the outputs, it means that the bug location is closer to the primary inputs as the error has been propagated through all the circuit. Thus, the location of the bug is in the intersection of cones which constructs the faulty outputs.

Example 5. Consider the faulty 2-bit multiplier shown in Fig. 9.6. Suppose the AND gate with inputs (M, N) has been replaced with an OR gate by mistake. So, the remainder is $R = 4.A_1.B_0 + 4.A_0.B_1 - 8.A_0.A_1.B_0.B_1$. The assignments that activate the Trojan are calculated based on Algorithm 9.2. Tests are simulated and the faulty outputs are obtained as $E = \{Z_2, Z_3\}$. Then, the netlist is partitioned to find fanout free cones. The cones involved in construction of faulty outputs are: $C_{Z_2} = \{2, 3, 4, 6, 7\}$ and $C_{Z_3} = \{2, 3, 6, 4, 8\}$. The intersection of the cones that produce faulty outputs is $C_S = \{2, 3, 4, 6\}$. As a result, gates $\{2, 3, 4, 6\}$ are potentially responsible as a source of the error. □

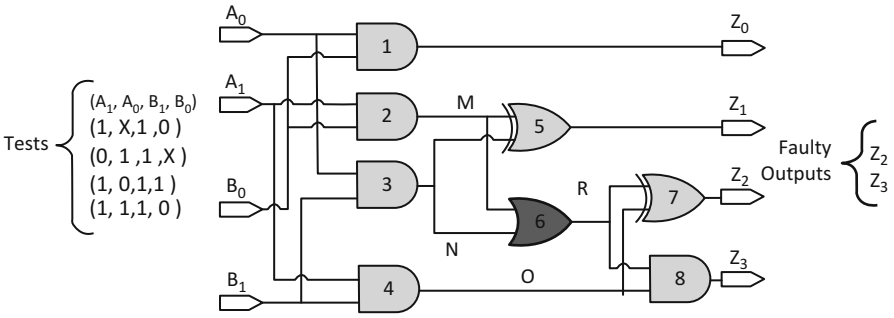

Fig. 9.6 A Trojan-inserted gate-level netlist of a 2-bit multiplier with associated tests to activate the Trojan

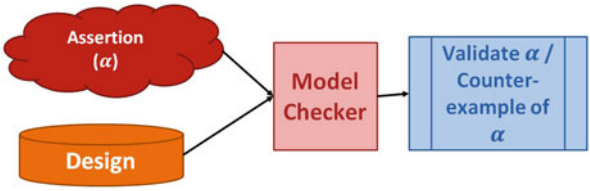

Fig. 9.7 Model checker functionality

9.4 Trojan Detection Using Model Checking

A model checker checks a design against its specification properties (assertions). It takes the design along with the properties (functional behaviors written as Linear Time Temporal Logic formulas) and it formally checks whether properties are satisfied for all of possible states of the design implementation. The high level overview of a model checker functionality is shown in Fig. 9.7. Design specification can be formulated as a set of Linear Time Temporal Logic (LTL) properties [21]. Each property describes one expected behavior of the design. Example 6 shows a property related to a router design.

Example 6. Figure 9.8 shows a router that receives a packet of data from its input channel. The router analyzes the received packet and sends it to one of the three channels based on the packet's address. F_1, F_2, and F_3 receive packets with address of 1, 2, and 3, respectively. Input data consists of three parts: (1) parity (*data_in*[0]) (2) payload (*data_in*[7..3]), and (3) address (*data_in*[2..1]). The RTL implementation consists of one FIFO connected to its input port (F_0) and three FIFOs (one for each of the output channels). The FIFOs are controlled by an FSM. The routing module reads the input packet and asserts the corresponding target FIFO's write signal (write1, write2, and write3).

Suppose the designer wants to make sure that whenever router receives a packet with address 1 (*data_in*[2..1] $= 2'b01$), it eventually resides in the F_1 fifo which is responsible for receiving the data with address 1 (the corresponding write signal of the F_1 fifo will be asserted). The property can be formulated as an LTL formula as follows:

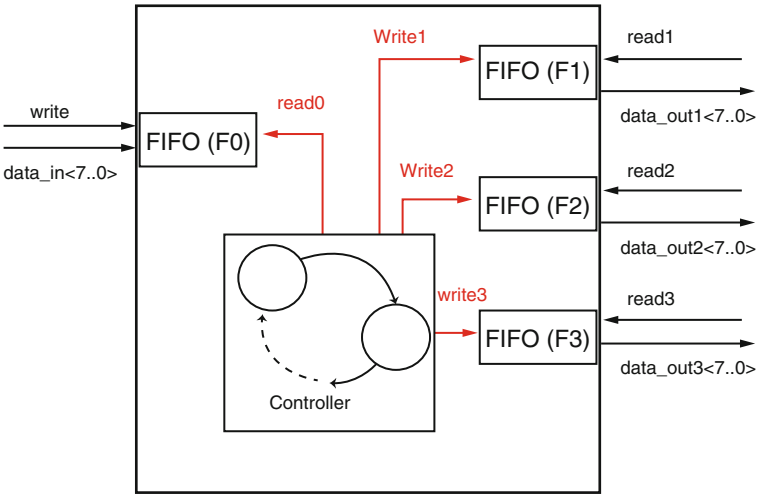

Fig. 9.8 Block diagram of a router design

$assert\ \ P1\ :\ always\ \ ((F_0.data\_in[2]\ =\ 0\ \wedge\ F_0.data\_in[1]\ =\ 1)\ \rightarrow\ eventually\ \ (write1\ =\ 1))$ □

To formally verify the design, model checkers check the design for all possible states. For example, if a design has a 5-bit input, the input introduces 2^5 different states for the design. Therefore, model checkers have state-space constraint when they are applied on large and complex design. Bounded model checking is used to overcome some limitations of model checkers. It checks the design for certain number of transitions and if it finds a violation, it produces a counter-example for the target property [2]. A bounded model checker unrolls the circuit for certain number of clock cycles and extracts Boolean satisfiability assignments for the unrolled design. It feed Boolean satisfiability assignments to the SAT solver to search for assignments that cause the assertion to fail. The model checker generates a counter-example whenever it faces a violation in the implementation of a property. The high level overview of a bounded model checker is shown in Fig. 9.9. Since bounded model checker does not check all states, it cannot formally validate the given property. In bounded model checking, the assumption is that the designer knows in how many cycles a property should be held. Then, the number of cycles (N) is fed to SAT solver engine to find a counter-example within N state sequence transitions (within N clock cycles).

Karri et al. [22] proposed a method to use bounded model checkers to detect malicious functionality in the design. Their considered threat model is that whether an attacker can corrupt the data residing in N-bit data critical registers such as registers containing secret keys of a cryptography design, stack pointer of a processor, or address of a router. A set of properties is written describing that a critical register's data should be accessed safely. Then properties are fed into

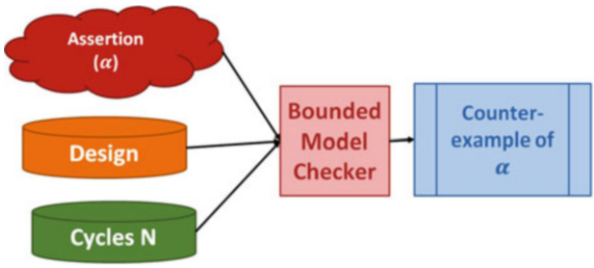

Fig. 9.9 Bounded model checker functionality

bounded model checker accompanied with the design. The bounded checker tires to find a set of states showing unauthorized access to the critical registers which violates the given property.

For example, it is important to make sure that the stack pointer of a processor is only accessed by one of these three ways: (1) increment by one instruction (I_1); (2) decrement by one instruction (I_2); (3) using the reset signal (I_3). The stack pointer should be only accessed from three mentioned ways and if it is accessed from a different way, it should be considered as a malfunction and a security threat. The property can be written as follows:

$$PC\_safe\_access : assert \quad \text{always} \quad (I_{\text{trigger}} \notin I = \{I_1, I_2, I_3\} \rightarrow PC_t = PC_{t-1})$$

The property is fed to a bounded model checker and malfunctions are detected whenever the model checker finds a counter-example for the given property. However, this approach requires prior knowledge to all of the safe ways to access critical data which may not be available. Moreover, translation of safe access ways to LTL formulas is not straightforward. Model checkers fail for large and complex designs and they need the design in a specific language/format. Translation of RTL design to those languages can be error-probe. Moreover, it will face state-space explosion when the Trojan is triggered after thousands or millions of cycles.

Model checkers and theorem provers are combined together in [14] to address the scalability and overhead issues of both methods in verifying security properties. In this method, security properties are translated to formal theorems. In the next step, theorems are converted into disjoint lemmas with respect to the design's submodules. Next, lemmas are modeled as Property Specification Language (PSL) assertions. Then model checker is invoked to check the presence of Trojans in the design.

References

1. M. Banga, M. Hsiao, Trusted RTL: trojan detection methodology in pre-silicon designs, in *Proceedings of 2010 IEEE Internationl Symposium on Hardware-Oriented Security and Trust (HOST)* (2010), pp. 56–59

2. A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without bdds, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Berlin, 1999), pp. 193–207
3. E. Biham, Y. Carmeli, A. Shamir, Bug attacks, in *CRYPTO 2008*, ed. by Wagner, D. LNCS, vol. 5157 (Springer, Heidelberg, 2008), pp. 221–240
4. B. Buchberger, Some properties of gröbner-bases for polynomial ideals. ACM SIGSAM Bull. **10**(4), 19–24 (1976)
5. B. Buchberger, A criterion for detecting unnecessary reductions in the construction of a groebner bases, in *EUROSAM* (1979)
6. R.S. Chakraborty, F. Wolf, C. Papachristou, S. Bhunia, Mero: a statistical approach for hardware trojan detection, in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)* (2009), pp. 369–410
7. M.J. Ciesielski, C. Yu, W. Brown, D. Liu, A. Rossi, Verification of gate-level arithmetic circuits by function extraction, in *IEEE/ACM International Conference on Computer Design Automation (DAC)* (2015), pp. 1–6
8. D. Cox, J. Little, D. O'shea, Ideal, Varieties and Algorithm: An Introduction to Computational Algebraic Geometry and Commutative Algebra (Springer, Berlin, 2007)
9. W. Decker, G.-M. Greuel, G. Pfister, H. Schönemann, SINGULAR 3.1.3 A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra (2012). http://www.singular.uni-kl.de
10. F. Farahmandi, B. Alizadeh, Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction, in *Microprocessors and Microsystems - Embedded Hardware Design* (2015), pp. 83–96
11. F. Farahmandi, P. Mishra, Automated test generation for debugging arithmetic circuits, in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, New York, 2016), pp. 1351–1356
12. F. Farahmandi, B. Alizadeh, Z. Navabi, Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits, in *2014 IEEE Computer Society Annual Symposium on VLSI* (IEEE, New York, 2014), pp. 338–343
13. X. Guo, R.G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *ACM/IEEE Design Automation Conference (DAC)* (2015)
14. X. Guo, R.G. Dutta, P. Mishra, Y. Jin, Scalable soc trust verification using integrated theorem proving and model checking, in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016)
15. M. Hicks, M. Finnicum, S. King, M. Martin, J. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *IEEE Symposium on Security and Privacy (SP)* (2010), pp. 159–172
16. M. Knežević, K. Sakiyama, J. Fan, I. Verbauwhede, Modular reduction in gf (2 n) without pre-computational phase, in *International Workshop on the Arithmetic of Finite Fields* (Springer, Berlin, 2008), pp. 77–87
17. C. Koc, T. Acar, Montgomery multiplication in $GF(2^k)$, Des. Codes Crypt. **14**(1), 57–69 (1998)
18. J. Lv, P. Kalla, F. Enescu, Efficient groebner basis reductions for formal verification of galois field multipliers, in *Proceedings Design, Automation and Test in Europe Conference (DATE)* (2012), pp. 899–904
19. J. Lv, P. Kalla, F. Enescu, Efficient groebner basis reductions for formal verification of galois field arithmetic circuits. IEEE Trans. CAD **32**, 1409–1420 (2013)
20. M. Oya, Y. Shi, M. Yanagisawa, N. Togawa, A score-based classification method for identifying hardware-trojans at gate-level netlists, in *Design Automation and Test in Europe (DATE)* (2015), pp. 465–470
21. A. Pnueli, The temporal semantics of concurrent programs, in *Semantics of Concurrent Computation* (Springer, Berlin, 1979), pp. 1–20
22. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Proceedings of the 52nd Annual Design Automation Conference* (ACM, New York, 2015), p. 112

23. S. Saha, R. Chakraborty, S. Nuthakki, Anshul, D. Mukhopadhyay, Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability, in *Cryptographic Hardware and Embedded Systems (CHES)* (2015), pp. 577–596
24. N. Shekhar, P. Kalla, F. Enescu, Equivalence verification of polynomial datapaths using ideal membership testing. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(7), 1320–1330 (2007)
25. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating uci: building stealthy and malicious hardware, in *IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
26. X. Sun, P. Kalla, T. Pruss, F. Enescu, Formal verification of sequential galois field arithmetic circuits using algebraic geometry, in *Design Automation and Test in Europe (DATE)* (2015), pp. 1623–1628
27. A. Waksman, S. Sethumadhavan, Silencing hardware backdoors, in *2011 IEEE Symposium on Security and Privacy* (IEEE, New York, 2011), pp. 49–63
28. A. Waksman, M. Suozzo, S. Sethumadhavan, Fanci: identification of stealthy malicious logic using boolean functional analysis, in *ACM SIGSAC Conference on Computer & Communications Security* (2013), pp. 697–708
29. A. Waksman, S. Sethumadhavan, J. Eum, Practical, lightweight secure inclusion of third-party intellectual property. IEEE Des. Test Comput. **30**(2), 8–16 (2013)
30. O. Wienand, M. Welder, D. Stoffel, W. Kunz, G.M. Greuel, An algebraic approach for proving data correctness in arithmetic data paths, in *Computer Aided Verification (CAV)* (2008), pp. 473–486
31. X. Zhang, M. Tehranipoor, Case study: detecting hardware trojans in third-party digital ip cores, in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (IEEE, New York, 2011), pp. 67–70
32. J. Zhang, F. Yuan, Q. Xu, Detrust: defeating hardware trust verification with stealthy implicitly-triggered hardware trojans, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (ACM, New York, 2014), pp. 153–166
33. J. Zhang, F. Yuan, L. Wei, Y. Liu, Q. Xu, Veritrust: verification for hardware trust. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(7), 1148–1161 (2015)