5-2022

Structural Checking Tool Restructure and Matching Improvements

Derek Taylor
University of Arkansas, Fayetteville

Structural Checking Tool Restructure and Matching Improvements

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Derek Taylor
University of Arkansas
Bachelor of Science in Computer Science, 2020

May 2022
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____
Jia Di, Ph.D.
Thesis Director

_____
Alexander Nelson, Ph.D.
Committee Member

_____
Miaoqing Huang, Ph.D.
Committee Member

ABSTRACT

With the rising complexity and size of hardware designs, saving development time and cost by employing third-party intellectual property (IP) into various first-party designs has become a necessity. However, using third-party IPs introduces the risk of adding malicious behavior to the design, including hardware Trojans. Different from software Trojan detection, the detection of hardware Trojans in an efficient and cost-effective manner is an ongoing area of study and has significant complexities depending on the development stage where Trojan detection is leveraged. Therefore, this thesis research proposes improvements to various components of the soft IP analysis methodology utilized by the Structural Checking Tool. The Structural Checking Tool analyzes the register-transfer level (RTL) code of IPs to determine their functionalities and to detect and identify hardware Trojans inserted. The Structural Checking process entails parsing a design to yield a structural representation and assigning assets that encompass 12 different characteristics to the primary ports and internal signals. With coarse-grained asset reassignment based on external and internal signal connections, matching can be performed against trusted IPs to classify the functionality of an unknown soft IP. Further analysis is done using a Golden Reference Library (GRL) containing information about known Trojan-free and Trojan-infested designs and serves as a vital component for unknown soft IP comparison. Following functional identification, the unknown soft IP is run through a fine-grained reassignment strategy to ensure usage of up-to-date GRL assets, and then the matching process is used to determine whether said IP is Trojan-infested or Trojan-free. This necessitates a large GRL while maintaining a balance of computational resources and high accuracy to ensure effective matching.

CONTENTS

1 INTRODUCTION

As use of integrated circuits (ICs) becomes more pervasive in all areas of industry and government, it is critical to develop hardware in a timely, cost-effective manner. Stemming from this, third-party hardware intellectual property (IP) is needed to reduce development cost and time due to it being impractical to design every component of an IC in house. These third-party IPs may not always be trustworthy and may comprise hardware Trojans. A hardware Trojan is defined as malicious, intentional modifications to a circuit to perform behavior such as leaking sensitive data, performing a denial-of-service attack, or causing other undesired behaviors. If even a single IP component of an IC is compromised, the integrity of the entire IC is compromised as well.

Currently, there are many approaches proposed for detecting hardware Trojans. These can typically be distinguished by when they are used for Trojan detection. Techniques may target hardware Trojans after production of an IP, after synthesis of an IP, or before synthesis of an IP. For post-production IPs, detection methods focus on side-channel analysis and may use a separate chip to detect hardware Trojans from the resultant IP. One application of side-channel analysis is to examine the power metrics of an IP to detect whether it is Trojan-infested. This particular approach is accomplished using a model containing Trojan-free IP power metrics for comparison [1]. Using this technique, the authors are able to differentiate between Trojan-free and Trojan-infested ICs with 100% accuracy. However, this does have drawbacks incurred by requiring a reference library with a large amount of data to cover a variety of ICs. Another method of hardware Trojan detection using side-channel analysis is presented in [2]. In this variant of side-channel analysis, no golden models are needed, and a support vector machine (SVM) is used to determine the presence of a hardware Trojan. This approach sacrifices accuracy but retains a Trojan detection rate of up to 93% and a classification accuracy of 91.85%. Side channel analysis can also be

performed using current draw. The authors of [2] define a current-related metric called "consistency," and experiments show the consistency measurement is markedly different in a Trojan-infested IC, allowing their detection algorithm to effectively identify ICs infested with Trojans. With the aforementioned techniques requiring a post-production IP, significant cost is incurred in performing Trojan detection, leading to tradeoffs between accuracy, monetary costs, and the time taken to get through analysis.

Another option for Trojan detection is to perform analysis post-synthesis. After the synthesis of an IP, various structures such as netlists and other descriptors can be leveraged for hardware Trojan detection. The authors in [4] conducted post-synthesis examination using netlists. An SVM is used to analyze a netlist to detect three types of hardware Trojans. However, the proposed method was not tested on Trojans without trigger circuits. Additionally, the paper demonstrated a true negative rate of 70%. A true negative refers to successfully ignoring acceptable behavior. Another example of netlist usage is described in [5], where the authors used netlists in conjunction with a neural network to detect hardware Trojans within a gate-level netlist. Results show an average true positive rate of 72.9% with an average true negative rate of 90%. The true positive rate refers to when hardware Trojans are correctly categorized by the method. The authors of [6] posed several techniques and tools utilizing post-synthesis methods based on Boolean function analysis as well as graph neighborhood analysis to perform gate-level Trojan detection. These two methods are combined into their ANGEL (Analyzing the Neighborhood of Graphs to Expose Leakers) analysis technique to yield a false positive rate of between 30 to 40 percent. The authors note the challenge of finding a proper threshold value to use with the ANGEL analysis technique without an automated way to determine such a value. Regarding these post-synthesis detection processes, a lesser penalty of speed is taken for some penalty to accuracy.

Evaluated against the other two sets of methods, this method acts as a middle ground between maximizing either speed or accuracy.

There are pre-synthesis methods to analyze register-transfer-level (RTL) code for hardware Trojans and to convert RTL code into other representations to aid in hardware Trojan detection. An approach described in [7] used RTL code to generate and analyze electromagnetic signatures to detect hardware Trojan types with an accuracy nearing 83%. However, this method may face issues with clock variance due to the hardware Trojan detection method operating within the frequency domain of electromagnetic side-channel radiation. The authors in [8] detailed a technique utilizing machine learning to detect hardware Trojans in RTL code. According to the authors, all Trojan benchmarks were completed without false positive detection on a non-Trojan benchmark. These Trojan benchmarks consisted of nine different Trojans contained across nine variants of *RS232* RTL code as well as a normal *RS232* with no Trojans. While their results were promising, the method employed relies on other processes to aid it at different abstraction levels.

Similar to the procedures used on soft IPs above, Golden Reference Matching methods described in [9] and [10] focus entirely on RTL code analysis rather than netlists, intermediate representations, or post-production IPs. The techniques in [10] build upon those described in [9] to form the overall matching system. Golden Reference Matching operates by analyzing RTL code to extract any component and its primary ports as well as internal signals. These ports and signals are then labeled with assets to describe their functionality in the IP and compared against a Golden Reference Library (GRL). The GRL is a collection of both Trojan-free and Trojan-infested entries with pre-assigned functionalities. The unknown IP being matched against the GRL is compared with each GRL entry to evaluate the entries it most closely resembles. If the unknown IP's best

match is a Trojan-free entry, then it is likely to be Trojan-free. Similarly, if the IP best matches a Trojan-infested entry, then the IP has a higher likelihood of containing a Trojan.

Building upon the methods described in [9], the first step of the matching process in [10] is to utilize a new concept called the Champion GRL. The Champion GRL consists of manually selected designs that are considered to be the most representative of their associated functionality. This means there is a single design associated with each functionality. These champion entries are used to make an initial match to assign a functionality to the unknown soft IP. Included in the Champion GRL are 10 assets used to increase functional matching percentage for the unknown IP. Then, the unknown IP must be compared against the Functionality GRL entries to determine the functionality it best matches. This vastly decreases computational time as the overall GRL size increases since only a single functionality match is required using this two-step process.

This thesis describes accomplished work and changes added to the preceding version of the matching algorithm along with changes to the parsing stage and GRL format. Section 2 explains background information concerning the tool such as internal and external assets, structural checking, and the Golden Reference Matching process and accompanying GRL. Section 3 proposes additions to the existing tool and subsequent implementation. This includes updates to Hardware Description Language (HDL) RTL parsing, HDL RTL internal representation, GRL file format, and optimizations made to the Golden Reference Matching process. Section 4 validates the efficacy of these changes and demonstrates results and improvements, including increased matching effectiveness. Finally, Section 5 details future work and potential changes with the new Structural Checking Tool configuration.

2 BACKGROUND

2.1 ASSETS

To help describe functionality and label primary ports and internal signals, assets are assigned to signals. These labels are crucial to the Golden Reference Matching process, explained in Section 2.2, and describe the purpose and functionality of the signal in the unknown IP. Multiple assets may be assigned to a signal depending on whether more than one asset is needed to capture a signal's functionality. Within the Structural Checking tool, internal and external assets are the fundamental asset categories. Internal assets can further be defined as automatically assigned or manually assigned.

2.1.1 Internal Assets

The authors in [11] and [12] describe the original work detailing initial definition and use of internal assets. Internal assets primarily describe the workings of internal signals but may also be applied to primary ports signals. Internal assets can be further distinguished as manually assignable and automatically assigned assets. Automatically assigned internal assets are used to describe different HDL code structures and the signals within them. These HDL structures include process statements, conditional expressions, concurrent expressions, sequential expressions, procedure statements, functions, generate statements, and constant values. Each of these types of internal assets are called automatically assigned since they are not manually assigned by a user and are instead assigned during the parsing step of the Structural Checking Tool. Currently, the manually assignable internal assets remain unchanged from the aforementioned original works.

2.1.2 External Assets

The original work in [11] and [12] also details the basic external asset system and initial external asset definitions. These external assets are divided into Data, Timing, System Control, Specific System Control, and Miscellaneous. These categories as well as several example assets are show in Table 1. External assets are assigned to primary ports and internal signals to describe the use of signals in an unknown IP.

Table 1. External Asset Categories and Example Assets

External Asset Category	Example Assets
Data	DATA_COMPUTATIONAL, DATA_MEMORY, DATA_PERIPHERAL
Timing	STATUS, SYSTEM_TIMING, COUNT
System Control	RESET, ENABLE, INSTRUCTION
Specific System Control	MEMORY_OP, REGISTER_FILE_CONTROL, BUS_CONTROL
Miscellaneous	CRITICAL, KEY, DUTY_CYCLE

Each category has varying numbers of assets relating to different kinds of signal usages. The Data category contains assets relating to the transfer of data such as computational data or memory data. Assets in the Timing category pertain to signals used for anything related to timing including assets for counters and delays. For System Control, all assets within are related to more broad control of systems. In contrast to these assets, Specific System Control contains those assets that are for specific control of systems and typically apply to only one type of system. The final category, Miscellaneous, is used to hold several assets that do not properly fit into other categories in addition to a few assets used as default values for various internal usages. Some internal usage assets from this category include *UNUSED* and *UNKNOWN* while other assets like *KEY* and *REGISTER* may be assigned to signals.

2.1.3 Asset Filtering

Asset filtering is detailed in [9] and pertains to propagating assets assigned to a signal, all connected signals, and any ancestors or descendants. With propagation, the tool obtains a better understanding of the correlation between signals and can detect conflicting asset assignments or evaluate if a suspicious asset has been propagated through to areas where it should not be used. External assets assigned to primary circuit inputs propagate through the entire circuit and onto their descendant primary circuit outputs. Additionally, the reverse is also true, so external assets assigned to primary circuit outputs propagate backwards to primary inputs.

2.2 GOLDEN REFERENCE MATCHING

The authors of [9] define Golden Reference Matching as the process of matching an unknown IP by comparing it against a GRL containing a mix of Trojan-free and Trojan-infested IPs. By conducting this matching process, it is determined whether the unknown IP contains Trojans. For each entry contained in the GRL, a percentage match is calculated between the entry and the unknown IP based on measures of asset similarity. Using the highest percentage match, Golden Reference Matching yields a probabilistic result based on the general functionality of the unknown IP as well as indicating presence or absence of Trojans.

2.2.1 Basic Matching

Basic matching stems from calculating a percentage match of the asset characteristics between the asset sets of an unknown soft IP and GRL entries. By comparing individual assets within an asset set, a percentage match for the given characteristic can be calculated. Averaging the percentage match of all characteristics allows a determination of the overall percent match for a particular characteristic. After comparing all characteristics, the six percentage matches are

averaged to surmise an overall match for the unknown soft IP. As an exception, there are cases where an unknown IP or a GRL entry may not have assets in each characteristic. In this instance, the empty characteristics are not included in the overall percentage match calculation.

2.2.2 Partial Matching

Originally introduced to the Structural Checking Tool by the authors of [9], partial matching is utilized when assets are not perfectly identical but share a similar purpose within a soft IP. Furthermore, basic matching provides a partial match if an asset contained in the unknown IP or a GRL entry is a generic asset while the other is specific. When this applies, a 50% match is assigned to the two assets. One possible example of a 50% match between related assets could be a match between *DATA_SENSITIVE* and *DATA_MEMORY*. This happens as a result of *DATA_MEMORY* being a less specific form of *DATA_SENSITIVE*.

2.2.3 Asset Reassignment

Introduced in [14], asset reassignment is the process by which a specific asset is changed into a more general asset. The idea of asset reassignment originates from the previous subsection when a more specific asset may be matched to a generic counterpart. If two signals are theoretically the same but differ due to changes in assets introduced over time, a generic asset can be given to the signals instead. Given that these two assets are in the same category when comparing a specific and general asset, the specific asset is reassigned to the general asset to increase the matching percentage. Examples of this include the *DATA_COMPUTATIONAL* and *DATA_SENSITIVE* assets. Both of these assets are data category assets. In this case, *DATA_COMPUTATIONAL* can be reassigned to *DATA_SENSITIVE* to increase the matching percentage from 50% to 100%.

2.2.4 Statistical Matching

Also presented by the authors of [14], statistical matching adds various statistical formula to aid the matching process. Frequently used assets included within a single characteristic of many GRL entries should be treated as having a lower weight compared to assets found in a small subset of GRL entries because the less common assets contribute more to the identifiability of a GRL entry. Using this technique, an average asset weight can be evaluated based on the sum of all matched asset weights divided by the total number of matched assets of a particular characteristic.

$$Weight_{char} = \frac{Characteristic_{char} * AverageAssetWeight}{\sum_{i=A}^{F} Characteristic_i * AverageAssetWeight} * 100$$

Equation 1. Characteristic Weight Calculation [10]

Equation 1 shows the average asset weight calculation of a particular characteristic. After determining the average asset weight for a single characteristic, it is divided by the sum of all characteristics' average asset weights and converted into a percentage based upon the sum of all six characteristics' average asset weight in the GRL.

2.2.5 Golden Reference Library

The GRL is a collection of soft IPs retrieved from both Trust-Hub [15, 16] and OpenCores [17]. A general functionality is associated with each design file to label the overall function of the soft IP, and the tool also generates an asset pattern for the known IP. Entries are manually labeled with a functionality describing whether they are Trojan-free or Trojan-infested. These designs are well-documented, so there is confidence concerning the design functionality.

2.2.6 Champion Golden Reference Library

Introduced by the authors of [10] to reduce the computational complexity of establishing the functionality of an unknown IP, each GRL entry is manually inspected to copy specific entries

into a separate GRL. Both the Champion Golden Reference Library and the Functionality Golden Reference Library discussed further in Section 2.2.7 supersede the Golden Reference Library discussed in Section 2.2.5. Any entry containing too few asset sets or insufficient asset set variety will introduce bias during the asset reassignment process. To remedy this, the manual inspection process is employed, so one GRL entry is chosen as the most representative of a specific functionality based on the analysis of its asset sets, asset variety, and asset makeup.

2.2.6.1 Coarse-Grained Asset Reassignment

Due to entry limitations and decreased entry count of the Champion GRL, a coarse-grained asset reassignment approach is necessary. Since a fine-grained comparison of the unknown IP and the Champion GRL entries is implemented, the top Champion GRL matches will have a lower overall matching percentage with the unknown IP when they are compared against a design with identical functionality. To address this, matching with the Champion GRL is aided by usage of coarse-grained matching similar to asset reassignment but only applied to external characteristics.

2.2.7 Functionality Golden Reference Library

To supplement the Champion GRL, a Functionality GRL is introduced by the authors of [10]. The introduction of the Champion GRL obsoleted the original GRL and its concepts. Therefore, the Functionality GRL is partitioned according to functionality, so the resource cost of matching is reduced by only matching to a specific category established by the Champion GRL Matching process. Consequently, the GRL process breaks down into a two-step process where an unknown IP will first have its functionality determined by its Champion GRL match and is then further compared against the Functionality GRL entries pertaining to the chosen functionality.

2.2.7.1 Fine-Grained Asset Reassignment

For GRL entry matching, fine-grained asset reassignment was added to increase the matching percentage between an unknown soft IP and the Functionality GRL entries. In this type of asset reassignment, only Functionality GRL designs are assigned. Any unknown IP used will have its most recently assigned assets while the Functionality GRL entries may not be as up-to-date, negatively affecting matching results. During asset reassignment using a specific characteristic from the Functionality GRL, the same characteristic of the unknown IP is also used. Initially, all asset sets from a single characteristic are considered alongside external characteristics since most internal assets are automatically assigned. Next, each Functionality GRL entry receives the asset sets from the same characteristic as the unknown IP, and assets from the Functionality GRL are compared against those of the unknown IP. Since the most recent and accurate assets are assigned to the unknown IP, this reassignment only happens for assets within the Functionality GRL. If two particular assets from the unknown IP and Functionality GRL entry are the same, then asset reassignment is unnecessary.

3 METHODOLOGY AND IMPLEMENTATION

While the matching process described in Section 2 is a starting point, there are further issues to be addressed. These include optimizing the content of asset sets, defining new assets to help describe functionalities, and improving computational efficiency of the matching and related processes. In rebuilding several sections of the Structural Checking Tool, the parser was modified to use the *hdlConvertor* Python library. This allows for conversion of RTL code into an abstract syntax tree (AST). Additionally, the internal representation of parsed RTL code from a soft IP has been reworked, so each design can be represented as a directed graph where the direction of node edges is based on driving and driven signals within the RTL code. As a byproduct of directed graph representation, other processes, like asset propagation, are made simpler by being tied to ancestor and descendant nodes. These processes can also utilize the directed nature of the graph to further convey the logic flow of the IP from the primary inputs to the primary outputs. Complementing these changes, the formatting and information contained within GRL entry files were upgraded to effectively detail GRL entries.

3.1 HDL RTL PARSING

To begin the process of converting RTL code into a directed graph representation, *hdlConvertor* is employed to translate the RTL code into a descriptive AST. In addition to the AST representation, *hdlConvertor* also supports a larger group of syntax of HDL languages, allowing improved code coverage. To manipulate the AST provided by *hdlConvertor*, several *builder* classes are utilized to traverse the tree and extract the desired information for directed graph construction.

3.1.1 Composed Graph Builder

The construction of the directed graph begins and ends in this phase of the building process. The Composed Graph Builder is first given the file path of the top-level HDL file of a soft IP. Afterwards, the directory of the file is set as the top-level directory (TLD), so all HDL files located within are considered a *design*. In this instance, the top-level HDL file contains the top-level component for a given design. Once the top-level file and TLD are located, the top-level HDL code is parsed to find the top-level component. After this component is found, regular expressions are used to reduce computational time in finding declared components within each HDL file in the TLD. While the Composed Graph Builder iterates over these HDL files, a cache of known components is built to include component locations. Additionally, a signal graph is created and cached along with a cache of user-defined packages to contain important definitions used throughout the design. After the construction of both caches, it is possible to establish all subcomponents based on the body of the top-level HDL file. To maintain a hierarchy for internal representation of the IP, a tree structure links components as parent-child relationships using the Component Tree Builder. Finally, after all caches are built and the component hierarchy is founded with the Component Tree Builder, the signal graphs built for each component are combined to form the final representation of the soft IP as a directed graph. The component cache contains a signal graph for each component to prevent the computationally expensive process of using *hdlConvertor* on any given component more than once.

3.1.2 Component Tree Builder

The Component Tree Builder is used to understand the hierarchy and composition of the overall design based on parent-child relationships between components and subcomponents. This builder traverses the relevant component HDL files while maintaining a cache of parsed

hdlContext objects given by *hdlConvertor* to avoid redundant parsing. Furthermore, this builder recursively calls itself for each subcomponent. These calls then branch to all descendant components until the entire design hierarchy is established. After determining this hierarchy, each node in the constructed component tree is given an instance designator to allow tracking of component instantiations. Using this information, functionally-identical components can be differentiated from one another.

3.1.3 Signal Graph Builder

This builder class is the chief component in constructing the directed graph from the RTL code. It is used in conjunction with the builders detailed in Sections 3.1.1 and 3.1.2 to achieve maximal accuracy in a hierarchical manner. The Signal Graph Builder traverses all relevant objects in the AST and constructs a graph such that each signal is a node, and edges are drawn based on the driving-driven relationship contained in the HDL code. In addition to serving this purpose, the Signal Graph Builder automatically assigns relevant internal assets to ensure accurate description of various HDL statements and signal usage. In every signal graph, statement hierarchies are defined to ensure conditional driving statements or statements with driving signal lists have their driving-driven relationships propagated throughout any nested code blocks. For example, an *if* statement wrapping a *for* loop would lead to the interior *for* loop being driven by the exterior *if* statement. This statement hierarchy ensures all connections are formed to show the proper flow of logic throughout various HDL statement types. In addition to statement representations, HDL functions and procedures are also parsed. Internally, these functions and procedures are represented as independent signal graphs and later composed into the signal graphs of their parent components. When composing a function graph into a component's signal graph, a unique

designator is assigned to function signals to ensure they are uniquely identifiable based on the number of function calls within the component.

3.2 GRAPH REPRESENTATION

RTL designs are parsed and converted into a directed graph. This graph is used to convey all connections between signals while highlighting the flow of logic throughout a design. Understanding the logic flow assists with asset propagation to ancestor and descendant nodes, so every path through a design can be more accurately described. For the purposes of graph

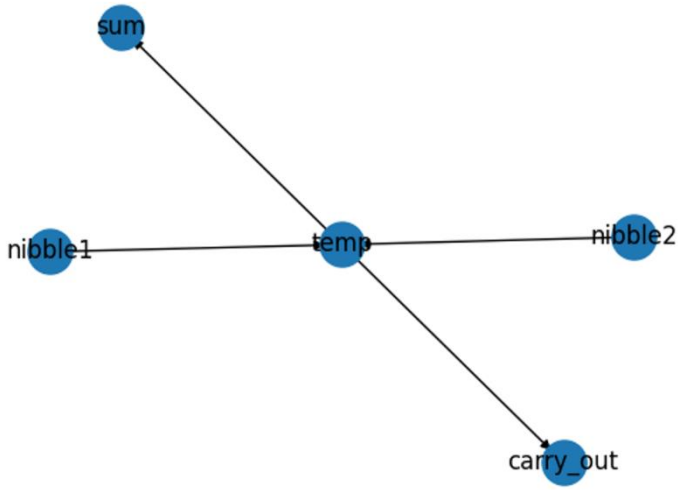

Figure 1: Example Graph for a 2-Bit Adder

representation, a path through the design entails a connection from a primary input port to a primary output port that may pass through intermediary internal signals. In addition to helping with path description, these elaborated logic paths also improve functionality assignment and matching since every path through the design is properly represented from the expression of all signal graphs in the top-level component and all subcomponents. Figure 1 shows an example graph for a 2-bit adder. In this graph, *nibble1* and *nibble2* are input signals, *sum* and *carry_out* are output signals, and *temp* is an internal signal. For this adder, temp is driven by both *nibble1* and *nibble2* while *temp* drives both *sum* and *carry_out*. In addition to the previous points, graph representation

may be leveraged in the future to easily identify certain types of Trojans. For example, it could be used to identify disconnected nodes which point to unwanted or unused logic. It could also allow tracking of each signal assigned as a clock asset to identify undesirable actors and prevent them from affecting clock behavior.

3.3 GOLDEN REFERENCE LIBRARY FILE FORMAT

The Golden Reference Library (GRL) was also reformatted while making improvements to other algorithms related to the matching process. To structure GRL file entries, new GRL files are formatted to JavaScript Object Notation (JSON) specifications. Figure 2 details a generalized layout of the new GRL format. The overall design has a primary and secondary functionality assigned to it as well as a name, creation time, and format version. Additionally, each component is assigned a name, primary functionality, and secondary functionality. Components contain parent component and child component data to determine their positions in the design hierarchy. To save computation time when a GRL entry is read, components also store a copy of their asset frequency data to detail the asset types appearing within it and its child components. The final asset data stored is internal and external asset data for input, output, inout, buffer, linkage, and internal signals. This accounts for all signal types and increases the total number of characteristics to twelve. To supplement the aforementioned information, all components and subcomponents are fully expressed in the GRL file instead of featuring only one instance of each component. Using this method, different functionalities and asset layouts for components are considered instead of registering only a single functionality. Since GRL files can be large and will need to be stored long-term, the new GRL format applies standard compression. The compression for GRL files uses the built-in Python *pickle* library to create a binary object dump of the *entry_data* portion of the GRL file. Next, the binary dump is compressed using the LZMA algorithm and encoded to the

A85 format to realize size benefits when encoding binary data. Lastly, the data is decoded to UTF-8 to maintain file readability and avoid writing binary blobs directly in line with JSON text.

```
{
   "format_version": ...,
   "design_info": {
      "name": ...,
      "primary_functionality": ...,
      "secondary_functionality": ..., "creation_time": ...
   },
   "entry_data": {
      "components": [
         {
            "name": ...,
            "primary_functionality": ..., "secondary_functionality": ...,
            "parent_component": ..., "child_components": [...],
            "asset_frequency_data": {...},
            "input_signal_asset_data": {...}, "output_signal_asset_data": {...},
            "buffer_signal_asset_data": {...}, "linkage_signal_asset_data": {...},
            "inout_signal_asset_data": {...}, "internal_signal_asset_data": {...}
         },
         ...
      ]
   }
}
```

Figure 2. Example Golden Reference Library entry

Because only the *entry_data* portion of the GRL JSON is compressed, it is possible to view the overall design summary without decompressing the file. In the case of the GRL file for the *f32c* processor, the GRL file size was decreased from approximately 1.28 megabytes to 15 kilobytes for a total decrease in size of approximately 98%. This will allow for more GRL entries without taking up significantly more disk space. Overall, these changes produce GRL files that contain more information while improving accuracy and readability.

3.4 REORGANIZING FUNCTIONALITIES

To begin matching improvements, the list of functionalities was elaborated upon and expanded into a set that is more representative of the functionalities that designs may implement. The prior functionality system organized functionalities into a whitelist and blacklist. Whitelist functionalities are those without Trojan behavior while blacklist functionalities do exhibit Trojan behavior. This functional organization is provided below in Table 2.

Table 2. Whitelist and Blacklist Functionalities [10]

Whitelist Functionality	Blacklist Functionality
SHIFT_REGISTER	TROJAN_ENCRYPTION_UNIT
INTERRUPT_UNIT	TROJAN_TRIGGER
COMMUNICATION	TROJAN_COMMUNICATION
ENCRYPTION_UNIT	TROJAN_SHIFT_REGISTER
COMPUTATIONAL	
TIMING	
CONTROL_GENERATION	
REGISTER_FILE	
PERIPHERAL	
DECODER_ENCODER	
DEBUG_INTERFACE	

To revise these functionalities, it is necessary to ensure each functionality describes a category of IP at a similar level of granularity, so no functionality is more heavily weighted than others. Additionally, chosen functionalities must accurately describe any IP. The new system of functionalities is below in Table 3.

Table 3. Updated Functionalities

Functionality	Has Trojan Equivalent
unassigned	No
cannot_determine	Yes
processor	Yes
hardware_accelerator	Yes
memory	Yes
datapath	Yes
power_management	Yes
clock_generation	Yes
timing	Yes
pwm	Yes
cryptography	Yes
computational	Yes
control_unit	Yes
debugging	Yes
port	Yes
communication	Yes
decoder_encoder	Yes
library	Yes
peripheral	Yes

While the updated functionalities no longer categorize blacklist and whitelist functionalities, each functionality now has a Trojan equivalent of itself to precisely express what kind of functionality the design shows and whether it exhibits malicious behavior. Furthermore, the larger number of functionalities helps prevent categories from being too broad in scope, and they more accurately identify and distinguish designs. The *unassigned* functionality is the default functionality used when no other functionality has been assigned. The *cannot_determine* functionality's Trojan and clean variants are used when a currently defined functionality does not fit, but it is known whether or not the design contains a Trojan. Any *processor* functionality components perform operations on an external data source, usually facilitated by an instruction set. Components designated as *hardware_accelerator* aid or optimize some other computation. The *memory* functionality describes components that store information for immediate or later use. Any

components that perform data processing operations or control data flow throughout a design are described by the *datapath* functionality. Components labeled as *power_management* affect power distribution. In the *clock_generation* functionality are components that create or drive the clock signal for distribution throughout a design while the *timing* functionality applies to components that configure or process the clock signal generated by a *clock_generation* component and perform other timing-related actions. Components that modulate an electrical signal to reduce its average power belong to the *pwm* functionality. The *cryptography* functionality encompasses cryptographic functions or cryptographic operations performed by components, and the *computational* functionality details components that implement arithmetic functions or perform arithmetic operations. Components that manage the operation of other components or devices fall into the *control_unit* functionality. The *debugging* functionality describes components that output debugging information and perform exception handling and detection. Functionalities that fall into *port* are components that act as a physical interface to another device. The functionalities related to *communication* are components that facilitate the transfer of data over wired or wireless connections either internally or externally. Any component that decodes or encodes information can be described by the *decoder_encoder* functionality. The *library* functionality applies to packages or libraries from which components or signals are loaded and cannot be expressed as a physical part of an IP despite its presence in RTL code.

3.5 GOLDEN REFERENCE MATCHING

While some stages of matching remain unchanged, others have been removed due to deprecation. Those remaining stages have been overhauled and optimized in various ways. The stages remaining from the initial Golden Reference Matching process are the Statistical Matching Step, the Champion Golden Reference Library Matching Step, and the Functionality Golden

Reference Library Matching Step. The overall matching process has been reorganized to remove extraneous prior work and streamline added matching optimizations.

3.5.1 Champion Golden Reference Library Matching

The Champion Golden Reference Library portion of the matching process, introduced in [10], determines the initial assigned functionality of an unknown soft IP. In addition to the declaration of an unknown soft IP's functionality, this GRL only contains one design per functionality, allowing it to use less resources. The Champion GRL entries are a subset of the Functionality GRL entries and are manually selected so each entry in the Champion GRL is the best representative of the related functionality. The efficacy of the Champion GRL is further increased due to the various functionality changes implemented to more effectively describe and categorize designs. If a Champion GRL entry contains too few asset sets or too few unique assets, then bias is introduced due to the reassignment of general assets to specific assets. Designs containing many asset sets with many uniquely identifying assets may also exhibit bias when the soft IP contains fewer unique assets than a given Champion GRL entry. Addressed previously in this section, the process of selecting a Champion GRL entry to represent a functionality must consider asset sets and all uniquely identifying assets within the entry before being evaluated and added to the Champion GRL. This process is critical to avoid the biases highlighted above and to ensure the initial functionality match is as accurate as possible.

3.5.1.1 Coarse-Grained Asset Reassignment

The coarse-grained asset reassignment approach, retaining its original usage described in [10], is utilized due to the entry limitations of the Champion GRL. The comparisons between an unknown soft IP and the Champion GRL are fine-grained. This leads to top Champion GRL

matches having a lower overall matching percentage against the unknown soft IP when compared to matches of the soft IP against designs of similar or the same functionality. Because of this, the Champion GRL matching process is performed in conjunction with coarse-grained asset reassignment utilized exclusively on external characteristics.

3.5.1.2 Asset Set One

Asset Set One has increased from ten generalized external asset categories to eleven. All external assets are encompassed by these categories. Table 4 shows all categories in *Asset Set One* and the assets which map to *Asset Set One* categories.

Table 4. Reworked Asset Set One

Category	Assets
Data._any	Data: computational, sensitive, critical, test_in, test_out
Data.communication	Data.peripheral
Data.encryption	Data: decryption, _hash, encoding, decoding, key
Data.address	Data.memory
Timing.system_timing	Timing: clock, subsystem_clock, subsystem_timing, test_clock
Timing.status	Timing: ready, done, busy, hold, count, wait, standby
	SpecificSystemControl.communication_status
InstructionSet.instruction	SystemControl: enable, _set, reset, execute, read, write, select, load, shift, interrupt, mode, acknowledge, handshaking, dataflow, flag, request, test_mode_select, test_reset

Table 4. Reworked Asset Set One (Cont.)

Category	Assets
SpecificSystemControl.peripheral_control	Timing: clock_control, subsystem_clock_control
	SpecificSystemControl: interrupt_control, memory_control, communication_control, communication_protocol, bus_control, duty_cycle, phase
	InstructionSet: operand, operation_type, source, destination, program_counter, branch, offset, program_counter_op, data_op, memory_op, interrupt_op, priority, availability, pipeline_clear, pipeline_lock
SpecificSystemControl.exception_handling	SpecificSystemControl.error_handling
Parameter.configuration	Parameter: initialization, frequency, timing, phase, data_width, generate_control, enable
MiscellaneousAsset.unused	MiscellaneousAsset: component, unknown

Many of these assets either map to new asset categories or are newly created external assets entirely. These changes help improve the descriptive capabilities for each reassigned category, thereby improving the efficiency of *Asset Set One*. This stage of reassignment is employed on both the unknown soft IP and the Champion GRL entries, so the assets and functionality yield the highest possible percent match.

3.5.1.3 Asset Set Two

Asset Set Two has also increased from ten general categories to eleven. Typical of GRL entries, broader assets become much more common across designs than other assets with higher specificity. Because data assets are particularly common due to most IP processing data in some way, the primary focus of *Asset Set Two* is to classify data assets into a new category. Table 5 describes the asset categories and assets of *Asset Set Two*. In contrast to *Asset Set One*, some categories in *Asset Set Two* may not have any assets mapped to them. This is due to *Asset Set Two* primarily handling data assets, important distinguishing factors in many designs. As such, some

assets remain without necessitating reassignment since they are uncommon enough to aptly express design characteristics.

Table 5. Reworked Asset Set Two

Category	Assets
Data._any	Data.computational
Data.memory	
Data.communication	
Data.peripheral	
Data.encryption	Data: decryption, _hash, encoding, decoding, key
Data.address	
Data.sensitive	Data.critical
Timing.system_timing	Timing: clock, subsystem_clock, subsystem_timing, status, ready, done, busy, hold, count, wait, standby, test_clock
	SpecificSystemControl.communication_status
InstructionSet.instruction	Timing: clock_control, subsystem_clock_control
	SystemControl: enable, _set, reset, execute, read, write, select, load, shift, interrupt, mode, acknowledge, handshaking, dataflow, flag, request, test_mode_select, test_reset
	SpecificSystemControl: interrupt_control, peripheral_control, memory_control, communication_control, communication_protocol, bus_control, duty_cycle, phase, exception_handling, error_handling
	InstructionSet: operand, operation_type, source, destination, program_counter, branch, offset, program_counter_op, data_op, memory_op, interrupt_op, priority, availability, pipeline_clear, pipeline_lock
Parameter.configuration	Parameter: initialization, frequency, timing, phase, data_width, generate_control, enable
MiscellaneousAsset.unused	Data: test_in, test_out
	MiscellaneousAsset: component, unknown

Top matching percentages determine when to employ *Asset Set Two.* If a certain threshold is met, then functionality matching can continue with only the usage of *Asset Set One*. After some unknown soft IP has completed the asset reassignment process and these reassigned assets have been propagated through the IP, the matching process will perform the reassignments detailed in *Asset Set One*. Once the matching process completes and the soft IP matches to the Champion

GRL entries, the top two matches yielded are compared with a configurable matching threshold. The default configuration value of the matching threshold is 40% and was determined from initial testing in [10]; however, this value may be altered to achieve better results in certain scenarios. Any designs with less than a 40% match are low confidence functionality assignments. If the first match exceeds 40%, then it is necessary to compare with the second highest matching functionality.

Table 6. Functionality Threshold Example

Functionality	Match Percentage
computational	90%
datapath	85%

If an unknown soft IP has matching percentages below this threshold, the confidence in any match is not high enough to justify a functionality assignment. In addition to this first threshold, a second configurable threshold is used. The significance of the second threshold is to signify functionality differences due to design variability within each functionality category, and the default value for this threshold is 15%. If the top two matches have a difference exceeding 15%, then the unknown soft IP is considered part of the top matches' functionalities and moves forward to match against the Functionality Golden Reference Library. In Table 6, the matches for *computational* and *datapath* do not have a difference of greater than 15% and are determined to not be sufficient in deciding the component's functionality.

3.5.2 Functionality Golden Reference Library Matching

Also originating in [10], the Functionality Golden Reference Library Matching (FGRLM) process complements the Champion GRL matching process. As outlined in the original GRL matching process, a design would be matched against all entries within the GRL, which scales poorly as the number of GRL entries increases. To reduce this computational overhead, FGRLM is employed after the Champion GRL matching process is completed. After an unknown IP has its

initial functionality decided, it is then matched against the FGRLM associated with that functionality. By doing this, any unknown soft IP will only be matched against Champion GRL and GRL entries pertaining to the FGRLM-determined functionality rather than being matched against all GRL entries. This approach has the additional benefit of increasing matching percentages because an unknown soft IP is only compared against designs similar in functionality.

3.5.2.1 Fine-Grained Asset Reassignment

To supplement the coarse-grain reassignment sets iterated previously, a fine-grained reassignment strategy, originally reported in [10], is also utilized. Fine-grained reassignment increases the matching percentage between an unknown soft IP and the Functionality GRL entries. In the case of asset reassignment, only the assets of Functionality GRL entries undergo reassignment. Table 7 shows all categories in *Asset Set Three* and their respective assets. Unknown soft IPs will generally have the most recently available assets assigned to them while entries within the Functionality GRL may not contain up-to-date asset assignments, causing biases that could negatively impact matching results. This negative effect is addressed by this reassignment strategy. Because this reassignment strategy is fine-grained, many assets are not reassigned to a particular category and remain as they are.

Table 7. Reworked Asset Set Three

Category	Assets
Data._any	Data.computational
Data.memory	
Data.communication	
Data.peripheral	
Data.encryption	Data: decryption, _hash, key
Data.encoding	Data.decoding
Data.address	
Data.sensitive	

Table 7. Reworked Asset Set Three (Cont.)

Category	Assets
Data.critical	
Timing.clock	Timing.subsystem_clock
Timing.clock_control	Timing.subsystem_clock_control
Timing.system_timing	Timing.subsystem_timing
Timing.status	Timing: ready, done, hold
Timing.busy	Timing: wait, standby
Timing.count	
SystemControl._set	
SystemControl.reset	
SystemControl.execute	
SystemControl.read	SystemControl: write, load
SystemControl.select	SystemControl.enable
SystemControl.shift	
SystemControl.interrupt	
SystemControl.mode	
SystemControl.acknowledge	
SystemControl.handshaking	
SystemControl.dataflow	
SystemControl.flag	
SystemControl.request	
SystemControl.test_mode_select	Data: test_in, test_out Timing.test_clock, SystemControl.test_reset
SpecificSystemControl.peripheral_control	
SpecificSystemControl.communication_control	SpecificSystemControl: communication_protocol, communication_status
SpecificSystemControl.bus_control	
SpecificSystemControl.duty_cycle	SpecificSystemControl.phase
SpecificSystemControl.exception_handling	SpecificSystemControl.error_handling
InstructionSet.instruction	
InstructionSet.operand	
InstructionSet.operation_type	InstructionSet.data_op
InstructionSet.source	InstructionSet: destination, branch, offset
InstructionSet.program_counter	InstructionSet.program_counter_op
InstructionSet.memory_op	SpecificSystemControl.memory_control

Table 7. Reworked Asset Set Three (Cont.)

Category	Assets
InstructionSet.interrupt_op	SpecificSystemControl.interrupt_control
InstructionSet.priority	
InstructionSet.availability	
InstructionSet.pipeline_clear	InstructionSet.pipeline_lock
Parameter.configuration	Parameter.initialization
Parameter.frequency	Parameter.phase
Parameter.timing	
Parameter.data_width	
Parameter.generate_control	
Parameter.enable	
MiscellaneousAsset.component	
MiscellaneousAsset.unknown	
MiscellaneousAsset.unused	

It is necessary to establish a metric for assets needing reassignment. The process of using asset reassignment strategies can be seen in Figure 3. For this metric, the same characteristic from both the Functionality GRL and unknown IP is used. First, only asset sets belonging to one characteristic are considered, and those characteristics must have external assets. Internal assets are disregarded because a significant portion of them are automatically assigned

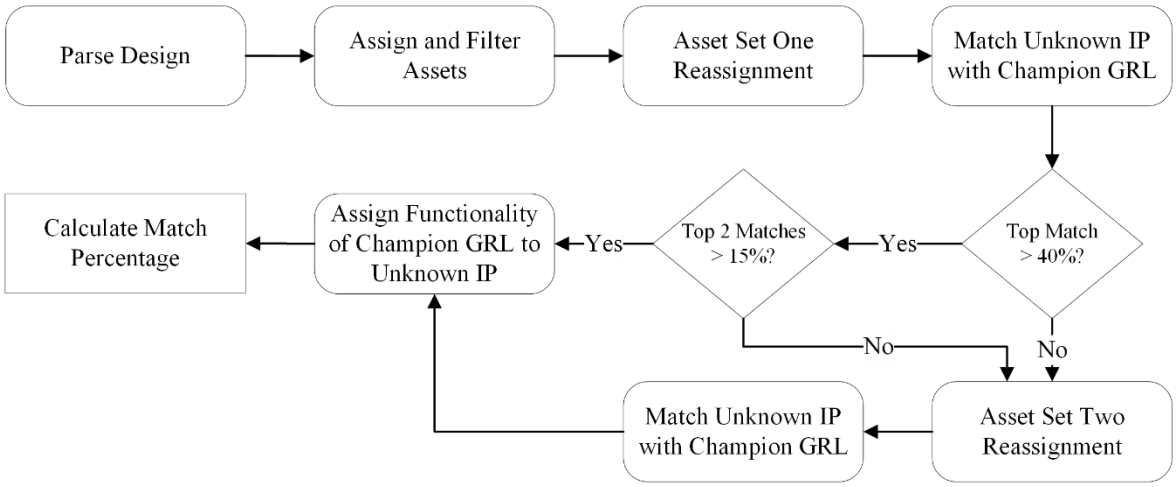

Figure 3: Reassignment Process Overview

during the parsing phase of the process. Next, all Functionality GRL entries are iterated over, receiving asset sets from the same characteristic as the unknown soft IP. These assets from the Functionality GRL are compared against the unknown IP's assets, and only the Functionality GRL entry's assets are reassigned because the unknown IP features up-to-date asset assignments. Therefore, only asset sets within the same characteristic are reassigned. In the event both designs have a similar and more applicable asset, frequency analysis is employed to determine whether the original or similar assets are the better fit.

3.5.3 Statistical Matching

Statistical matching techniques from [10] are employed to calculate the matching percentages used in Sections 3.5.1 and 3.5.2. However, these techniques have been modified, and their usages are different. In general, within the overall percentage match calculation, a new approach of dynamic weighting is employed.

$$Overall\ \%\ Match = \frac{\sum_{i=A}^{N} \%\ Match_i}{N}$$

Equation 2: Percent Match Equation with Dynamic Weighting

To implement this dynamic weighting, Equation 2 is used. In this equation, the characteristics used are denoted from A to N. For the dynamic weighting approach, a characteristic is ignored to increase matching percentage if neither design has the characteristic. This ensures the yielded matching percentage is always on a 100% scale rather than being artificially lowered if characteristics are not present. Determining weight for each characteristic leverages the same calculation used in [10] and is seen in Equation 3.

$$P(Asset) = \frac{\sum_{i=1}^{n} Asset \in Entry_i}{n}$$

Equation 3: Asset Probability Calculation from [10]

Equation 3 is used to determine the weight of any characteristics relevant to dynamic weighting. In this case, n is the number of GRL entries, and the summation is incremented by 1 for each GRL entry that contains the asset. This sum is then divided by the number of GRL entries to yield the probability of a GRL entry contains the asset. After calculating the probability of each asset, the weights of relevant assets are calculated.

$$W_{Asset} = 1 - P(Asset)$$

Equation 4: Asset Weight Calculation from [10]

The weight of an asset is determined by the probability that it will not be contained in a GRL entry and its calculation can be seen in Equation 4. An uncommon asset will have a high weight because it is more uniquely identifying while a common asset will have a low weight since many GRL entries have it, making the common asset a less unique identifier. After the calculation of asset weights and probabilities, it is possible to leverage the results of these calculations to determine average asset weight.

$$Avg\ W_{Asset} = \frac{\sum_{i=1}^{n} W_{MatchedAsset_i}}{N}$$

Equation 5: Average Asset Weight Calculation from [10]

In Equation 5, the average weight of a particular asset is equal to the sum of all asset weights in a matched characteristic, and this sum is divided by N, the total number of matched assets. If an asset has a higher weight than other assets, it is less common in the GRL and more

identifying than other assets, making high-weight assets more important for matching purposes. These results are used to calculate the final characteristic weight in Equation 6.

$$W_{Char} = \frac{Char_{AvgW_{Asset}}}{\sum_{i=A}^{N} i_{AvgW_{Asset}}} * 100$$

Equation 6: Characteristic Weight Calculation from [10]

For this equation, the weight of a characteristic is equal to the average asset weight of characteristic $Char$ divided by the summation of the average asset weight of all characteristics. In this case, i iterates through the dynamically-weighted characteristics relevant to the entries from A to N. Additionally, this quotient is multiplied by 100 to convert it into a percent contribution of each $Char$ to the total average asset weight from all dynamically weighted characteristics. To optimize the efficiency of Equations 3, 4, 5, and 6, asset frequency data is precalculated and stored with each GRL entry. This can be seen in the *asset_frequency_data* key in the GRL format example in Figure 2. By doing the calculation in advance, it is no longer necessary to determine asset probabilities and weights for all GRL entries in an iterative manner.

3.5.4 Revised Matching Process

The combination of Champion GRL Matching, FGRLM, and Statistical Matching as the overall matching process has been revised to allow for improved matching. After selecting a GRL entry to match against, the first step in calculating a design match is to find all unkonwn asset sets

with a 100% match to GRL asset sets within a given characteristic and add them to the final match data. Match data is represented as a dictionary where the first-level mapping is each asset

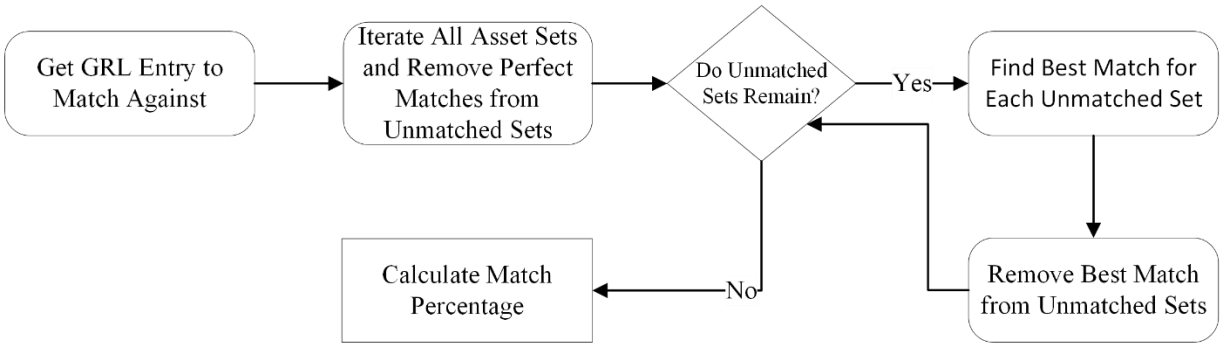

Figure 4. Asset Set Matching Overview

characteristic. Within each characteristic, another level of the dictionary relates a signal's name to its best GRL match along with the number of sets from that match. Here, the number of sets refers to the bit-width of the signal such that there is one instance of a given asset set for each bit of a signal. All bit-level calculations are done at the parsing stage. After removing all 100% match data, a check is performed to see if all asset sets have been matched between the two designs. If match data remains, then best-fit matching is performed on the remaining asset sets. This is done by iterating through all unmatched GRL asset sets within the same characteristic. Each GRL asset set is matched against all unknown asset sets within the same characteristic, with its best match being initialized to the highest-matching unknown characteristic. After setting this match, all other GRL asset sets are checked to ensure the unknown asset set does not match more highly than the current best match. In the event that another GRL asset set matches better to the unknown asset set in question, then the current GRL asset set has its best match changed to the next highest match and the verification process starts again. If a GRL asset set has the same percentage match as the current best match, the number of asset sets is checked and the match that is closer to this number is chosen instead. In the event that all unknown asset sets have a higher match with other GRL asset sets,

the GRL asset set is temporarily skipped. Finally, once a best match is found, it is added to the best match data and removed from the matched assets sets. Then, the remaining unmatched data is updated, and this process is continued for the next GRL asset set. This process of removing matches from unmatched data is detailed in Figure 4. Any characteristic containing asset sets in one design but not in the other is given a 0% match and added to the final match data. Throughought the entire process, characteristic matching data is cached to avoid unnecessary recalculations and to maintain match data after each asset set match is performed.

4 RESULTS AND ANALYSIS

When comparing the original and new matching processes, it is important to highlight the differences in matching functionalities as well as GRL differences addressed in Section 3.3. Notably, the updated GRL is significantly larger and includes bias towards clean designs. Both Functionality GRLs do not contain an even distribution of entries across all functionalities and may lead to discrepancies with functionalities such as *clock_generation* having no entries at the time of testing. These issues and others are explained in the following subsections. All functionalities provided in the following tables regarding the new matching process are clean, Trojan-free functionalities unless otherwise specified. The matching percentage data shown for each matching process is calculated from different metrics and is not directly comparable.

4.1 BUS INTERFACE

The bus interface design is composed of a microcontroller containing ROM, SPRAM, LED outputs, and a UART communication module. This is a significantly large design and includes several hundred component instances. External assets are assigned to the primary port signals of each component as well as the top *Bus_Interface_Top* module. In the original matching process, it was not possible for asset filtering to fully define signals to subcomponents because each unique subcomponent was expressed as a single instance instead of multiple. For instance, if there were 100 instantiations of an SPRAM module, it only considered one of them. The asset filtering step has been improved in the new matching process because the filtering is fully expressed throughout all subcomponents and their instances, thereby offering improved results. The percentages in Table 8 below differ from the baseline results in [10] because a newer bus interface design was utilized.

Table 8. Bus Interface Matching Results

	Original Matching Process		New Matching Process	
Component	**Functionality**	**Match**	**Functionality**	**Match**
Bus_Interface_Top	COMMUNICATION	15.0%	control_unit	8.4%
osch	COMPUTATIONAL	28.0%	computational	11.3%
PLL_Clk	COMPUTATIONAL	12.0%	datapath	8.3%
vlo	COMPUTATIONAL	54.0%	computational	12.5%
ehxpllj	COMPUTATIONAL	24.0%	control_unit	19.4%
Bus_Master	COMMUNICATION	44.0%	memory	16.7%
SPRAM	COMMUNICATION	39.0%	memory	19.9%
inv	COMMUNICATION	13.0%	memory	19.9%
rom16x1a	COMMUNICATION	13.0%	memory	40.6%
vhi	COMMUNICATION	9.0%	memory	4.0%
fd1p3dx	COMMUNICATION	27.0%	memory	30.3%
mux321	COMMUNICATION	20.0%	datapath	19.1%
spr16x4c	COMMUNICATION	32.0%	memory	18.6%
RS232_Usr_Int	COMMUNICATION	44.0%	communication	12.2%
STD_FIFO	COMMUNICATION	41.0%	memory	27.9%
Bus_Int	COMMUNICATION	41.0%	datapath	24.2%
Std_Counter	COMMUNICATION	29.0%	datapath	23.3%
LED_Ctrl	COMMUNICATION	35.0%	datapath	10.9%
PWM_16b	COMMUNICATION	33.0%	debugging	14.7%

Regarding the original matching process results, almost all components are categorized as *COMMUNICATION*. This can be attributed to asset filtering adding many assets to each signal in large designs, making classification more difficult with the increased volume of information. However, the new matching process fully expresses all subcomponents. In the case of *Bus_Master*, there are several hundred memory-related components which leads to most functionalities being expressed as memory despite not necessarily being memory. The full expression of all subcomponents has caused memory asset saturation due to the number of memory modules utilized in the bus interface design. This will need to be addressed in the future but shows improvements in the representation of designs in the new matching process. Additionally, a larger number of

functionalities are shown during matching instead of having an abnormally high number of *COMMUNICATION* functionalities assigned.

4.2 PS/2 KEYBOARD CONTROLLER

The PS/2 Keyboard Controller is IP consisting of a top-level *Ps2_keyboard* component and two lower-level *debounce* components, and it facilitates communication between a computer and a user's keyboard. The results of the old and new matching processes on the unknown PS/2 Keyboard Controller IP are presented in Table 9. This data is slightly different than the baseline data in [10] because previous data does not contain matching percentages for the *debounce* subcomponent.

Table 9. PS/2 Keyboard Controller Matching Results

	Original Matching Process		New Matching Process	
Component	**Functionality**	**Match**	**Functionality**	**Match**
Ps2_keyboard	PERIPHERAL	100.0%	control_unit	21.7%
debounce	COMMUNICATION	63.0%	communication	9.5%

Above, *Ps2_keyboard* was labeled *PERPHERAL* functionality by the original matching process and *control_unit* functionality by the new matching process. Of these two, *control_unit* is more indicative of the true functionality of the PS/2 Keyboard Controller. Both matching processes assign *communication* functionality to the *debounce* component; however, it would be more properly identified by the *peripheral* functionality. This discrepancy is caused by a lack of *peripheral* functionality components in both the former and updated GRLs.

4.3 LCD16×2 DISPLAY CONTROLLER

The LCD16×2 Display Controller is a single component consisting of a large vector input for each line of the two-line LCD displays used on some Xilinx evaluation boards. It features two 128-bit vector inputs where data inputs correspond to one of the two lines of the LCD display. The

matching results for an unknown LCD16×2 Display Controller are shown in Table 10. For this example, the resulting functionality match for the new and original matching processes are similar.

Table 10. LCD16×2 Display Controller Matching Results

	Original Matching Process		New Matching Process	
Component	**Functionality**	**Match**	**Functionality**	**Match**
lcd16x2	PERIPHERAL	75.0%	peripheral	20.2%

The original matching process assigns a *PERIPHERAL* functionality with a 75% match while the new matching process also assigns a *peripheral* functionality but with a smaller 20.2% match. Since both matching processes arrive at the proper functionality, both are correct for this design. However, the new matching process yields a smaller percent match due to the total match being distributed across several subcomponents.

4.4 BASIC RSA-T200

The Basic RSA-T200 design is a smaller design consisting of a denial-of-service Trojan in the *RSACypher* component. This Trojan disables encoding at the transmitter and decoding at the receiver. In Table 11 below, the results from identifying the functionalities of the unknown Basic RSA-T200 IP are exhibited for the new matching process alongside data from [10] for comparison.

Table 11. Basic RSA-T200

	Data From [10]		New Matching Process	
Component	**Functionality**	**Match**	**Functionality**	**Match**
RSACypher	TROJAN_ENCRYPTION_UNIT	83.2%	communication	17.0%
Modmult	COMPUTATIONAL	100.0%	computational	18.6%

The original matching process assigned *TROJAN_ENCRYPTION_UNIT* to *RSACypher* and *COMPUTATIONAL* to *Modmult*. The new matching process assigns *communication* to *RSACypher* and *computational* to *Modmult*. An explanation for the discrepancy in *RSACypher* identification is the limited number of Trojan designs in the Functionality GRL which leads to a

lack of similar designs for matching. For *Modmult*, both matching processes arrive at equivalent functionalities, but the new matching process has a reduced match percentage due to subcomponent matching.

5 CONCLUSION AND FUTURE WORK

Rebuilding the codebase of the Structural Checking Tool and standardizing several aspects of the matching and parsing processes have resulted in a more organized tool which allows for streamlined development and increased matching accuracy. The matching and parsing processes are more lightweight and optimized, focusing on current statistical analysis and Champion and Functionality GRL Matching. Additionally, the user experience of the Structural Checking Tool has been improved, and the time required to prepare HDL code for parsing within the tool has been significantly reduced. Issues pertaining to parseable VHDL syntax have also been addressed, allowing more designs to be usable out-of-the-box without necessitating changes. These improvements should allow the Functionality GRL to grow at a faster rate which will improve matching as a result.

While the new structure of the tool is more modular and easier to expand, there are still many opportunities for improvement. The size of the Functionality GRL will always need to increase, and there is an ongoing effort to find the most representative designs for each functionality to use in the Champion GRL. However, the matching processes will need further optimization to improve execution performance as GRL size increases. Additionally, there is room to expand the parsing functionality to encompass other HDL languages and to more accurately express the structure of designs with improved syntax coverage.

6 REFERENCES

[1] R. Shende and D. D. Ambawade, "A side channel based power analysis technique for hardware trojan detection using statistical learning approach," *2016 Thirteenth International Conference on Wireless and Optical Communications Networks (WOCN)*, 2016, pp. 1-4, doi: 10.1109/WOCN.2016.7759894.

[2] T. Hu, L. Wu, X. Zhang, Y. Yin and Y. Yang, "Hardware Trojan Detection Combine with Machine Learning: an SVM-based Detection Approach," *2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2019, pp. 202-206, doi: 10.1109/ICASID.2019.8924992.

[3] Yousra Alkabani and Farinaz Koushanfar. 2009. Consistency-based characterization for IC Trojan detection. In *Proceedings of the 2009 International Conference on Computer-Aided Design* (*ICCAD '09*). Association for Computing Machinery, New York, NY, USA, 123–127. DOI:https://doi.org/10.1145/1687399.1687426

[4] T. Inoue, K. Hasegawa, M. Yanagisawa and N. Togawa, "Designing hardware trojans and their detection based on a SVM-based approach," *2017 IEEE 12th International Conference on ASIC (ASICON)*, 2017, pp. 811-814, doi: 10.1109/ASICON.2017.8252600.

[5] T. Inoue, K. Hasegawa, Y. Kobayashi, M. Yanagisawa and N. Togawa, "Designing Subspecies of Hardware Trojans and Their Detection Using Neural Network Approach," *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 2018, pp. 1-4, doi: 10.1109/ICCE-Berlin.2018.8576247.

[6] M. Fyrbiak *et al*., "HAL—The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion," in *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 498-510, 1 May-June 2019, doi: 10.1109/TDSC.2018.2812183.

[7] Jiaji He, Haocheng Ma, Yanjiang Liu, and Yiqiang Zhao. 2020. Golden Chip-Free Trojan Detection Leveraging Trojan Trigger's Side-Channel Fingerprinting. *ACM Trans. Embed. Comput. Syst.* 20, 1, Article 6 (January 2021), 18 pages. DOI:https://doi.org/10.1145/3419105

[8] H. S. Choo *et al*., "Machine-Learning-Based Multiple Abstraction-Level Detection of Hardware Trojan Inserted at Register-Transfer Level," *2019 IEEE 28th Asian Test Symposium (ATS)*, 2019, pp. 98-980, doi: 10.1109/ATS47505.2019.00018.

[9] L. Weaver, T. Le and J. Di, "Golden Reference Library Matching of Structural Checking for securing soft IPs," *SoutheastCon 2016*, 2016, pp. 1-7, doi: 10.1109/SECON.2016.7506737.

[10] N. Waller, H. Nauman, D. Taylor, R. Del Carmen and J. Di, "Character Reassignment for Hardware Trojan Detection," *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2021, pp. 861-864, doi: 10.1109/MWSCAS47672.2021.9531813.

[11] M. Hinds, J. Brady, and J. Di, "Signal Assets - a Useful Concept for Abstracting Circuit Functionality," presented at the Government Microcircuit Applications & Critical Technology Conference (GOMACTech), 2013.

[12] T. Le, J. Di, M. Tehranipoor, and L. Wang, "Tracking data flow at gate-level through structural," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, 2016, pp. 185-189.

[13] J. Yust, M. Hinds, and J. Di, "Structural Checking: Detecting Malicious Logic without a Golden Reference," *Journal of Computational Intelligence and Electronic Systems*, vol. 1, no. 2, p. 8, 2012.

[14] B. McGeehan, F. Smith, T. Le, H. Nauman and J. Di, "Hardware IP Classification through Weighted Characteristics," *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2019, pp. 1-6, doi: 10.1109/HPEC.2019.8916225

[15] H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development", IEEE Int. Conference on Computer Design (ICCD), 2013.

[16] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, M. Tehranipoor, “Benchmarking of Hardware Trojans and Maliciously Affected Circuits”, Journal of Hardware and Systems Security (HaSS), April 2017.

[17] *OpenCores*. Available: http://opencores.org/