Pre-silicon Formal Verification of JTAG Instruction Opcodes for Security

Nicole Fern
ECE Department
University of California Santa Barbara, USA
Email: nicole@ece.ucsb.edu

Kwang-Ting (Tim) Cheng
Hong Kong University of
Science and Technology, Hong Kong
Email: timcheng@ust.hk

***Abstract*—Widely implemented standards such as IEEE 1149.1 (JTAG) and 1687 (iJTAG) are essential in providing improved chip and board testability, but it has been demonstrated that undocumented or poorly obfuscated scan and debug instructions can be exploited by hackers to undermine system security. Prior work proposes adding authentication or encryption to JTAG to improve security, but these methods can only protect functionality known to the design and test team. Out-of-spec JTAG functionality can be inserted accidentally or with malicious intent (e.g. hardware Trojans). Our proposed technique can detect anomalous JTAG instructions not present in the specification using commercial formal equivalence checking tools. We demonstrate the effectiveness of our technique by characterizing the entire JTAG instruction set space for the OpenSPARC T2 benchmark in a completely automated manner. In the original design our technique formally proves all undefined opcodes map to the benign bypass instruction and provides the size and location within the design hierarchy of all data registers. In a modified version of the design our technique correctly detects several undefined opcodes that are used to access the L2 cache, as well as extra out-of-spec elements in a data register selected by an existing instruction.**

I. Introduction

IEEE 1149.1, commonly referred to as JTAG (an abbreviation for Joint Test Action Group) was originally created to standardize the interface and functionality of boundary scan architectures to facilitate board-level testing [1]. It has been widely adopted and the flexibility built into the standard has allowed debug and test functionality beyond board test to be provided through the JTAG interface. JTAG can be used to control built-in-self-test (BIST), observe internal design registers for debugging, and even write firmware images.

At its inception, JTAG was assumed to be used only for test and debug purposes, offering no security measures other than obscurity. Hackers soon realized that JTAG provides access to powerful hidden test and debug commands and have been able to successfully reverse engineer obfuscated and undocumented JTAG implementations to gain unintended system access (e.g. load unsigned code [2], recover encryption keys [3], and perform memory forensics during device operation [4]).

Discovering the set of available test and debug commands through exhaustive exploration of every possible instruction opcode is key to attacking system security via JTAG [5]. Figure 1 shows a block diagram of on-chip JTAG circuitry. The Test Access Port (TAP) controller state machine directs the updating, capturing, and shifting data in/out of various registers based on the JTAG interface inputs. JTAG registers are chains of scan flip-flops and can be divided into an Instruction Register (IR) and Data Registers (DRs). In IEEE 1149.1, an opcode loaded into the instruction register selects a specific data register (e.g. the bypass or boundary scan register). An attacker will explore every instruction opcode to discover data registers that leak information or provide the ability to modify design state.

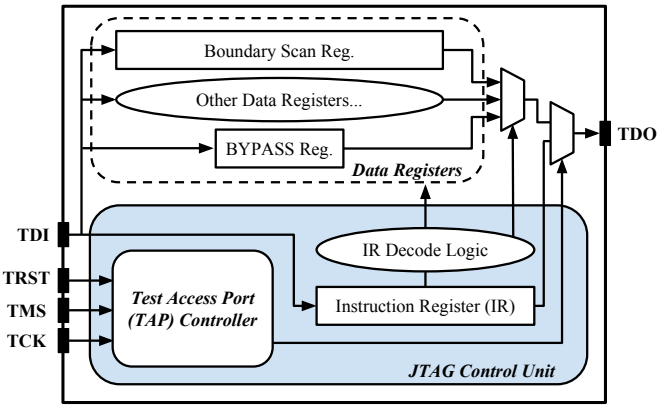

Fig. 1. JTAG Circuitry

To prevent an attacker from discovering invasive JTAG commands, disabling/removing the test interface before deployment and adding access control mechanisms or circuitry monitoring for malicious access patterns have been proposed [6], [7]. There also exist several pre-silicon verification techniques specifically targeting test and debug circuitry [8], [9], [10]. These methods are presented in more detail in Sections II-B and II-C, but it should be noted that to the best of our knowledge there are no methods which analyze JTAG circuitry to identify *extra* anomalous instruction opcodes or data registers. Existing pre-silicon verification techniques for JTAG focus on verifying the correctness of the set of instructions and data registers listed in the specification (but do not identify extra circuitry outside the spec), while prior defense techniques assume the set of implemented instructions is correct and focus on restricting usage of this functionality by adding extra protection circuitry.

978-1-5386-8382-8/18/$31.00

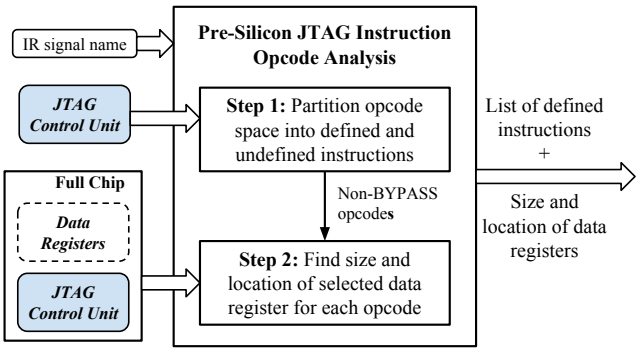

Fig. 2. Overview of Proposed JTAG Instruction Set Analysis Technique

Threat Model: Our technique identifies anomalous JTAG instructions and data registers in the design before tape-out. Anomalous instructions include those implemented using opcodes that are undefined/unused according to the specification. Anomalous data registers are those that are a different size than outlined in the specification, or contain scan cells from locations in the design that are unexpected given the purpose of the data register. These out-of-spec commands may be included by accident due to misinterpretation of the specification, miscommunication between different design teams, or inserted with malicious intent.

The increased controllability and observability JTAG provides through pins exposed at the board level makes it an attractive target for malicious modification. Such modifications, known as hardware Trojans, can be inserted by any party with access to the design [11]. Hundreds of engineers and a wide array of 3rd party design tools have access to the design before fabrication and the ability to insert additional JTAG functionality not listed in the written specification or Boundary Scan Description Language (BSDL) file. Unused opcodes provide an opportunity to add secret backdoors into a chip even if access control mechanisms protect a subset of privileged instructions because its impossible to protect functionality you don't know about.

Regardless of their source, out-of-spec commands have the potential to be abused by an attacker if they are not discovered and either removed or protected adequately using an existing access control technique before tape-out. Automated formal characterization of the *entire* JTAG instruction space, including both specified and undefined/unused opcodes, is the main feature of our proposed method and can be employed by the design verification team to increase confidence that the JTAG functionality is Trojan-free and bug-free. Our technique extracts knowledge about implemented functionality directly from the design and is not limited to analyzing only known instructions making it ideal for addressing Trojans hiding in unused instruction opcodes. By clearly listing the number and nature of implemented JTAG instructions our technique enables automated comparison against the BSDL file, useful for regression testing, as well as easy manual checking against the written specification.

Method Overview: Figure 2 provides an overview of the proposed method. First, the set of opcodes that do not map to the bypass instruction are identified using formal logic equivalence checking. If an instruction is formally proven to be identical to bypass it is benign and does not require further analysis. The second step is to extract properties of the data registers corresponding to the instructions found in the first step. The analysis output is 1) a list of instructions which are not identical to bypass, and 2) the size of each instruction's corresponding data register along with the design modules where scan cells in the data register reside and a list of signals used to implement the instruction. This information can be verified against the specification.

Despite the use of formal methods, our technique is scalable to large SoC designs because we perform logic equivalence checking between two nearly identical circuits, which can be solved efficiently for industry-scale designs by commercial tools such as Cadence Conformal [12]. Moreover, our method is highly automated, only requiring the identification of the instruction register signal. Because it is a pre-silicon verification technique, no additional circuitry is required to improve design security, making our technique applicable to low-cost embedded designs.

The main contributions of our work are the following:

- A highly automated pre-silicon formal verification technique capable of detecting both Trojans hiding within undefined instruction opcodes and accidentally included out-of-spec functionality
- Use of commercial formal equivalence checking tools for analysis making the proposed technique scalable to industry SoC designs
- Demonstration of the potential of our technique to detect maliciously inserted out-of-spec JTAG functionality in the OpenSPARC T2 SoC, which contains a rich JTAG instruction set

The rest of the paper is organized as follows: Section II explores related work in attacking, defending, and verifying JTAG, Section III presents our instruction characterization technique in a tool agnostic manner while Section IV details how to implement our technique using Cadence Conformal. Our technique is demonstrated on an OpenSPARC T2 SoC design in Section V and in Section VI we conclude.

II. Related Work

A. Attacking JTAG

Known techniques for attacking a system through the JTAG interface fall into the following categories:

1) Reverse engineering obfuscated JTAG functionality to gain unintended system access (e.g. [5], [13], [2], [4])
2) Analyzing scan chain data to recover encryption keys (e.g. [3], [14], [15], [16])
3) Inserting hardware Trojans to attack JTAG [17]

In Category 1, the attacker has extremely limited knowledge about the debug and test infrastructure. The main security mechanism protecting the test infrastructure in many consumer

electronic products is obfuscation of both the location of the test interface pins on the PCB board and the set of implemented JTAG commands. To identify the JTAG interface there are push-button hardware solutions available for purchase such as JTAGulator [18] as well as projects such as JTAGenum [19], which provide code to leverage popular development boards (e.g. Arduino) to find the test interface.

Once the JTAG interface has been found, the attacker will identify the length of the instruction register (IR) then exhaustively explore the instruction opcode space to identify any useful undocumented commands using the techniques presented in [5]. Such undocumented commands have played a central role in gaining unauthorized read/write access to a bitstream in a military grade FPGA [13], running unsigned software on the Xbox 360 [2], and performing online forensics on SDRAM and Flash memory in embedded systems [4].

Attacks targeting encryption key recovery (Category 2) using scan chain data have been actively researched [3]. These attacks combine scan chain data with knowledge about properties and common implementations of specific encryption algorithms to extract the key despite the fact that the signals selected for scan and their ordering in the scan chain are unknown to the attacker. Scan chain attacks have been successfully applied to AES [14], ECC [15], and RSA [16], and often work even in the presence of partial scan, test response compaction, and X-masking.

Category 3 assumes the attacker has the ability to insert or modify circuitry in a subset of chips sharing a boundary scan chain. In [17], Trojan-infected chip(s) on the boundary scan path drive JTAG control signals to non-infected (victim) chip(s) in order to snoop sensitive data between the tester and victim chip(s) or alter test responses. The electrical drive strength of the tester and Trojan-infected chip determines attack feasibility. Trojans modifying the JTAG controller circuitry itself are not considered.

In this work, we propose and address the threat of Trojans implementing malicious functionality using undefined JTAG instruction opcodes. By formally finding all implemented instructions and their associated data registers our technique can detect these Trojans as well as find any instructions unintentionally left in the design. Any extra functionality implemented using undefined opcodes can potentially be discoverable by hackers using Category 1 attack methods if not identified using our technique and either removed or protected by access control mechanisms.

B. Securing JTAG

Given the number and successfulness of attacks outlined in Section II-A, other mechanisms besides security through obscurity are necessary to protect system test and debug circuitry. Chen et. al. give an overview of existing defense techniques, which range from disabling the test and debug interface via fuses before product deployment to adding authentication-based access control and data encryption [6]. Completely disabling test/debug pins prevents most attacks through the JTAG port, but removes the opportunity for in-field failure diagnosis, configuration, and updates. Protection schemes such as locking iJTAG segment insertion bits (SIBs) [20] and scrambling scan chain data or ordering (e.g. [21]) using a key are more lightweight than full blown authentication schemes (e.g. [22], [17]), which require hardware implementations of cryptographic primitives, but are vulnerable to replay attacks.

For all key-based defense schemes, key management is an issue, and there is the risk of compromise if every chip has the same programmed key. [7] takes a different approach and observes that authorized users know the functionality of JTAG instructions beforehand whereas attackers exhaustively explore different configurations, and these access patterns can be differentiated during device operation via machine learning using special circuitry to implement the classifier.

In relation to these existing techniques, this work provides a different form of security. We prove the absence of instructions not listed in the specification, and increase confidence in the correctness of the implemented functionality, while other defense techniques assume this already and focus on restricting access to the implemented test features by adding additional circuitry. Our technique should be applied alongside JTAG protection circuitry for designs that can afford the addition of defense circuitry, but since our verification technique does not increase silicon area or circuit design time it can also easily be applied to low-cost embedded systems where protection circuitry is not viable.

C. Verifying JTAG

There are several works which focus on pre-silicon verification methods for JTAG. [8] uses formal methods to verify access control properties for restricted JTAG registers, but requires the set of restricted registers to be known ahead of time whereas our proposed technique does not make any assumptions about the instructions or data registers and performs a complete characterization of the space. One possible application of our method is to use the identified set of data registers as input to the method detailed in [8].

The pre-silicon JTAG verification tool presented in [9] checks compliance with the 1149.1 standard and the BSDL file for the design, but states in a footnote that "while the tool can verify instruction opcodes of arbitrary length, the tool does not verify that each unspecified instruction opcode selects the bypass register." For verification of IEEE 1687 (iJTAG) circuitry [10] proposes a technique to verify functionality at SoC level using IP-level Instrument Connectivity Language (ICL) and Procedural Description Language (PDL) files, but it is not clear if the technique can identify out-of-spec instructions not present in the ICL or PDL files.

III. Instruction Set Characterization

As shown in Figure 2, our methodology consists of 2 steps:

1) Identifying instruction opcodes whose behavior is not identical to the bypass instruction
2) Extracting data register characteristics

In most JTAG implementations unused/undefined opcodes are required to map to the bypass instruction. The bypass

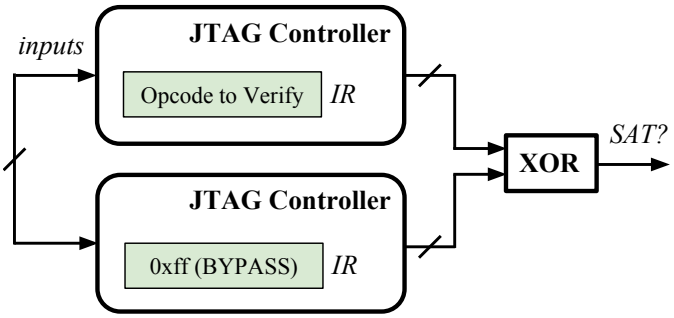

Fig. 3. Pairwise Opcode Comparison Equivalence Checking Formulation

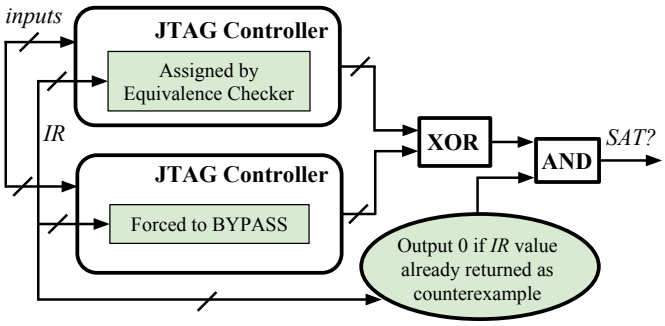

Fig. 4. Identifying Non-BYPASS Opcodes Using Counterexample Exclusion

instruction selects a single bit data register which passes Test Data In (TDI) directly to Test Data Out (TDO), as seen in Figure 1. The bypass instruction is benign because the internal design state is not revealed or modified. One assumption our technique makes is that the bypass instruction is implemented correctly and does not contain hidden functionality. By formally proving different opcodes map to this instruction, we can conclude that no hardware Trojans or out-of-spec instructions are implemented using these opcodes.

Step 2 analyses the set of non-bypass opcodes, found in Step 1, and for each opcode produces a list containing the full hierarchical path name of signals used to implement the instruction. These signals include control bits and data register signals, and can be used to find all data registers in the JTAG implementation. This signal list can be used to quickly identify any anomalies such as too many or too few scan flip-flops in a given data register or flip-flops located in a portion of the design unrelated to the instruction's functionality.

A. Instruction Opcode Space Analysis

Opcode space characterization only requires analyzing the JTAG Controller module (circuitry bounded by the blue box in Figure 1), not the entire design. The control signals which select a data register to connect to the TDI and TDO pins, and the signals from the TAP Controller directing capture, shift, and update operations are all outputs of the JTAG control unit. Any opcodes formally proven to produce JTAG controller outputs identical to the bypass instruction under all possible inputs can't possible produce differences outside the module.

Formally proving two opcodes implement identical functionality can be formulated generally as a satisfiability problem, and more specifically as an equivalence checking problem. Figure 3 shows the basic formulation. Two copies of the JTAG controller are created, and in one copy the instruction register is hard-coded to the bypass instruction opcode, and in the other the IR is assigned the opcode under verification. If the two design versions are proven to be equivalent (the output of the XOR unsatisfiable), then the opcode under verification maps to bypass.

Complexity: Pairwise comparison of every possible IR value against the bypass opcode requires $2^n - 1$ calls to the equivalence checker, where n is the number of bits in the instruction register. Although the JTAG instruction register is typically much smaller than the IR in a general purpose processor architecture, the complexity can still be prohibitive. The number of defined opcodes is typically less than the number of unused opcodes. For example, the OpenSPARC T2 JTAG instruction register is 8 bits, meaning there is a total of 256 possible opcodes, but only 86 map to unique instructions. This observation can be used to overcome the complexity of pairwise comparison.

Formulation Using Counterexample Exclusion: An alternate formulation is given in Figure 4. Instead of fixing the output of the instruction register inside the JTAG controller module, the IR is made a primary input in both design versions, however in one version the output of the IR is still masked with the bypass opcode while in the other version the IR remains a free input and can be assigned by the equivalence checking tool when searching for counterexamples.

Counterexamples are patterns of primary input assignments which produce different outputs in the two design versions and are used to prove the two designs are not equivalent. In Figure 4, the first call to the equivalence checker will return a counterexample which includes a value for the instruction register. This value could be the opcode for IDCODE, EXTEST, or any other instruction that does not map to bypass. To avoid the equivalence checker returning the exact same opcode during the second call, a constraint is added to the output (the green oval in Figure 4) preventing the designs from registering as non-equivalent under already analyzed opcodes. One more opcode is added to this set after each call to the equivalence checker until all non-bypass opcodes have been analyzed and the designs are equivalent. Using this strategy the number of calls to the equivalence checker is equal to the number of defined instructions, which is typically much smaller than 2^n.

In comparing the complexity of the two formulations, the final factor to take into account is opportunity for parallelization. Each comparison in the pairwise formulation shown in Figure 3 is independent and there is no limit to the number of comparisons that can be run simultaneously. If 255 opcodes need to be compared against the bypass opcode, the analysis can be run on 255 parallel threads, cores, or

machines. The counterexample exclusion strategy (Figure 4) can't be parallelized as the results of the current comparison are constrained by the results of previous comparisons.

B. Data Register Characterization

In combinational equivalence checking the sequential behavior of the design is not taken into account. Any state elements in the design are cut, meaning flip-flop and latch inputs become pseudo-primary outputs (PPOs), and outputs become pseudo-primary inputs (PPIs). During the pairwise comparison of two non-equivalent opcodes, the equivalence checking tool will return a list of outputs (which include PPOs) present in both designs, called *key points*, that differ. When one of the opcodes is bypass, this signal list consists of input signals to scan flip-flops in the data register selected by the non-bypass opcode and any additional registers or control signals related to the non-bypass instruction.

Because data register logic is not guaranteed to reside entirely within the JTAG control unit, the entire design must be analyzed using the pairwise equivalence checking formulation shown in Figure 3 in order to identify these non-equivalent key points. While full chip analysis is significantly more complex than analyzing the JTAG control unit, it only has to be performed for opcodes which do not map to bypass. Additionally, since the instruction register is the only signal altered, the two design versions being compared are nearly identical, which is precisely the case commercial logic equivalence checking tools are optimized for as the primary application for equivalence checking technology is to check conformance of a synthesized design with the original RTL code.

C. Extension to IEEE 1687 (iJTAG)

Because of the variety of functionality accessible through the JTAG interface, another standard, IEEE 1687 (iJTAG) [23] exists to allow dynamic configuration of the scan network using data registers instead of requiring a fixed instruction set. This is accomplished through the use of Segment Insertion Bits (SIBs), which when set expand the 1687 gateway data register to include different functionality, referred to as test instruments. The 1687 gateway register is accessed like a typical 1149.1 data register and can be used to access the hierarchy of test instruments. Because scan chain configuration is no longer dependent only on the value of the instruction register, the space which must be explored by our verification method has to include SIBs in addition to bits in the instruction register. This has the potential to significantly increase the complexity of the technique and may be an ideal application for the counterexample exclusion formulation.

IV. Adaption to Conformal LEC

While any equivalence checking tool can be used to perform our analysis method, Cadence Conformal Logic Equivalence Checker (LEC) [12] is widely used in industry for verification of logic synthesis transformations. The robust front-end Verilog and VHDL parser, structural design analysis capabilities, and rich diagnosis features make LEC an ideal tool for the full chip analysis necessary to characterize JTAG functionality.

```
1  /// Mark modules which should be black-boxed
2  add notranslate modules [module names] -design -Both
3
4  /// Read same design for both Golden and Revised
5  read design [list of design files] -Verilog -Golden
       -root [top module] -define [macro]=[value]  ...
6  read design [list of design files] -Verilog -Revised
       -root [top module] -define [macro]=[value] ...
7
8  /// Make IR a primary input in both Golden and
       Revised
9  set system mode setup
10 add primary input instr[*] -golden
11 add primary input instr[*] -revised
12
13 /// Constrain IR (hard-code to specific op-code)
14 /// Golden: 0xff, Revised: 0x0
15 set system mode setup
16 delete pin constraints -all_pin -module [top module]
       -golden
17 delete pin constraints -all_pin -module [top module]
       -revised
18 add pin constraints 1 instr[7] instr[6] instr[5]
       instr[4] instr[3] instr[2] instr[1] instr[0] -
       module [top module] -golden
19 add pin constraints 0 instr[7] instr[6] instr[5]
       instr[4] instr[3] instr[2] instr[1] instr[0] -
       module [top module] -revised
20
21 ///Perform Equivalence Checking
22 set system mode lec
23 add compared points -all
24 compare
25
26 ///Run Diagnosis
27 diagnose -noneq
```

Fig. 5. Example LEC Dofile for JTAG Analysis

In the previous section, we presented a pairwise equivalence checking formulation (Figure 3) for opcode space analysis and data register characterization, as well as a formulation involving counterexample exclusion (Figure 4) for opcode space analysis. This section provides the LEC commands used to implement both analysis strategies in an efficient manner by walking through an example command file run by LEC (referred to as a "dofile") shown in Figure 5.

Loading the Design: Lines 1-6 in the dofile specify how to read and elaborate the design. LEC allows macro definition and black-boxing specific modules (e.g. memories) to control the configuration of the design and which portions are analyzed. For JTAG instruction set characterization, the exact same design is read for both the Golden and Revised designs.

Constraining IR: While Figure 3 shows the instruction register hard-coded in the Golden and Revised designs, it is more efficient to make IR a primary input then constrain the input to two different values. This removes the need to augment the RTL code itself and load the entire design into LEC for every opcode pair comparison. Lines 1-11 are only run before the first comparison, which when performing full chip analysis saves a significant amount of time. The commands in Lines 10-11 cut the design to make the instruction register signal a primary input in both design versions, and Lines 16-19 constrain individual bits in the IR signal to specific values.

Equivalence Checking and Diagnosis: In Lines 22-24 the equivalence checking of all corresponding *key points* in Golden and Revised designs is performed. Key points in LEC are all primary outputs, flip-flops, latches, black-box, and cut points present in the design. LEC will match key points in the Golden design with those in the Revised design (in our case the designs are nearly identical) then check the equivalence of the logic cone for each key point. Non-equivalent key points can be analyzed using the `diagnose` command (shown in Line 27). This command will list the full hierarchical signal names of the non-equivalent key points which is useful for data register characterization.

Examples of the report produced by the `diagnose` command can be seen in Figures 8, 9, and 10. Because the format is parsable, the information provided by the report can be further condensed and transformed to be more human-readable to aid in manual verification. If the documentation specifying data register length and location within the design hierarchy is also parsable the verification check can be completely automated for use in regression testing.

Counterexample Exclusion: Constructing the formulation involving counterexample exclusion (Figure 4) using LEC requires the ability to ignore equivalence checking results under certain conditions. The `$constraint` function is an undocumented LEC feature which forces LEC to ignore counterexamples that do not satisfy the constraint. For example, inserting `$constraint(IR != 8'b0 && IR != 8'b1)` in the design source code forces LEC to ignore counterexamples where the IR opcode is 0 or 1. Unfortunately the constraint function must be placed with the source code and the design must be reloaded when new constraints are added.

V. OpenSPARC T2 Experiment

OpenSPARC T2, a full SoC design that has been included in commercial products [24], is an ideal benchmark to evaluate our technique because it contains a rich JTAG instruction set. Figure 6 provides a block diagram of the full SoC architecture. For the full chip analysis in this experiment we select the option to instantiate only 1 SPARC Core instead of 8 and black-box the L2 cache banks and the memory control units (MCUs). The analyzed design contains 683 primary inputs, 389 primary outputs, and 257152 state elements (flip-flops and latches). The JTAG controller is contained within the Test Control Unit (TCU). The JTAG instruction register is 8 bits, meaning there is a total of 256 possible opcodes, but according to the specification [25] only 86 map to unique instructions.

Table I summarizes the results of performing opcode analysis and data register characterization on the OpenSPARC T2 design. Out of the 256 opcodes analyzed, 171 map to BYPASS. With the exception of a single opcode (discussed further in Section V-A), the list of undefined opcodes matches the specification exactly. The pairwise opcode comparison strategy is used, and even without parallelization only takes a little over a minute to complete. Analyzing the 84 non-bypass opcodes to extract data register characteristics requires the entire SoC design and takes over a day. The reports produced by this effort (detailed further in Section V-B) can be easily interpreted by an engineer who is not familiar with the debug logic for manual comparison against the written specification or used as input to further automated analysis comparing the extracted information to the BSDL file.

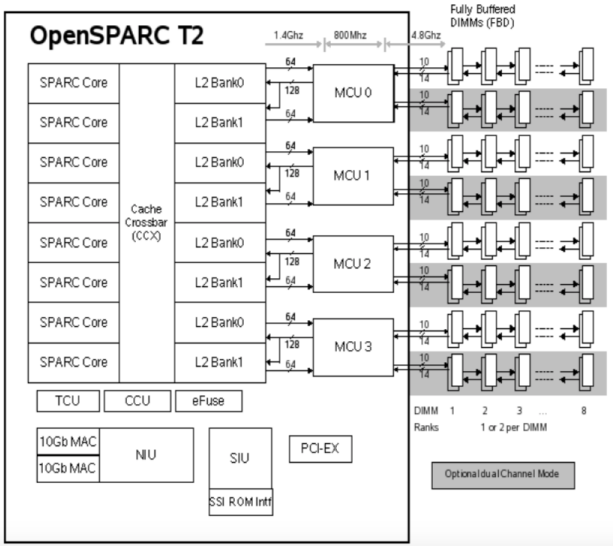

Fig. 6. OpenSPARC T2 SoC Architecture [25]

TABLE I
OpenSPARC T2 Experiment Results Summary

Experiment Description	Top-level Module	# Opcodes Analyzed	Analysis Time
Opcode Analysis	JTAG Controller	256	61.73 sec
Data Reg. Characterization	Entire SoC	84	31.4 hrs

Trojan Insertion: In addition to analyzing the original design, we insert additional out-of-spec functionality to illustrate how our verification technique can highlight Trojans in the JTAG circuitry. The modifications of the design include:

1) Mapping undefined opcodes 0xa5, 0xa6, 0xa7, and 0xa8 to the TAP_L2_ADDR, TAP_L2_WRDATA, TAP_L2_WR, and TAP_L2_RD instructions
2) Adding 32 extra bits to the IDCODE data register

The motivation behind mapping undefined opcodes to existing instructions is circumvention of access control circuitry. The instructions chosen allow complete access to the L2 cache. If the test and debug circuitry is protected presumably these instructions will not be accessible to an unprivileged user. By assigning "shadow" opcodes to implement this functionality, an attacker knowledgable about this Trojan or able to find the Trojan opcodes through exhaustive exploration will still be able to access the L2 cache through JTAG even though the original opcodes have been marked as protected functionality.

Increasing the length of the IDCODE data register highlights the ability of our technique to identify extra functionality in existing instructions. The IDCODE register is read-only and contains a number used to identify the device and manufac-

TROJAN-FREE DESIGN	
Op-code(s)	Equivalent to BYPASS?
0x00-0x06	N
0x07	Y
0x08-0x0a	N
0x0b	Y
0x0c-0x10	N
0x11-0x12	Y
0x13-0x16	N
0x17	Y
0x18-0x1f	N
0x20-0x27	Y
0x28-0x38	N
0x39-0x3f	Y
0x40-0x46	N
0x47	Y
0x48-0x4a	N
0x4b	Y
0x4c-0x52	N
0x53-0x57	Y
0x58-0x5b	N
0x5c-0x5f	Y
0x60-0x65	N
0x66-0x7f	Y
0x80-0x84	N
0x85-0x87	Y
0x88-0x89	N
0x8a-0x8f	Y
0x90	N
0x91-0x9f	Y
0xa0-0xa4	**N**
0xa5-0xff	**Y**
172 out of 256 opcodes EQUIVALENT to BYPASS. **84** out of 256 opcodes NOT EQUIV. to BYPASS.	

TROJAN-INFECTED DESIGN	
Op-code(s)	Equivalent to BYPASS?
0x00-0x06	N
0x07	Y
0x08-0x0a	N
0x0b	Y
0x0c-0x10	N
0x11-0x12	Y
0x13-0x16	N
0x17	Y
0x18-0x1f	N
0x20-0x27	Y
0x28-0x38	N
0x39-0x3f	Y
0x40-0x46	N
0x47	Y
0x48-0x4a	N
0x4b	Y
0x4c-0x52	N
0x53-0x57	Y
0x58-0x5b	N
0x5c-0x5f	Y
0x60-0x65	N
0x66-0x7f	Y
0x80-0x84	N
0x85-0x87	Y
0x88-0x89	N
0x8a-0x8f	Y
0x90	N
0x91-0x9f	Y
0xa0-0xa8	**N**
0xa9-0xff	**Y**
168 out of 256 opcodes EQUIVALENT to BYPASS. **88** out of 256 opcodes NOT EQUIV. to BYPASS.	

Fig. 7. Opcode Maps for Trojan-free and Trojan-infected OpenSPARC T2 JTAG Instructions (differences highlighted using bold font)

```
1  Opcode: 0x8 (TAP_CREG_ADDR)
2  # NEq: PO=0  DFF=43  DLAT=0  BBOX=0  CUT=0  Total=43
3
4  DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
5  DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
6  DFF /tcu/jtag_ctl/tap_cregaddr_shift_reg/d0_0/q_reg[39]
7  DFF /tcu/jtag_ctl/tap_cregaddr_shift_reg/d0_0/q_reg[38]
8  ...
9  DFF /tcu/jtag_ctl/tap_cregaddr_shift_reg/d0_0/q_reg[0]
10 DFF /tcu/jtag_ctl/tap_creg_addr_en_reg/d0_0/q_reg[0]
11
12 Opcode: 0x9 (TAP_CREG_WDATA)
13 # NEq: PO=0  DFF=67  DLAT=0  BBOX=0  CUT=0  Total=67
14
15 DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
16 DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
17 DFF /tcu/jtag_ctl/tap_cregwdata_shift_reg/d0_0/q_reg[63]
18 DFF /tcu/jtag_ctl/tap_cregwdata_shift_reg/d0_0/q_reg[62]
19 ...
20 DFF /tcu/jtag_ctl/tap_cregwdata_shift_reg/d0_0/q_reg[0]
21 DFF /tcu/jtag_ctl/tap_creg_data_en_reg/d0_0/q_reg[0]
22
23 Opcode: 0xa (TAP_CREG_RDATA)
24 # NEq: PO=0  DFF=67  DLAT=0  BBOX=0  CUT=0  Total=67
25
26 DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
27 DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
28 DFF /tcu/jtag_ctl/tap_cregrdrtrn_shift_reg/d0_0/q_reg[64]
29 DFF /tcu/jtag_ctl/tap_cregrdrtrn_shift_reg/d0_0/q_reg[63]
30 ...
31 DFF /tcu/jtag_ctl/tap_cregrdrtrn_shift_reg/d0_0/q_reg[0]
```

Fig. 8. Data Register Characterization Report for Register Debug Instructions

turer. In OpenSPARC this register is 32 bits, but the inserted Trojan extends this register to be 64 bits. Possible uses for the extra 32 bits include leaking information from the design normally not accessible via scan.

A. Instruction Opcode Space Analysis

Instruction opcode space analysis uses the JTAG controller (`tcu_jtag_ctl`) as the top-level module. Comparison of every opcode against 0xff (BYPASS) is accomplished using Cadence Conformal LEC commands similar to the example dofile in Figure 5. Our technique summarizes the comparison results by producing opcode maps, shown in Figure 7, which can be quickly checked against the JTAG instruction summary table in Section 4.2.3 of the specification [25].

In the Trojan-free original design all instructions equivalent to bypass are undefined in the specification except for opcode 0x91, which implements the TAP_STCI_CLEAR instruction. Even though this instruction is defined, the specification states it "clears STCI mode for SERDES Test Configuration Interface Bus," and also mentions that "to clear JTAG access to STCI, use TAP_STCI_CLEAR or reset the TAP state machine" meaning simply mapping this instruction to bypass makes sense.

The undefined opcodes utilized by Trojan circuitry are successfully identified using our technique. The opcode map produced for the Trojan-infected design, shown on the right-hand side of Figure 7, states there are 88 opcodes not equivalent to bypass, while in the Trojan-free design there are only 84. The last 2 lines of the Trojan-infected design opcode map reveal the extra 4 opcodes are 0xa5-0xa8, which are precisely the opcodes chosen to implement shadow access instructions to the L2 cache.

B. Data Register Characterization

The 84 non-bypass opcodes identified in the Trojan-free design are analyzed using the pairwise comparison formulation shown in Figure 3. The entire SoC is analyzed to extract the list of key points differentiating the opcode under verification with the bypass instruction. Due to the complexity of analyzing the full chip design (logic for 235089 key points must be analyzed) the 84 opcodes are divided into 6 batches of 14 opcodes analyzed using 6 parallel instances of LEC on a Dell PowerEdge R730 machine running Ubuntu 16.04 containing 32GB of memory and 20 Intel Xeon CPU E5-2650 v3 cores. 6 is chosen as the number of parallel instances because each LEC process takes approximately 5GB of memory, putting the total memory utilization at 30GB.

Due to space limitations, the full results for all 84 opcodes in the Trojan-free design are not given, but instead a select few are presented in detail showing the information provided by our technique along with details for the 5 opcodes used to implement Trojans. Figure 8 shows a portion of the results file for 2 instructions in the Trojan-free design which select JTAG data registers containing the address and write data for

```
1  Opcode: 0x01 (TAP_IDCODE)
2  # NEq: PO=0  DFF=66  DLAT=0  BBOX=0  CUT=0  Total=66
3
4  DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
5  DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
6  DFF /tcu/jtag_ctl/tap_idcode_reg/d0_0/q_reg[63]
7  DFF /tcu/jtag_ctl/tap_idcode_reg/d0_0/q_reg[62]
8  ...
9  DFF /tcu/jtag_ctl/tap_idcode_reg/d0_0/q_reg[0]
```

Fig. 9. Data Register Characterization Report for Trojan-infected Design: IDCODE Instruction

accessing design configuration registers for debugging through the Non-Cacheable Unit (NCU) (a bus connecting various SoC components). The full hierarchical signal names and type (D flip-flop, latch, black box, etc.) of the non-equivalent key points are provided using the `diagnose` command in LEC.

Analysis results for the TAP_CREG_ADDR instruction (Lines 1-10 in Figure 8) show that a total of 43 key points are non-equivalent to bypass. Some of the hierarchical signals paths are abbreviated for compact formatting. The first 2 points (Lines 4-5) are signals related to the bypass instruction, which are expected to differ when comparing against a non-bypass opcode. The next 40 points (Lines 6-9) correspond to the 40-bit CREG Address data register. A list of JTAG data registers can be found in Section 4.2.4, Table 4-4 of [25]. The final non-equivalent point (Line 10) is an enable bit, presumably used to place the address on the NCU interconnect after the address bits have been scanned into the CREG Address register.

Results for the TAP_CREG_WDATA instruction (Lines 12-21) are similar, except that the CREG Write_Data register is 64 bits instead of 40 bits. The length of the CREG Write_Data register is clearly shown by our analysis as well as the exact location of the register in the design hierarchy. The TAP_CREG_RDATA opcode results (Lines 23-31) reveal that the CREG Read_Data register is 65 bits. The specification states that one of the bits is used as a sentinel to indicate the availability of the read data, explaining why the write and read data registers are not the same size.

Trojan-infected Design Analysis Results: Figure 9 shows the analysis results the TAP_IDCODE instruction. The inserted Trojan extends this register from 32 to 64 bits and this is clearly reflected from just a quick glance at the results. Opcodes 0xa5-0xa8 are undefined in the original design but map to existing L2 cache access instructions in the Trojan-infected version. The signals identified in Figure 10 point to the L2 cache as the target of the modification. Opcodes 0xa5 and 0xa6 implement instructions to load an address and write data into JTAG registers which access the L2 cache. Opcode 0xa7 initiates the write operation, and opcode 0xa8 both initiates the read operation and captures the data read from the cache into a JTAG data register.

In addition to clearly detecting functionality implemented using undefined opcodes, the analysis results for these L2 cache access instructions illustrate the potential of our technique to verify properties of JTAG instructions already present in the specification (the analysis of the original opcodes for these instructions would yield an identical report). Our analysis reveals that the address, write data, and read data registers are all implemented using the same shift register (Lines 5-8, Lines 16-19, and Lines 35-38 in Figure 10 refer to the same 65 bit signal), however the control signals (Lines 9, 20, and 39-40) are different between the instructions.

```
1  Opcode: 0xa5 (Mapped to TAP_L2_ADDR)
2  # NEq: PO=0  DFF=67  DLAT=0  BBOX=0  CUT=0  Total=67
3  DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
4  DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
5  DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[64]
6  DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[63]
7  ...
8  DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[1]
9  DFF /tcu/regs_ctl/tcuregs_l2addrupd_syncreg/xx0/q_reg
10
11 Opcode: 0xa6 (Mapped to TAP_L2_WRDATA)
12 # NEq: PO=0  DFF=67  DLAT=0  BBOX=0  CUT=0  Total=67
13
14 DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
15 DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
16 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[64]
17 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[63]
18 ...
19 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[1]
20 DFF /tcu/regs_ctl/tcuregs_l2dataupd_syncreg/xx0/q_reg
21
22 Opcode: 0xa7 (Mapped to  TAP_L2_WR)
23 # NEq: PO=0  DFF=4  DLAT=0  BBOX=0  CUT=0  Total=4
24
25 DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
26 DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
27 DFF /tcu/regs_ctl/tcuregs_l2vld_syncreg/xx0/q_reg
28 DFF /tcu/regs_ctl/tcuregs_l2wr_syncreg/xx0/q_reg
29
30 Opcode: 0xa8 (Mapped to TAP_L2_RD)
31 # NEq: PO=0  DFF=69  DLAT=0  BBOX=0  CUT=0  Total=69
32
33 DFF /tcu/jtag_ctl/.../bypass_reg/d0_0/q_reg[0]
34 DFF /tcu/jtag_ctl/.../bypass_ll_reg/d0_0/q_reg[0]
35 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[64]
36 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[63]
37 ...
38 DFF /tcu/jtag_ctl/tap_l2access_shift_reg/d0_0/q_reg[0]
39 DFF /tcu/regs_ctl/tcuregs_l2vld_syncreg/xx0/q_reg
40 DFF /tcu/regs_ctl/tcuregs_l2rd_syncreg/xx0/q_reg
```

Fig. 10. Data Register Characterization Report for Trojan-infected Design: Opcodes 0xa5-0xa8 (Mapped to L2-cache Access Instructions)

Interestingly these registers are all 65 bits, but the least significant bit (`q_reg[0]`) does not appear in the results for the address and write data registers. After consulting the data register table (Table 4-4 in Section 4.2.4 of [25]) it is revealed that the least significant bit is not used in the address and write data registers but is used by the read data register to indicate when data in bits 64:1 are valid. Although this seemingly anomalous functionality turned out to be within spec, the ease with which our results can be manually scanned for irregularities demonstrates how our technique can be harnessed for verification in addition to Trojan detection.

This unique use of LEC diagnosis information enables fast verification of the data register set and can be further processed for automated comparison against the BSDL file. Suspicious signals are easy to identify as most instructions select data registers comprised of signals clustered in a few modules. Any additional signals, added accidentally or malicious intent can be removed before tape-out.

VI. Conclusion

A scalable automated method is presented to analyze JTAG circuitry pre-silicon in order to identify out-of-spec instructions with the potential to pose security risks to the system. This paper is the first to focus on identifying extra test and debug instructions not present in the specification as well as verifying the correctness of already specified functionality. In addition to providing a tool agnostic verification formulation for JTAG instruction set characterization, we detail how existing commercial logic equivalence checking tools can be leveraged to ensure efficient chip-level analysis for industry scale designs. We demonstrate the effectiveness of our technique by performing complete verification of the OpenSPARC T2 JTAG instruction opcode space. Our technique identifies the set of defined opcodes by formally proving these opcodes are not equivalent to the bypass instruction, and provides information about data registers and control logic corresponding to each defined instruction. In the original Trojan-free OpenSPARC design, our results verify the opcode space matches the specification. Our method also successfully detects malicious functionality implemented using undefined opcodes as well as the modification of an existing data register in a Trojan-infected version of the design.

VII. Acknowledgements

This work was supported by NSF/SRC STARSS (1526695) and a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKUST 16207917).

References

[1] "IEEE standard for test access port and boundary-scan architecture," *IEEE Std 1149.1-2013*, pp. 1–444, May 2013.

[2] SMC Xbox Hack. [Online]. Available: http://free60.org/wiki/SMC_Hack

[3] J. Da Rolt, A. Das, G. Di Natale, M.-L. Flottes, B. Rouzeyre, and I. Verbauwhede, "Test versus security: past and present," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 50–62, March 2014.

[4] M. Breeuwsma, "Forensic imaging of embedded systems using JTAG (boundary-scan)," *Digital Investigation*, vol. 3, no. 1, pp. 32–42, 2006.

[5] F. Domke, "Blackbox JTAG reverse engineering," in *Proceedings of the 26th Chaos Communication Congress (CCC)*, 2009.

[6] W. Chen, J. Bhadra, and L.-C. Wang, "SoC security and debug," in *Fundamentals of IP and SoC Security: Design, Verification, and Debug*, S. Bhunia, S. Ray, and S. Sur-Kolay, Eds. Springer, Janurary 2017, ch. 3, pp. 29–48.

[7] X. Ren, V. G. Tavares, and R. Blanton, "Detection of illegitimate access to JTAG via statistical learning in chip," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 109–114.

[8] M. A. Kochte, R. Baranowski, M. Sauer, B. Becker, and H. J. Wunderlich, "Formal verification of secure reconfigurable scan network infrastructure," in *Proceedings of the 21th IEEE European Test Symposium (ETS)*, May 2016, pp. 1–6.

[9] K. Melocco, H. Arora, P. Setlak, G. Kunselman, and S. Mardhani, "A comprehensive approach to assessing and analyzing 1149.1 test logic," in *Proceedings of the 2003 International Test Conference (ITC)*, vol. 2, September 2003, pp. 40–49.

[10] T. Payakapan, S. Kan, K. Pham, K. Yang, J. F. Cote, M. Keim, and J. Dworak, "A case study: leverage IEEE 1687 based method to automate modeling, verification, and test access for embedded instruments in a server processor," in *Proceedings of the 2015 IEEE International Test Conference (ITC)*, October 2015, pp. 1–10.

[11] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, May 2016.

[12] Cadence. Conformal Equivalence Checker. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/equivalence-checking/conformal-equivalence-checker.html

[13] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip." in *Procedings of the 2012 Conference on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, E. Prouff and P. Schaumont, Eds., vol. 7428. Springer, 2012, pp. 23–40.

[14] S. S. Ali, O. Sinanoglu, S. M. Saeed, and R. Karri, "New scan-based attack using only the test mode," in *Proceedings of the 21st IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, October 2013, pp. 234–239.

[15] J. Da Rolt, A. Das, G. Di Natale, M.-L. Flottes, B. Rouzeyre, and I. Verbauwhede, "A scan-based attack on elliptic curve cryptosystems in presence of industrial design-for-testability structures," in *Proceedings of the 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2012, pp. 43–48.

[16] J. Da Rolt, A. Das, G. Di Natale, M.-L. Flottes, B. Rouzeyre, and I. Verbauwhede, "A new scan attack on rsa in presence of industrial countermeasures," in *Proceedings of the 2012 International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2012, pp. 89–104.

[17] K. Rosenfeld and R. Karri, "Attacks and defenses for JTAG," *IEEE Design & Test of Computers*, vol. 27, no. 1, 2010.

[18] Grand Idea Studio. JTAGulator. [Online]. Available: http://www.grandideastudio.com/jtagulator

[19] JTAGenum. [Online]. Available: https://github.com/cyphunk/JTAGenum

[20] A. Zygmontowicz, J. Dworak, A. Crouch, and J. Potter, "Making it harder to unlock an LSIB: Honeytraps and misdirection in a P1687 network," in *Proceedings of the 2014 Conference on Design, Automation & Test in Europe (DATE)*, 2014, pp. 195–201.

[21] J. Lee, M. Tehranipoor, C. Patel, and J. Plusquellic, "Securing designs against scan-based side-channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 325–336, Oct 2007.

[22] C. Clark, "Anti-tamper JTAG TAP design enables DRM to JTAG registers and P1687 on-chip instruments," in *Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2010, pp. 19–24.

[23] "IEEE standard for access and control of instrumentation embedded within a semiconductor device," *IEEE Std 1687-2014*, pp. 1–283, Dec 2014.

[24] Oracle. OpenSPARC T2. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html

[25] *OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification*, Revision A ed., Sun Microsystems, Inc., May 2008.