Effective Pre-Silicon Verification of Processor Cores by Breaking the Bounds of Symbolic Quick Error Detection

Abstract

Existing techniques to detect logic design bugs or hardware Trojans in processor cores during pre-silicon verification face severe challenges. We extend Symbolic Quick Error Detection, a recent bounded model checking-based logic bug detection technique, with symbolic starting states. Using open-source RISC-V cores, we demonstrate: 1. Quick (≤5 minutes (≤2.5 hours) for an in-order (out-of-order superscalar) core) detection of 100% of hundreds of logic bug and hardware Trojan scenarios from commercial chips and research literature, and 97.9% of "extremal" bugs (randomly-generated bugs requiring ~100,000 activation instructions taken from random test programs). 2. Quick (~1 minute) detection of several previously unknown bugs in open-source RISC-V designs.

1. Introduction

Pre-silicon verification requires major effort in a typical design flow [Foster 15]. This paper focuses on pre-silicon verification of single processor cores, which are critical components of any System-on-Chip (SoC). Generally, pre-silicon verification mainly targets logic design errors (*logic bugs*). Future pre-silicon verification techniques might also need to detect Hardware Trojans (*HTs*) [King 08], which are unauthorized modifications of an integrated circuit (*IC*), resulting in incorrect functionality and/or the exposure of sensitive data [Karri 10]. While initial research on HTs focused on attacks implemented during fabrication, there is growing concern about HTs being inserted in third-party Intellectual Property (*IP*) cores by malicious sources [Zhang 11].

Similar to logic bugs, we consider HTs that affect functionality; i.e., such an HT can cause an error that can (ultimately) create an error in the software-visible state (software-visible registers or memory). This encompasses many catastrophic attacks on processor cores [King 08].

Symbolic Quick Error Detection (QED) [Singh 18] is a new pre-silicon verification technique based on ideas from QED [Lin 14]. It uses Bounded Model Checking (BMC) [Clarke 01] for formal analysis of the design. It is an automatic and fast bug detection and localization technique (Appendix B). Symbolic QED analyzes a design symbolically, but it requires a concrete (non-symbolic, i.e., with 0s and 1s specified explicitly) starting state. This means, to find bugs or HTs that require very long activation sequences (i.e., many instructions are required to activate such bugs), Symbolic QED must rely on very deep BMC runs (i.e., runs that unroll the circuit far enough to include all the activation instructions). This may be difficult for practical designs. For example, [Singh 18] states that a BMC tool might only unroll for up to around 30 clock cycles, within 24 hours of verification time. The following HT example [Zhang 14], shows that Symbolic QED, while highly effective for many logic bugs, can be insufficient for detecting HTs.

Motivating Example. Consider the following HT that may be difficult to find using existing HT detection techniques: *The HT changes opcodes of the next several decoded instructions after it sees a specific sequence of 256 instructions*. This HT could inject a short instruction sequence to bypass physical memory protection and run a privileged instruction. Such privilege escalation attacks [King 08] can be catastrophic.

Because the HT requires many instructions (and many clock cycles) for activation, Symbolic QED (like other BMC-based methods [Rajendran 15, Reece 16]) fails to detect the example, unless the starting state quickly transitions to the state where the HT activates. Stumbling upon such a "close" state by starting at a concrete state (from simulation) or a reset state is highly unlikely to succeed, since the HT could be designed with an arbitrary activation sequence that is not known *a priori*.

To overcome this major challenge, we extend Symbolic QED so that it is now capable of starting from a symbolic (instead of concrete) starting state (i.e., we give the BMC tool the ability to choose an arbitrary starting state). As explained in the next paragraph, BMC from a symbolic starting state can be highly challenging. Our approach overcomes the associated challenges and allows us to start BMC-based Symbolic QED from states that would otherwise require very deep BMC runs that aren't supported by current formal tools. As a result, logic bugs and HTs that require very long activation sequences can now be detected by applying a very short sequence of instructions after starting from a "close" state.

Existing BMC-based pre-silicon verification techniques (including existing Symbolic QED) face the following challenge when used in conjunction with symbolic starting states: BMC needs to check a property and may require property-specific constraints on the symbolic starting state. These constraints are necessary to prevent the BMC tool from producing spurious counter-examples (false positives) for the property it checks. If the BMC tool selects a starting state that is not reachable from the set of all valid reset states of the system using a valid sequence of instructions, a false positive might occur. For example, assume that each word in a memory is protected with a single even parity bit. Assume that a BMC tool is asked to check the following property: *for any sequence of reads and writes to the memory, the parity bits remain consistent with the data.* If the starting state of the design is not constrained, the BMC tool can initialize the memory to contain an all-zero word with a '1' for the parity bit, read from this location, and report this spurious counterexample (which is a false positive). Traditional methods rely on verification engineers to (manually) create constraints to rule out such false positives, which can be extremely time-consuming for practical designs (with many complex properties).

This paper overcomes the above challenge by: i) defining *QED constraints*: sufficient constraints to ensure false positives do not occur when using Symbolic QED with symbolic starting states in a single processor core; and ii) introducing *QED recorders,* which observe a small subset of internal signals within the processor to ensure the QED constraints are satisfied. QED recorders are used for pre-silicon verification only. They do not incur area overhead for the final design.

Our simulation results demonstrate:

1) We automatically, correctly, and quickly (~1 minute) detect several previously unknown (real) logic bugs in open-source out-of-order (OoO) superscalar [Ridecore1], and in-order scalar [Vscale] cores.
2) We automatically, correctly, and quickly (within 25 seconds for in-order, 18 minutes for OoO) detect 100% of (117 in-order, 120 OoO) simulated logic bugs, representing a wide variety of "difficult" bug scenarios (Appendix A) in commercial designs. Symbolic QED with concrete starting state detects only 33% (in-order) and 5% (OoO).
3) We automatically, correctly, and quickly (within 5 minutes for an in-order core; 2 hours for an OoO superscalar core) detected 100% of (156 in-order, 195 OoO) simulated HTs, representing a wide variety of scenarios (Appendix A) in HT research literature. Symbolic QED with concrete starting state detects 15% (in-order) and 9% (OoO).
4) We automatically, correctly, and quickly (within 2.5 hours) detected 97.9% of an "extremal" bug family (randomly-generated pre-condition-based bugs which require ~100,000 activation instructions taken from random test programs) in an OoO superscalar core. In contrast, Symbolic QED with a concrete starting state detected 0%.

The following are some of the important features of our technique:

1) It is highly effective for detecting both logic bugs and HTs (despite long activation sequences) during pre-silicon verification of in-order and OoO superscalar cores, as demonstrated by our results.
2) It does not require the verification engineer to manually craft design-specific assertions/constraints to detect logic bugs or HTs.
3) No false positives, as demonstrated by our results.
4) It does not require a "golden" model or simulation data of the design under test for detection of logic bugs and/or HTs.
5) Its effectiveness does not depend on the way HTs are designed, i.e., our method is HT-design agnostic.

The rest of this paper is organized as follows. Sec. 2 describes Symbolic QED with symbolic starting states. Results are presented in Sec. 3, followed by related work in Sec. 4. Sec. 5 concludes this paper.

2. Extending Symbolic QED with Symbolic Starting States

In this section, we discuss how Symbolic QED [Singh 18] is extended with symbolic starting states (background on QED and Symbolic QED in

Appendix B). To avoid false positives, we define a set of constraints (*QED constraints*) on the symbolic starting state (Sec. 2.1). To implement the QED constraints, we introduce *QED recorders*, additional hardware modules (used only for pre-silicon verification—not required for the final design) that record a small subset of internal logic values of the processor core to ensure that the QED constraints are satisfied (Sec 2.2). Fig. 1 contrasts Symbolic QED with/without symbolic starting state.

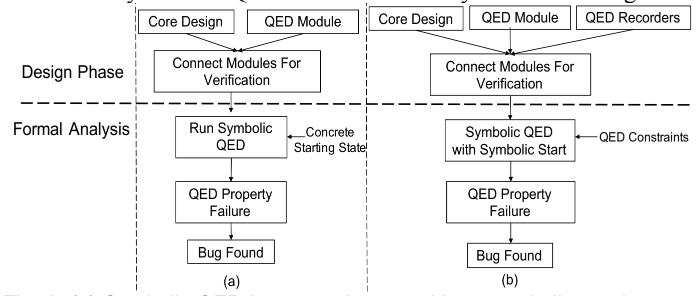

Fig. 1. (a) Symbolic QED inputs and steps without symbolic starting state, and (b) Symbolic QED inputs and steps with symbolic starting state.

2.1. QED Constraints

We first define some terminology used in the constraint definitions: i) *Symbolic In-Flight (SIF) "instructions"*: symbols (i.e., state bits), part of the symbolic starting state (which will be assigned 0s and 1s by the BMC tool), corresponding to (microarchitectural) flip-flops within the pipeline that hold instructions during normal operation of the core[1]; ii) T_C: the point in time when all SIF instructions commit (i.e., write to architectural state) - it is determined by the BMC tool; iii) *Symbolic QED instructions*: symbols which represent the instructions that form the bug trace (which is part of the counter-example, along with the starting state that BMC assigns) generated by the BMC tool after T_C; and iv) *Symbolic QED operand data:* symbols representing the operand[2] data of dispatched Symbolic QED instructions (dispatched before T_C)

Fig. 2 illustrates these definitions for a 3-stage in-order pipeline. When the formal analysis begins, there are up to 3 SIF instructions, and all commit by time T_C. The first Symbolic QED instruction (corresponding to *R1=R1+5*) is fetched into the pipeline, and its Symbolic QED operand data is available after the Dispatch stage.

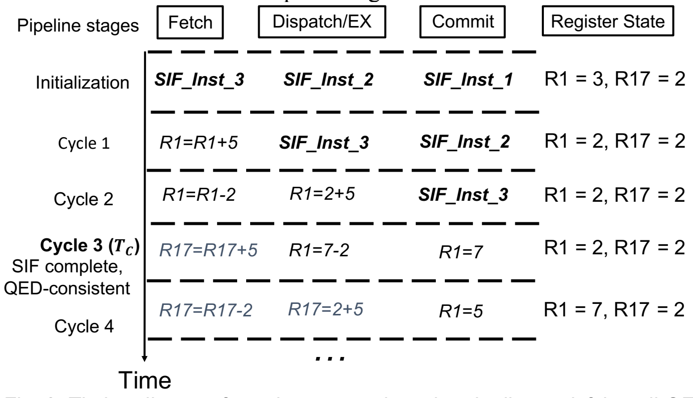

Fig. 2. Timing diagram for a three-stage in-order pipeline satisfying all QED constraints. SIF instructions commit by Tc, before all QED instructions.

Now, the QED constraints are given as follows ([Extended 18] further details how each constraint is enforced):

Constraint C-1: At T_C, all SIF instructions have committed (i.e., no SIF instruction can write to the architectural state after T_C), while all Symbolic QED instructions commit after T_C.

Constraint C-2: At T_C, the architectural state (program-visible registers and memory) is QED-consistent (Appendix B), and nothing but Symbolic QED instructions can write to architectural state after T_C (e.g., any test mode such as scan that can directly write to flip-flops must be disabled).

Constraint C-3: All the operand data for each Symbolic QED instruction I, must satisfy one of the following properties:

i) if the operand data is available (i.e., I has already read data for this operand) then this data must match the corresponding register/memory location (i.e., source operand location) data at T_C.

ii)[3] if the operand data is not available at T_C, then I is waiting for the result of an earlier Symbolic QED instruction for this operand data.

The QED constraints form a sufficient condition to ensure no false positives, given that bug-free designs satisfy two assumptions after T_C:

1. If a Symbolic QED instruction is executed twice on the same data, it results in the same value being stored to architectural state, e.g., Rx=1+2 and Ry=1+2 always result in the same values stored to registers Rx, Ry.
2. If a Symbolic QED instruction has a read-after-write dependency with earlier instructions, it uses the most recent value of the data in its computation. For example, in the program {R1=5; R2=R1+2; R3=R2-2}, the second instruction uses value '5' for R1, and the third instruction uses value '7' for R2.

The two assumptions are further detailed in [Extended 18].

2.2. Symbolic QED Recorders

QED recorders record a small number of internal signals in a design (to track T_C and Symbolic QED operands) so that we can specify the QED constraints to the BMC tool. For ease of understanding, we take an in-order core with single instruction fetch and 5-stage pipeline as a running example in Sec. 2.2, but we explain how the technique is generalized. In Sec. 3, we present results for both in-order and OoO superscalar cores.

2.2.1. Recorder for T_C

As T_C depends on the starting state chosen by the BMC tool, it cannot be statically determined before the formal analysis begins. A recorder is used to give this information to the tool dynamically. For an in-order core, T_C can be determined by simply tracking the progress of the first Symbolic QED instruction (the first symbolic instruction the BMC tool creates as part of the bug trace) until it reaches the commit stage (write-back stage) of the pipeline. At this time, all SIF instructions must have committed, as the pipeline is occupied by Symbolic QED instructions.

Specifics of the T_C recorder for a 5-stage, single-fetch, in-order pipeline is given in Fig. 3. Inputs are ready signals for all stages that precede the commit stage (e.g., *fetch_ready* is high when the fetch stage is ready to receive an instruction). The output *SIF_complete* is true when the first Symbolic QED instruction goes through all pipeline stages and reaches the commit stage. The output *mode* keeps track of progress made so far by the Symbolic QED instruction (we later make use of this output in Sec. 2.2.2). This T_C recorder for a 5-stage pipeline can be easily modified to support in-order pipelines with a different number of stages.

```
INPUT: fetch_ready, dispatch_ready, exec_ready, mem_ready
OUTPUT: SIF_complete, mode
// initialization
mode ← S0;   SIF_complete ← false;
// end initialization
if (mode == S0) && (fetch_ready) then
  mode ← S1;   SIF_complete ← false;  // instruction completes fetch stage
end if
if (mode == S1) && (dispatch_ready) then
  mode ← S2;   SIF_complete ← false;  // instruction completes decode stage
end if
if (mode == S2) && (exec_ready) then
  mode ← S3;   SIF_complete ← false;  // instruction completes execute stage
end if
if (mode == S3) && (mem_ready) then
  mode ← S4;   SIF_complete ← true;  // instruction completes memory stage
end if
```

Fig. 3. Pseudocode for T_C recorder.

For an OoO core, the T_C recorder is even simpler, and utilizes the reorder buffer (ROB). The idea is to mark the entry allocated in the ROB for the first Symbolic QED instruction. After this, *SIF_complete* is assigned true when the ROB head pointer reaches the marked instruction. For cores with no ROB, but OoO *commit* (e.g., [Aquarius]), an additional constraint is required (see Sec. 4).

2.2.2. Recorder for Symbolic QED Operands

Like T_C, Symbolic QED operands also depend on the starting state. The Symbolic QED operand recorder stores information for both register and memory operands. Specifics of the Symbolic QED operand recorder for a

5-stage, single-fetch, in-order pipeline is given in Fig. 4. Inputs are: 1) *_addr*, which gives register/memory address of the corresponding operand; 2) *_data*, which gives operand data; 3) *_valid*, which is high when *_addr* is valid and *_data* is valid; 4) *mode*, which gives the state of the T_C recorder (Fig. 3). Output *_buffer* stores all Symbolic QED operands and their values (buffer depth is determined by the maximum number of instructions in flight at a given time). We only store the information for Symbolic QED instruction operands in buffers i.e., we do not store operand information of any SIF instruction. This is enforced by checking T_C recorder state, i.e., *mode* (we do not add entries to *_buffer* until all SIF instructions pass through dispatch stage). In Fig. 4, we assume that each instruction requires at most two register values and one memory value, but the idea is easily extended to more source operands.

For an OoO core, Fig. 4 is extended to include Symbolic QED operands that are waiting on results of earlier Symbolic QED instructions. For each waiting operand, we also store the instruction tag (ROB entry number) of the instruction it is waiting for. This information is used to specify Constraint C-3 for an OoO core (details in [Extended 18]).

```
INPUT: src1_addr, src1_data, src1_valid, src2_addr, src2_data, src2_valid, mem_addr,
mem_data, mem_valid, mode
OUTPUT: src1_buffer, src2_buffer, mem_buffer
// initialization
src1_buffer ← empty_buffer;  src2_buffer ← empty_buffer;
mem_buffer ← empty_buffer;
// end initialization
if (mode != $S_0$) && (mode != $S_1$) && (src1_valid || src2_valid) then
  if (src1_valid) then
    src1_buffer.add_entry(src1_addr, src1_data);
  end if
  if (src2_valid) then
    src2_buffer.add_entry(src2_addr, src2_data);
  end if
end if
if ((mode == $S_3$) || (mode == $S_4$)) && (mem_valid) then
  mem_buffer.add_entry(mem_addr, mem_data);
end if
```

Fig. 4. Pseudocode for Symbolic QED operand recorder.

3. Results

We demonstrate the effectiveness of our new technique on two open-source RISC-V processor cores: i) V-scale [Vscale], an in-order core targeting embedded applications; and ii) RIDECORE [Ridecore1], an OoO superscalar core (2-way pipeline, 64 maximum instructions in-flight, 2 ALUs, 1 multiplier, 1 load/store unit) for high performance applications. For BMC, we used the Questa Formal tool (version 10.5c) from Mentor Graphics on an AMD Opteron 6438 with 128GB of RAM. For each core, we instrumented a new QED module (Appendix B.3) and the QED recorders and QED constraints (Sec. 2).

3.1. Previously Unknown Bugs

We first found three previously unknown logic bugs in the multiplier reservation station (RS-m) of the RIDECORE design (all three confirmed by RIDECORE designers [Ridecore2], see Table 1). Importantly, these bugs were detected due to our new QED module in this paper (Appendix B.3). The QED module of [Singh 18] does not detect these bugs. Our new QED model improves upon [Singh 18] by allowing arbitrary interleaving of original and duplicate instruction subsequences.

We also found two bugs in Vscale (Table 2), by running Symbolic QED starting at a concrete, power-on reset state in less than 40 seconds (also confirmed by designers). These bugs are due to errors in the Vscale implementation of the RISC-V privileged ISA [RISCVP], within specific Control Status Registers (CSRs). Importantly, Vscale does not implement shadows for CSRs. To circumvent this, Symbolic QED's EDDI-V transformation (Appendix B) duplicates instructions using data memory for each CSR. The first bug occurs because of incorrect design of the MIP register interrupt bit logic. After this bug was fixed, a second bug was found in the MSTATUS register.

3.2. "Long" Logic Bugs and HT Scenarios

We simulated 120 (117) logic bug types using RIDECORE (Vscale). These are mostly "longer" (up to 256 consecutive activation instructions) versions of "difficult" logic bug scenarios (Appendix A) that occurred in various commercial SoCs [Singh 18]. We also simulated 195(156) difficult HT scenarios (Appendix A) from research literature using RIDECORE (Vscale). Results are in Table 3.

Observation 1: Symbolic QED with symbolic starting states correctly and automatically found all "long" logic bugs, in less than 30 mins, with no false positives. It found bugs that traditional BMC methods fail to detect (including Symbolic QED). Symbolic QED with concrete starting state detected only 5% (33%) of these bugs in RIDECORE (Vscale).

Observation 2: Symbolic QED with symbolic starting states correctly and automatically found all injected HTs (including those designed to evade state-of-the-art HT detection techniques), in less than 2.5 hours, without requiring design-specific assertions or debug of false positives. Symbolic QED with a concrete starting state detected only 9% (15%) of these HTs in RIDECORE (Vscale).

Table 1. New bugs in RIDECORE. Symbolic QED runtimes.

Bug Activation	Bug Effect	Runtime (symbolic starting state)	Runtime (power-on reset starting state)
All but one (buggy entry) RS-m entries occupied; MULH[4] instr. assigned to vacant entry.	First source operand of MULH instruction corrupted.	25 minutes	63 seconds
Same as above.	Second source operand of MULH instruction corrupted.	61 minutes	69 seconds
Same as above, but MULHU[5] instr. assigned to vacant entry.	Result of MULHU instruction corrupted.	64 minutes	93 seconds

Table 2. Confirmed bugs in VSCALE. Runtimes are for Symbolic QED with concrete, power-on reset starting state.

Bug Activation	Bug Effect	BMC Runtime
The value '1' is written to specific bit positions in the machine-interrupt CSR MIP.	MTIMECMP register corrupted; Causes repeated interrupts.	2 seconds
Any value with lower two bits '01' or '10' written to the machine-level CSR MSTATUS.	Design enters unspecified privilege level; MEPC register corrupted;	33 seconds

3.2. "Extremal" Bugs

To demonstrate the robustness of our presented technique, we inject "extremal" bugs (only triggered when the design reaches a specific set of states) into RIDECORE (since it is OoO, superscalar, and more complex than Vscale). Our extremal bug injection methodology is as follows: i) Run Matrix Multiply (1M cycles) on the design in simulation[6] and stops the simulation at a random point in time; ii) Run a uniform random sequence of 100 ALU or Load/Store instructions; iii) Select a uniformly random subset of flip-flops from the set of all flip-flops in the design and record their logic values; and iv) Generate a bug (effect A.1.b.3 of

[1]The formal tool is free to choose any values for symbols (state bits) associated with in-flight instructions, including those that are not consistent with the logic driving those symbols. Thus, values chosen for symbols in a SIF instruction need not constitute a valid instruction.
[2]Operands may come from either registers or memory locations. For register (memory) operands, the dispatch stage is the register read (memory read) stage.

[3]This condition is required for OoO cores, where there is a possibility that the Symbolic QED operand may wait on a SIF instruction instead of a Symbolic QED instruction.
[4]MULH is a signed multiply instruction selecting the upper half of the multiplier result.
[5]MULHU is an unsigned multiply instruction selecting the upper half of the multiplier result.
[6] This simulation is only for "extremal" bug creation – not for verification or bug detection.

Appendix A), injected into the design. This bug is only activated when the design reaches a state where all the selected flip flops (step iii) have a specific set of values recorded (step iii).

We present our results in Table 4. For generating such extremal bugs, we randomly chose 180 time points (step i), ranging from 26,026 to 988,159 clock cycles elapsed from program start. For each time point, we ran a random 100-instruction sequence (step ii), and then randomly selected 10 different subsets of 128 flip-flops (step iii), resulting in 1,800 total extremal bug count.

Whereas Symbolic QED with concrete starting state detected 0% of these 1,800 "extremal" bugs, Symbolic QED with symbolic starting state was able to detect 1,763 of the 1,800 bugs. For the remaining cases, the BMC tool timed out after 24 hours. A closer inspection reveals that the BMC tool was not able to unroll the design beyond 7 clock cycles (8 cycles are needed to observe these bugs). In future work, we plan to investigate ways to improve BMC tools to address such issues (following approaches such as [Ganai 04, 06]).

Table 3. "Long" logic bugs and HTs. We report [min, average, max].

		"Long" Bugs	HTs
Vscale	Total count injected	117	156
	Symbolic QED with symbolic starting state		
	Coverage	100%	100%
	Bug trace length (instructions)	[2, 2, 3]	[2, 2, 3]
	Bug trace length (clock cycles)	[5, 5, 6]	[5, 5, 6]
	BMC runtime (seconds)	[2, 4, 25]	[2, 11, 313]
	Symbolic QED with concrete starting state		
	Coverage	33%	15.3%
RIDECORE	Total count injected	120	195
	Symbolic QED with symbolic starting state		
	Coverage	100%	100%
	Bug trace length (instructions)	[4, 4, 4]	[4, 4, 4]
	Bug trace length (clock cycles)	[8, 8, 8]	[8, 8, 8]
	BMC runtime (minutes)	[7, 13, 18]	[7, 20, 121]
	Symbolic QED with concrete starting state		
	Coverage	5%	8.7%

Table 4. "Extremal" logic bugs for RIDECORE. We report [minimum, average, maximum].

Total count injected	1,800
Symbolic QED with symbolic starting state	
Coverage	97.9%
Bug trace length (instructions)	[4, 4, 4]
Bug trace length (clock cycles)	[8, 8, 8]
BMC runtime (minutes)	[8, 33, 149]
Symbolic QED with concrete starting state	
Coverage	0%

Observation 4: Our new Symbolic QED with symbolic starting states correctly and automatically found 97.9% of the "extremal" logic bugs and generated a bug trace in less than 3 hours. In contrast, Symbolic QED with concrete starting state detected 0% of the extremal bugs.

4. Related Work

Existing formal verification techniques employing BMC [Reid 16, Singh 18] have issues in detecting bugs that require a long activation sequence. Other works for processor cores use theorem proving [Bhadra 07], or try to learn invariants on the design [Thalmaier 10] to be used as constraints, but these techniques tend to be ad-hoc and require a high level of manual effort. Seminal work [Burch 94] verified abstracted models of processors by using a set of specifications. But, choosing proper specifications to avoid false positives remains a major problem.

While false positives are recognized as a major challenge for traditional BMC, Symbolic QED uniquely enables the approach presented in this paper. The same QED constraints may not prevent false positives for general property checking using BMC. The following example illustrates this point. Let a processor core start at a state where the Exception Program Counter (*EPC*) (register storing the return address for an exception) is misaligned (i.e., not aligned any word in the instruction cache), the current PC is within an exception handling routine, and there are only NOP instructions in the pipeline. This is an unreachable state for processors with strict alignment rules (e.g., MIPS). It is reasonable to check the property that the EPC is aligned, since returning to a misaligned address can cause programs to crash. Even at Tc, when the NOP sequence is finished, this EPC will still be misaligned, causing a false positive. When using QED constraints with Symbolic QED though, we do not get such a false positive. This is because the exception handling routine will be filled by valid QED tests, and any time we assert a QED check, it won't fail unless there is a bug in the design (Symbolic QED can still detect legitimate misalignment bugs by checking CSRs).

With similar objectives to [Fadiheh 18], our new approach is especially suited for a broader class of processor designs (including OoO superscalar processors). The QED constraints (Sec. 2.1), together with QED recorders (Sec. 2.2) and the new QED module (Appendix B.3), enable this capability. Our approach in this paper can detect bugs that would be missed by [Fadiheh 18]. An example is: *When all registers in the register file contain the value -1, the next register write is corrupted.* This bug escapes the [Fadiheh 18] technique, because it affects the original and duplicate instructions equivalently. In contrast, our technique will create scenarios where a mismatch between an original and duplicate instruction occurs. Our technique is also applied to HT detection, while that was not an objective of [Fadiheh 18]. Our technique assumes a ROB for OoO processors. For processors with static pipelines and OoO write-back (e.g., [Aquarius]), an additional constraint [Fadiheh 18; Eq. (3a-b)] is required.

Existing HT detection techniques that can be applied in pre-silicon verification broadly belong to two categories: i) design analysis methods; and ii) Formal methods [Xiao 16]. One class of design analysis techniques use the observation that signals associated with HTs may be mostly "unused" or "rare". [Cakır 15, Hicks 10, Zhang 11, Zhang 13b] use simulation data along with "rareness" metrics (e.g., code coverage, signal correlation). [Waksman 13, Yao 15] do not need simulation data, but still trade off false-positives (i.e., spurious detection of HTs) for false-negatives (i.e., failure to detect HTs) and vice-versa, depending on the thresholds set for their "rareness" metrics. Additionally, stealthy HTs have been designed [Zhang 14] to bypass such analyses. In contrast, our technique does not require simulation data, detects "stealthy" HTs given in [Salmani 13, Zhang 13a, Zhang 14], and does not produce any false positives. However, our technique is for processor cores, while the aforementioned analysis techniques are applicable for general designs.

Formal methods for finding HTs generally either use BMC [Rajendran 15, 16], SAT-based equivalence checking [Banga 10, Reece 16, Shrestha 12] or theorem proving [Guo 17, Jin 13, Love 12]. These techniques face similar challenges as BMC-based techniques for bug detection, especially those (bugs and HTs) that require long activation sequences (in addition to manual creation of properties, associated with BMC for HT detection).

Complementary approaches to ours include methods focusing on detection of HTs that leak sensitive data, but do not produce incorrect logic values [Fern 17, Hu 16, Jin 12, Rajendran 16], and HT prevention techniques [Chakraborty 09, Dupuis 14, Samimi 16].

5. Conclusion

In this paper, we extended Symbolic QED to include symbolic starting states. As a result, we overcome limitations of existing pre-silicon verification techniques for detecting logic bugs and HTs that require long activation sequences. The unique combination of Symbolic QED and QED constraints enable us to achieve this objective. Our results on multiple open-source RISC-V processor cores demonstrate the effectiveness and practicality of our approach: (i) detection of previously unknown logic bugs within minutes; (ii) detection of 100% of hundreds of long logic bugs and HTs (Symbolic QED with concrete starting state detects, at best, 33%); (iii) detection of 97.9% of "extremal" logic bugs (Symbolic QED with concrete starting state detects 0%). Future research directions include: i) extending our approach to detect bugs and HTs in other SoC components (beyond processor cores), e.g., uncore components and accelerators; ii) handling other QED transformations (beyond EDDI-V), e.g., CFTSS-V and CFCSS-V [Singh 18]; iii) automated methods for inserting the QED recorders and generating QED constraints on the symbolic starting state.

References

[Aquarius] "Aquarius," opencores.org/projects/aquarius.
[ARM] "Cortex-A15," https://tinyurl.com/y75sejwz.
[Banga 10] Banga, M., and M.S. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," Proc. *HOST*, 2010.
[Bhadra 07] Bhadra, et al., "A survey of hybrid techniques for functional verification," *IEEE Design & Test of Computers*, 24(2):112-122, 2007.
[Burch 94] Burch, J.R., and D. Dill, "Automatic verification of pipelined microprocessor control," Proc. *ICCAD*, 1994.
[Cakır 15] Cakır, B., and S. Malik, "Hardware Trojan detection for gate-level ICs using signal correlation based clustering," Proc. *DATE*, 2015.
[Chakraborty 09] Chakraborty, R.S., and S. Bhunia, "HARPOON: an obfuscation-based SoC design methodology for hardware protection," *IEEE Trans. CAD*, vol. 28: pp. 1493-1502, 2009.
[Clarke 01] Clarke, E., et al., "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in Sys. Design*,19(1):7-34, 2001.
[Dupuis 14] Dupuis, S., et al., "A novel hardware logic encryption technique for thwarting illegal overproduction and Hardware Trojans," Proc. *IOLTS.*, 2014.
[Extended 18] **Extended version of paper. Removed for anonymity – we will be glad to provide this if the program committee requests.**
[Fadiheh 18] Fadiheh, M. R., et al., "Symbolic quick error detection using symbolic initial state for pre-silicon verification," Proc. *DATE*, 2018.
[Fern 17] Fern, N., I. San, and K.T.T. Cheng, "Detecting hardware Trojans in unspecified functionality through solving satisfiability problems," Proc. *ASP-DAC*, 2017.
[Foster 15] Foster, H.D., "Trends in functional verification: A 2014 industry study," Proc. *DAC*, 2015.
[Ganai 04] Ganai, M.K., A. Gupta, and P. Ashar, "Efficient modeling of embedded memories in bounded model checking," Proc. *CAV*, 2004.
[Ganai 06] Ganai, M.K., and A. Gupta, "Accelerating high-level bounded model checking," Proc. *ICCAD*, 2006.
[Guo 17] Guo, X., et al., "Eliminating the Hardware-Software Boundary: A Proof-Carrying Approach for Trust Evaluation on Computer Systems," *IEEE Trans. Inf. Forensics Security*, 12(2):405-417, 2017.
[Hicks 10] Hicks, M., et al., "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," *SSP*, 2010.
[Hu 16] Hu, W., et al., "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking," *Computer*, 49(8):44-52, 2016.
[Jin 12] Jin, Y., and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," Proc. *VTS*, 2012.
[Jin 13] Jin, Y., and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," Proc. *ICCAD*, 2013.
[Karri 10] Karri, R., et al., "Trustworthy hardware: Identifying and classifying hardware Trojans," *Computer*, 43(10):39–46, 2010.
[King 08] King, S.T., et al., "Designing and implementing malicious hardware," Proc. *LEET*, 2008.
[Lin 14] Lin, D., et al., "Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection," *IEEE Trans. CAD*, 33(10):1573-1590, 2014.
[Love 12] Love, E., Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Inf. Forensics Security*, 7(1):25-40, 2012.
[MIPS 96] Yeager, K, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, 16(2):28–41, 1996.
[Rajendran 15] Rajendran, J., V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," Proc. *DAC*, 2015.
[Rajendran 16] Rajendran, J., et al., "Formal Security Verification of Third-Party Intellectual Property Cores for Information Leakage," Proc. *Intl. Conf. on VLSI Design*, 2016.
[Reece 16] Reece, T., and W.H. Robinson, "Detection of Hardware Trojans in Third-Party Intellectual Property Using Untrusted Modules," *IEEE Trans. CAD*, 35(3):357-366, 2016.
[Reid 16] Reid A., et al., "End-to-end verification of processors with ISA-Formal" CAV, 2016.
[Ridecore1] "RIDECORE," https://github.com/ridecore/ridecore.
[Ridecore2] "Issue: bugs in rs_mul," https://tinyurl.com/y8otzyxb.
[RISCVP] "Privileged Architecture," https://tinyurl.com/y8jgqqza.
[Salmani 13] Salmani, H., M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," *ICCAD*, 2013.
[Samimi 16] Samimi, M.S., et al., "Hardware enlightening: No where to hide your Hardware Trojans!" Proc. *IOLTS*, 2016.
[Shrestha 12] Shrestha G., and M.S. Hsiao, "Ensuring trust of third-party hardware design with constrained sequential equivalence checking," Proc. *IEEE Conf. on Tech. for Homeland Security*, 2012.
[Singh 18] Singh, E., et al., "Logic Bug Detection and Localization Using Symbolic Quick Error Detection," *IEEE Trans. CAD*, 2018.
[Thalmaier 10] Thalmaier, M. et al., "Analyzing k-step induction to compute invariants for SAT-based property checking," Proc. *DAC*, 2010.
[Vscale] "V-scale," https://github.com/ucb-bar/vscale.
[Waksman 13] Waksman, A. et al., "FANCI: Identification of stealthy malicious logic using boolean functional analysis," *CCCS*, 2013.
[Xiao 16] Xiao, K., *et al.,* "Hardware Trojans: Lessons Learned after One Decade of Research." *TODAES.* 22(1), May 2016.
[Yao 15] Yao, S. et al., "FASTrust," *ITC*, 2015.
[Zhang 11] Zhang, X. et al., "Case study: Detecting hardware Trojans in third-party digital IP cores," *HOST*, 2011.
[Zhang 13a] Zhang J. et al., "On hardware trojan design and implementation at register-transfer level," *HOST*, 2013.
[Zhang 13b] Zhang, J. et al., "VeriTrust," *DAC*, 2013.
[Zhang 14] Zhang, J. et al., "Detrust," *CCCS*, 2014.

Appendix A. Bug and HT scenarios

In the following tables, we give the different logic bug (harder versions of "difficult" bugs that occurred in various commercial designs [Singh 18]) and HT scenarios (from research literature) used in Table 3 of Sec. 3. Each "long" logic bug is modeled with two parts: i) activation criteria of the bug (Table A.1.a), i.e., the conditions which need to be satisfied for the bug to activate; and ii) effect of the bug once it is activated (Table A.1.b). For our experiments, we considered a whole range of values for the parameters in Table A.1, as follows, N=Y={2,4,8,16,32,64,128,256}, R=X={2,4,6,…,30}. This results in a total of 117 logic bugs in Vscale (Activation A.1.a.5 is not possible), and 120 logic bugs in RIDECORE.

Table A.1.a. Activation criteria for "long" logic bugs.

Processor Core	1. Data forwarding between pipeline stages.
	2. Two specific instructions within X cycles.
	3. R registers must each contain a specific value V.
	4. A specific sequence of N instructions must execute within Y cycles.
	5. A specific cache state.

Table A.1.b. Bug effect from [Singh 18].

Processor Core	1. Next instruction corrupted to NOP.
	2. Next instruction opcode incorrectly decoded.
	3. Next instruction register read corrupted.

Table A.2.a. Activation criteria for HTs from [Salmani 13].

Processor Core	1. Specific length N sequence on M_1 internal wires.
	2. X_1 bit counter reaching final value.
	3. Comparator on M_2 internal wires becomes true.
	4. X_2 bit rare event counter reaches a specific value.

Table A.2.b HT effects

Processor Core	1. An in-flight instruction changed to NOP.
	2. Opcode of an in-flight instruction changed.
	3. Next register read corrupted.
	4. Next result of an execution unit changed.
	5. Corrupts ROB. Prematurely commits next inst.

Table A.2.c. HT Design Techniques

Methodology	HT stealthy against following techniques
[Salmani 13]	Traditional pre-silicon verification techniques.
[Zhang 13a]	UCI [Hicks 10]; coverage metrics [Hicks 10, Zhang 11].
[Zhang 14]	[Hicks 10, Zhang 11, Waksman 13, Zhang 13b].

Table A.2.a gives HT activation scenarios that capture HT triggering mechanisms in the Trust-Hub benchmarks [Salmani 13]. Table A.2.b gives various effects an HT can have on the executing instructions [King 08]. Table A.2.c presents three HT implementation techniques used to inject HTs in designs [Rajendran 15]. We create stealthy HTs that are known to evade common detection techniques (e.g., HT designs from [Zhang 13a] evade detection techniques based on UCI [Hicks 10] and coverage metrics [Hicks 10, Zhang 11]). A HT scenario is formed by using one activation criteria (Table A.2.a) with one bug effect (Table A.2.b), along with an appropriate design strategy (Table A.2.c). We used a wide range of HT scenario parameters, given in Table A.2: N={2,4,8,…,256}, M_1=32, X_1=X_2={128,256}, M_2=64, resulting in 156 HT scenarios in Vscale (effect A.2.b.5 is not possible) and 195 HT scenarios in RIDECORE. These values make HTs harder to activate than benchmark HTs in [Salmani 13].

Appendix B. Background

B.1. QED and the EDDI-V Transformation

QED is a testing technique that takes existing system validation tests and automatically transforms them into a set of new tests using various QED transformations [Lin 14]. The EDDI-V transform of QED [Lin 14], which is the focus of this work, targets bugs inside processor cores by checking the results of *original* instructions against the results of *duplicate* instructions. First, the registers and memory space are divided into two halves, one for the original instructions and one for the duplicated instructions. Next, corresponding registers and memory locations for the original and the duplicated instructions are initialized to the same values. Then, for every load, store, arithmetic, logical, shift, or move instruction in the original test, EDDI-V creates a corresponding duplicate instruction that performs the same operation, but uses the registers and memory reserved for duplicate instructions. The duplicated instructions execute in the same order as the original instructions. The EDDI-V transformation also inserts periodic check instructions that compare the results of the original instructions against those of the duplicated instructions. A mismatch in any check indicates an error.

B.2. Symbolic QED

Symbolic QED [Singh 18] combines QED transformations with *Bounded Model Checking* (BMC) [Clarke 01]. Symbolic QED creates a BMC problem that searches through *all possible* EDDI-V tests within a bounded number of cycles. It searches for counterexamples to properties of form: Ra==Ra'. Here Ra is an original register, Ra' is corresponding duplicate register in an EDDI-V test. To ensure that counterexamples are *QED-compatible* (EDDI-V only for this paper): 1. Inputs must be valid instructions; 2. Instruction sequence is an EDDI-V test;

The QED module automatically transforms a sequence of original instructions into a QED-compatible sequence (e.g., as in Fig. 5). The QED module only requires that the input sequence is made up of valid instructions that read from or write to only the original registers and memory (conditions that can be specified directly to the BMC tool). After execution, a signal is asserted (*qed_ready*) when all original and corresponding duplicate registers should contain the same values under bug-free situations, i.e., the BMC tool should check the property:

$$qed\_ready \rightarrow \bigwedge_{a\in\{0..\frac{N}{2}-1\}} \mathrm{Ra} == \mathrm{Ra}',$$

where N is the number of registers defined by the ISA. Here (for a $\in \{0..N/2-1\}$), Ra and Ra$'$ correspond to original and duplicate registers.

The starting state for the BMC run must also be a *QED-consistent* state, in which the value of each original register or memory location matches the corresponding duplicate register or memory location. This is to prevent false counter-examples from being generated. One way to obtain such a state is to run an EDDI-V test in simulation and stop immediately after QED checks have compared all register and memory values.

```
                    R1 = R1 + 5
                    R3 = R1 * R2
R1 = R1 + 5         R17 = R17 + 5
R3 = R1 * R2        R19 = R17 * R18
    (a)                 (b)
```

Fig. 5: Example of QED transformation by the QED module. (a) A sequence of original instructions, and (b) transformed instructions executed by the processor.

Symbolic QED detects HTs by finding an EDDI-V test for which the HT affects original registers and duplicate registers differently. Example: assume a HT is inserted (completely unknown to designer) with activation criteria A.2.a.2 (X_1=128) and effect A.2.b.1. If the counter is initialized to 2^{128}-1, Symbolic QED can detect the HT using the EDDI-V test {ADDI R1,2; ADDI R17,2; CHECK R1==R17}. However, the existence of the counter is unknown, so it is impossible to pick the concrete starting state *a priori*. Symbolic QED with symbolic starting states will automatically detect this HT by starting the design at state {R1=0, R17=0, HT counter initialized to 1 cycle before activation}, and running the EDDI-V test.

B.3. New QED module for Single Processor Cores

Pseudocode for the new QED module is given in Fig. 6(a). Inputs are: 1) *enable*: disables the QED module if 0; 2) *next_instruction*: next instruction to be executed; 3) *fetch_next*: high when the core is ready to receive an instruction (i.e., fetch stage is not stalled); 4) *original*: tells the core to execute an original (if high) or duplicate (if low) instruction. Outputs are: 1) *instruction_valid*: indicates whether the output instruction is valid; and 2) *instruction_out*: instruction to be executed. The QED module has internal variables: 1) *queue*: a queue data structure that stores previous original instructions that have not yet been executed in the duplicate subsequence; 2) *head_instruction*: the previous head of the queue; 3) *insert_valid*: true when the QED module can execute an original instruction; 4) *delete_valid*: true when the QED module can execute a duplicate instruction; 5) *duplicate_instruction* next instruction in the duplicate subsequence to be executed (when *original* is 0).

To run Symbolic QED, we also need to determine when it is safe to assert QED checks. Pseudocode for determining the *qed_ready* signal is given in Fig. 6(b). To avoid false fails, QED checks occur when an equal number of commits (writes) have been made to original registers and duplicate registers. This is accomplished by keeping track of the number of original and duplicate commits to the register set, as shown in Fig. 6(b). For simplicity, in Fig. 6(b), we assume that at most one instruction commits per cycle. For superscalar processors that can commit multiple instructions in the same cycle, we track all corresponding pairs of *write_valid* (tells whether the input data is valid) and *write_address* (the address for the data to be written) signals, keep a separate *is_original* signal (identifies if an address corresponds to an original or duplicate location) for each instruction, and allow the original and duplicate counters to be incremented multiple times if needed.

The QED module of [Singh 18] requires that all original instructions complete, a waiting period occurs for the pipeline to be flushed, and duplicate instructions execute, before the *qed_ready* signal is asserted. This new QED module instead allows arbitrary interleaving (by giving control of the *original* input of Fig. 6(a) to the BMC tool) of the original and duplicate instruction subsequences without requiring the waiting period. QED ready enable logic (Fig. 6(b)) can be further enhanced:

1. The current QED ready enable logic is only applicable to single processor cores, since a multi-core system would require modification of the *qed_ready* logic to consider the original and duplicate commits across all cores. This can be challenging in situations where multiple cores operate with a shared address space. For simplicity, we do not consider this situation in this paper.
2. For some processors, e.g., superscalar processors with explicit register renaming (MIPS 10000 [MIPS 96] and ARM's Cortex-A15 [ARM]) the designation of original or duplicate instruction cannot be made solely on where they write (unlike in Fig. 6(b)). This issue can be corrected by including the current state of the register mapping table as an input to the function is_write_to_original_space. Each time a QED check happens, the same mapping table must be used to map logical to physical addresses before comparing "original" and "duplicate" values. The RISC-V cores used in this paper, however, do not have this issue.

```
INPUT: enable, next_instruction, fetch_next, original
OUTPUT: instruction_out, instruction_valid
// initialization
queue ← 0;  head_instruction ← 0;
// end initialization
insert_valid ← fetch_next & original & ~queue.is_full();
delete_valid ← fetch_next & ~original & ~queue.is_empty();
instruction_valid ← insert_valid | delete_valid;
if insert_valid then
  queue.push(next_instruction);  // store this instruction in queue
else if delete_valid then
  head_instruction ← queue.pop();  // remove instruction at the head from queue
end if
duplicate_instruction ← create_duplicated_version(head_instruction);
instruction_out ← (enable & ~original) ? duplicate_instruction : next_instruction;
```

(a)

```
INPUT: write_valid, write_address
OUTPUT: qed_ready
// initialization
qed_ready ← false;  count_original ← 0;  count_duplicate ← 0;
// end initialization
is_original ← is_write_to_original_space(write_address);
if write_valid then
  if is_original then
    count_original++;       // increment number of original instructions committed
  else
    count_duplicate++;      // increment number of duplicate instructions committed
  end if
end if
qed_ready ← (count_original == count_duplicate) ? true : false;
```

(b)

Fig. 6. Pseudocode for (a) QED module, and (b) QED ready enable logic