SEPE-SQED: Symbolic Quick Error Detection by Semantically Equivalent Program Execution

[1,2]Yufeng Li, [2]Qiusong Yang*, [2]Yiwei Ci, [1,2]Enyuan Tian
Institute of Software, Chinese Academy of Sciences, Beijing, China[1]
University of Chinese Academy of Sciences, Beijing, China[2]
crazybinary494@gmail.com,{qiusong,yiwei}@iscas.ac.cn,tianenyuan@nfs.iscas.ac.cn

arXiv:2404.03172v2 [cs.SE] 6 Apr 2024

ABSTRACT

Symbolic quick error detection (*SQED*) has greatly improved efficiency in formal chip verification. However, it has a limitation in detecting *single-instruction bugs* due to its reliance on the *self-consistency* property. To address this, we propose a new variant called *symbolic quick error detection by semantically equivalent program execution* (*SEPE-SQED*), which utilizes program synthesis techniques to find sequences with equivalent meanings to original instructions. *SEPE-SQED* effectively detects *single-instruction bugs* by differentiating their impact on the original instruction and its semantically equivalent program (instruction sequence). To manage the search space associated with program synthesis, we introduce the *CEGIS based on the highest priority first* algorithm. The experimental results show that our proposed CEGIS approach improves the speed of generating the desired set of equivalent programs by 50% in time compared to previous methods. Compared to *SQED*, *SEPE-SQED* offers a wider variety of instruction combinations and can provide a shorter trace for triggering bugs in certain scenarios.

CCS CONCEPTS

• **Hardware → Model checking**.

KEYWORDS

Formal Verification, *SQED*, Program Synthesis, CEGIS, Semantically Equivalent Program Execution, *SEPE-SQED*

ACM Reference Format:
[1,2]Yufeng Li, [2]Qiusong Yang, [2]Yiwei Ci, [1,2]Enyuan Tian. 2024. SEPE-SQED: Symbolic Quick Error Detection by Semantically Equivalent Program Execution . In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

As technology advances and the chip market expands, there is a growing need for cost-effective and efficient chip verification methods. However, ensuring the accuracy of a processor's behavior becomes more challenging due to aggressive microarchitectural optimizations and the daunting task of considering all possible instruction interleavings.

Formal verification (FV), such as model checking [1], excels at detecting corner cases through exhaustive design analysis. It constructs a mathematical representation of the system and formally proves desired properties. Pioneering work includes symbolic model checkers based on abstract models of microarchitecture design [2, 3]. However, abstract models often overlook elusive bugs that can arise in RTL description. ISA-Formal [4] for ARM processors and RISCV-Formal [5] for RISC-V processors verify designs directly from RTL descriptions, but the formulation of formal properties requires significant manual effort and substantial expertise.

To address these issues, a groundbreaking formal verification approach called symbolic quick error detection (*SQED*) [6, 7, 8] has been proposed. *SQED* utilizes model checking to prove that any instruction sequence up to a certain bound produces a correct result. It leverages the concept of design *self-consistency* to establish a single universal property, which declares that the outcomes produced by both original instructions and their duplicates are identical, regardless of the specific microarchitectural design details. Therefore, *SQED* does not require any manual property formulation. The logic bugs that can induce changes in processor architectural states can be categorized as either *single-instruction* or *multiple-instruction bugs*. *Single-instruction bugs* refer to the erroneous behavior of a processor when executing a specific individual instruction. These bugs are independent of all previously executed instructions. *Multiple-instruction bugs* refer to the erroneous behavior of a processor when executing a sequence of multiple instructions consecutively. Practical examples have demonstrated that *SQED* is capable of efficiently detecting *multiple-instruction bugs* that are otherwise difficult to detect [6, 7]. However, it cannot detect *single-instruction bugs* [8] that affect original and duplicate instructions uniformly. Therefore, the verification process of the *self-consistency* property may yield false positive results when dealing with bugs that occur within a single instruction. This motivated the work on *C-S*^{2}*QED* [9] which formulates *single-instruction* semantics by in-house metamodeling techniques [10], but requires providing timing information about instruction in the properties, making it not microarchitecture-independent.

In this paper, we present a novel variant for *SQED*, named *symbolic quick error detection by semantically equivalent program execution (SEPE-SQED)*. *SEPE-SQED* determines the correctness of an implementation by verifying whether the execution of the original instruction produces consistent architectural states with the execution of its semantically equivalent program. *SEPE-SQED* is capable of addressing both *single-instruction* and *multiple-instruction bugs* of the processor design. In the case of *single-instruction bugs*, their effect on the original instruction and its semantically equivalent program can vary, leading to a violation of consistency. On the other hand, *SEPE-SQED* offers a richer variety of instruction combinations compared to the singular pattern of combining original and duplicate instructions in *SQED*. As a result, *SEPE-SQED* provides greater flexibility in triggering bugs, and in some scenarios, it can lead to shorter bug traces. The *component-based counterexample-guided inductive synthesis (CEGIS)* [11, 12] is employed to search for programs that are semantically equivalent to the original instructions. To address the vast search space associated with existing program synthesis methods, we propose the *CEGIS based on the highest priority first (HPF-CEGIS)* algorithm. The experimental results demonstrate the effectiveness of our approach.

The contributions of this paper are as follows:

*Corresponding author
This work was supported by Basic Research Projects from the Institute of Software, Chinese Academy of Sciences (Grant No. ISCAS-JCZD-202307) and the National Natural Science Foundation of China (Grant No. 62372438).

- We improve *SQED* by incorporating program synthesis techniques, and propose *SEPE-SQED*, which can detect all types of logic bugs that can induce changes in processor architectural states.
- We introduce the *HPF-CEGIS* algorithm, which, compared to the previous CEGIS algorithm, synthesizes the desired program with an average reduction of 50% in time overhead.
- We verified the capability of *SEPE-SQED* to detect two types of logic bugs through mutation testing on a real open-source high-performance processor, and the experiments demonstrated that *SEPE-SQED* can generate bug traces shorter than *SQED* for certain *multiple-instruction bugs*.

2 BACKGROUND AND RELATED WORK

In this section, we provide an introduction to the background knowledge of *QED* and *SQED* used for formal verification and program synthesis techniques. Along the way, we present the related work.

2.1 QED and SQED

Quick Error Detection (*QED*) [13] is a testing technique that automatically transforms an existing test, which consists of a sequence of instructions, into a new test using various transformations (*QED* transformation). These transformations, such as *Error Detection using Duplicated Instructions for Validation (EDDI-V)* and *Proactive Load and Check (PLC)*, enhance coverage and reduce error detection latency. The EDDI-V transformation is particularly relevant to this paper as it involves the duplication of instructions in an existing instruction sequence using shadow registers and memory. The *design under verification (DUV)* has its registers and memory space divided into two halves, each mapped to the other through a bijective mapping. The original and duplicate instructions exclusively refer to their respective parts. In the EDDI-V transformation, every original instruction is replicated as a duplicate instruction, with the register and memory locations mapped to their corresponding values. During a *QED* test, both the original and duplicate instruction sequences are executed from a *QED-consistent* state, where values in corresponding registers and memory locations are identical. Duplicated instructions execute in the same relative order as the originals but may be interleaved [8]. Mismatched values between original and duplicate registers or memory locations indicate the presence of a bug trace.

Symbolic quick error detection (*SQED*) [6, 8] utilizes *QED* principles and *bounded model checking (BMC)* [14] to detect and localize logic bugs in RTL. *SQED* systematically explores all possible instruction sequences of increasing length in a symbolic manner. *QED* transformations are then applied to these enumerated instruction sequences. It automates the process of property formulation, a known challenging task, by checking a universal property (i.e., a property that is design-independent) based on *QED* testing. For the EDDI-V transformation, this property is referred to as self-consistency, and it can be expressed as follows:

$$QED\text{-ready} \Rightarrow QED\text{-consistent} \tag{1}$$

The *QED*-ready flag serves as an indicator of the successful commitment of both the original and duplicated instructions. In the context of a processor core equipped with 32 general-purpose registers, a state is deemed to be *QED*-consistent when the equality $\bigwedge_{i=0}^{15} regs[i] == regs[i+16]$ (where *regs* represents the register file) holds. It is important to note that registers 0 to 15 correspond to the original registers, while registers 16 to 31 are their respective duplicates, following a mapping scheme where register $regs[i]$ is associated with register $regs[i+16]$.

SQED with EDDI-V transformation is unable to detect *single-instruction bugs* that affect both the original and duplicate instructions in a uniform manner during a *QED* test. Consequently, the original and duplicate registers always hold the same value, leading to *QED*-consistent states.

2.2 Program Synthesis

The program synthesis aims to find a program that satisfies a given specification represented as a bit-vector formula in satisfiability modulo theories (SMT) [15]. The problem is an *exists-forall* problem expressed as follows:

$$\exists P : \forall \vec{I}, O : \Big(P(\vec{I}) == O\Big) \Rightarrow \phi_{spec}(\vec{I}, O) \tag{2}$$

(2) indicates that if the program is executed with inputs $\vec{I}$ and produces output O, then the specification ϕ_{spec} is satisfied. If a satisfiable result of (2) is obtained through the SMT solver query, then a program P is synthesized.

An effective approach for solving satisfiability problems in second-order logic (∃∀) is to employ CEGIS [16] to eliminate the universal quantifier. CEGIS involves two SMT solver calls: one to construct a candidate program and another to verify its validity for all possible inputs. Gulwani et al. [11] have utilized this technique for *component-based loop-free program synthesis*. They introduced first-order *location variables* L to establish component connections. Location variables determine the parameters of components based on their linear order. Hence, the synthesis problem is equivalent to solving the following constraint:

$$\exists L : \big(\psi_{wfp}(L) \land \forall \vec{I}, O, Q, R : \phi_{lib}(Q, R) \land \psi_{conn}(\vec{I}, O, Q, R, L) \Rightarrow \phi_{spec}(\vec{I}, O)\big)$$

$$where \quad Q ::= \bigcup_{j=1}^{N} \vec{I^j} \quad R ::= \bigcup_{j=1}^{N} O^j \quad L ::= \{l_x | x \in Q \bigcup R \bigcup \vec{I} \bigcup O\}$$

Here Q and R collectively denote the formal inputs and outputs of N components. ϕ_{lib} encapsulates the formula of the components, $\phi_{lib} ::= \bigwedge_j^N \phi_j(\vec{I^j}, O^j)$. l_x denotes the location of each variable x. The *well-formed program constraint* $\psi_{wfp}(L)$ mandates that the inputs of each component must be either the program inputs ($\vec{I}$) or the outputs of its preceding component, while also ensuring that the outputs of each component are distinct. Furthermore, the constraint ψ_{conn} guarantees that variables sharing the same location possess identical assignments.

However, the classical CEGIS necessitates multiple instances of each component in the library, resulting in a substantial performance overhead due to the excessive number of components. To address this, Buchwald et al. [12] proposed an iterative CEGIS algorithm, which involves using the *combinations with replacement* algorithm to select subsets of components from the library and form increasingly longer multisets. Each multiset allows for the repetition of components, and each iteration involves synthesizing the specification using the small-sized multiset. Since the goal is not to find all programs that satisfy the specification, this approach can yield shorter programs that satisfy the specification while significantly reducing the synthesis time overhead. However, the excessive number of components can lead to a large number of multisets ($\left(\!\binom{N}{n}\!\right) = \binom{N+n-1}{n}$). For instance, if there are $N = 29$ components and $n = 6$ are chosen each time to form a multiset, it would result in *1344904* multisets. Moreover, many of these multisets actually cannot synthesize the specification,

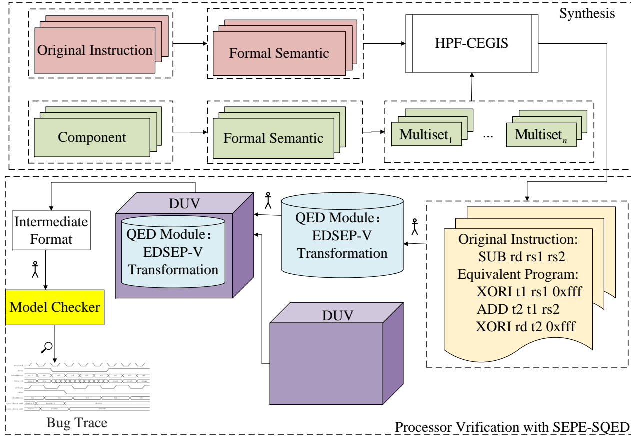

Figure 1: Workflow

```
#Original instruction      #Semantically equivalent program
SUB rd rs1 rs2             XORI t1 rs1 0xfff
                           ADD  t2 t1 rs2
                           XORI rd t2 0xfff
```

Listing 1: *SUB* instruction and its semantically equivalent program

leading to numerous invalid solver calls. Therefore, in this paper, we propose the *HPF-CEGIS* algorithm (see Section 4.2).

3 OVERVIEW

This section provides an overview of our approach, which involves two primary processes: synthesizing programs that are semantically equivalent to the original instructions and verifying the processor using *SEPE-SQED*. The workflow is illustrated in Figure 1.

The upper half of Figure 1 illustrates the process of program synthesis, where the semantic models of original instructions and components defined by us (see Section 4.1), are input into the program synthesizer. Subsequently, the program synthesizer invokes *HPF-CEGIS* algorithm (described in Section 4.2) to generate programs from the components that are semantically equivalent to the given original instructions. Once the instruction sequences that are semantically equivalent to the original instruction are identified, the second process involves utilizing this equivalent relation (as shown in Listing 1) to construct transformations.

To implement *SEPE-SQED*, a special *QED* module [13] is integrated with the DUV. The module is only used for pre-silicon verification and is not added to the manufactured integrated circuit. The module takes an original instruction as input and outputs an instruction sequence that is semantically equivalent to it. Both the original and the semantically equivalent instruction sequences are fed into the DUV. Once the *QED*-ready signal is activated, indicating the successful submission of both the original and its semantically equivalent instruction sequence, the model checker checks whether the universal property (formula (1)) holds. In this paper, we refer to this transformation as *Error Detection using Semantically Equivalent Program for Validation, EDSEP-V* (described in Section 5).

4 SYNTHESIS

We first present the formal semantic model of instructions (Section 4.1), followed by an introduction to our *HPF-CEGIS* algorithm (Section 4.2).

4.1 Formal Semantic Model

The synthesizer takes the semantic models of original instructions and components as inputs and utilizes these components to synthesize the original instructions that serve as the specifications (refer to formula (2)) [11, 12]. The semantic models are represented as bit-vector formulas that precisely describe the input-output behavior of the instructions. In this paper, we use a portion of the *RV32IM* [17] instruction set as the illustrative example.

The input and output parameters of an instruction semantic model represent register or immediate operands. They are all of the bit-vector type, but the inputs may have different bit widths. Some instructions in the library components have *internal attributes* whose values are determined during synthesis. For instance, the *ADDI* instruction has two forms as components. The first form takes both register and immediate operands as input parameters, while the second form takes only the register operand as input parameter, with the immediate operand being treated as an *internal attribute*. When this component is selected, the immediate operand is assigned a specific value.

The formal semantics of the instruction is expressed as:

$$\phi_{instr}(\vec{I}, A, O)$$

where the tuple of input parameters $\vec{I}$, the internal attribute parameter A and the output parameter O form the instruction's interface. For example, the semantic of *ADD rd rs1 rs2* is:

$$\phi_{ADD}(I_1, I_2, O) ::= (O = I_1 + I_2)$$

A library is formed by a set of specifications of components, which are expressed as:

$$\left\{\langle \overrightarrow{I^j}, \overrightarrow{A^j}, O^j, \Phi_j(\overrightarrow{I^j}, \overrightarrow{A^j}, O^j)\rangle | j = 1, ..., N\right\}$$

where all variables $\overrightarrow{I^j}$, O^j are distinct, and $\Phi_j(\overrightarrow{I^j}, \overrightarrow{A^j}, O^j)$ is a semantic model for one component that belongs to one of three classes

- *Native Instruction Class (NIC):* The semantics of the component are equivalent to the chosen instruction. For example, for the *ADD* instruction:

$$\Phi(I_1, I_2, O) ::= \phi_{ADD}(I_1, I_2, O)$$

- *Derived Instruction Class (DIC):* Derived versions of instructions can be constructed as components, where the immediate operands are treated as internal attributes rather than being taken as inputs. For example, an *ADDI* instruction with a specific immediate operand can be derived as follows:

$$\Phi(I_1, A, O) ::= \phi_{ADDI}(I_1, A, O) ::= \Big(O = I_1 + sext(A)\Big)$$

  where A is a specific 12-bit immediate operand and $sext(A)$ denotes sign-extension of A to 32bits.
- *Composite Instruction Class (CIC):* To extend the coverage of *SEPE-SQED* to include instructions that are difficult to synthesize under bit-vector theory, such as multiplication of two 32-bit variables, which is hard for SMT solvers, we construct the *CIC* in this paper. *CIC* is designed to represent the semantics of a specific instruction sequence:

$$\Phi(\vec{I}, \vec{A}, O) ::= \phi_1(\overrightarrow{I^1}, A^1, O^1) \prec ... \prec \phi_N(\overrightarrow{I^N}, A^N, O^N)$$

where $\prec$ denotes the ordered sequence between instructions, each variable in $\overrightarrow{I^j}$ is either an input variable from $\vec{I}$, or a temporary output O^k such that $k < j$. The output of the last instruction serves as the output of the entire sequence of instructions, i.e., $O = O^N$. In this way, we can relax the conditions for solving. For example, to include multiplication instructions, we can allow operations that involve multiplying a 32-bit variable with a 32-bit constant:

$$\Phi(I_1, A, O) ::= \phi_{ADDI}(A, O^1) \prec \phi_{MUL}(I_1, O^1, O) \quad \Leftrightarrow$$
$$\Phi(I_1, A, O) ::= \Big(O = I_1 \times \big(0 + sext(A)\big)\Big)$$

To ensure that input parameters of components with different bit widths are restricted to sources of the same width, we refer to Buchwald et al.'s $\psi_{wfp}(L)$ [12]. In addition, we introduce an *input constraint* to eliminate cases where the synthesized program is identical to the original instruction g. The constraint can be defined as follows:

$$\Big(Name\big(\phi_g(\vec{I}, O)\big) == Name\big(\Phi_j(\overrightarrow{I^j}, O^j)\big)\Big) \Rightarrow L(\vec{I}) \neq L(\overrightarrow{I^j})$$

This constraint ensures that the synthesized program is not identical to itself, as self-equivalence would degrade into *SQED*.

4.2 CEGIS Based on the Highest Priority First

For the classical CEGIS [11], the number of components can cause considerable performance issues. The iterative CEGIS algorithm [12] can produce a large number of multisets through *combinations with replacement* algorithm (refer to Section 2.2), rendering it impractical to exhaustively enumerate them within a reasonable time frame. Therefore, we propose *HPF-CEGIS* (Algorithm 1).

In *HPF-CEGIS*, each component j is assigned a priority determined by choice weight c_j and exclusion weight e_j. A higher c_j indicates a higher priority, while a higher e_j value indicates a lower priority. Initially, the weights of all components are recorded in a global dictionary (line 2). Selecting the multiset with the *highest priority* (line 9, 10) before synthesis based on two factors:

- If a multiset contains some components with the same name as the original instruction, its priority is reduced to minimize the overlap between the data paths covered by the original instruction and its semantically equivalent program. For instance, we prefer using *{SUB t1 rs1 rs1, SUB t2 t1 rs2, SUB rd rs1 t2}* instead of *{SRAI t1 rs1 0x0, ADD t2 rs2 t1, SRAI rd t2 0x0}* to represent *ADD rd rs1 rs2*.
- If a multiset can synthesize the original instruction, the priorities of its components are increased due to their significant semantic similarity to the original instruction. Conversely, the priorities of its components are decreased.

The calculation of the priority for a multiset with n components is as follows:

$$priority = \frac{\Sigma_{j=1}^{n}(c_j - \alpha \times \chi_j)}{\Sigma_{j=1}^{n} e_j} \qquad \chi_j = \begin{cases} 1 & Name(j) == Name(g) \\ 0 & Name(j) \neq Name(g) \end{cases}$$

Here χ_j is a characteristic function indicating whether the type of the component j matches the original instruction g, and α is the influencing factor.

If the synthesis fails (CEGIS returns $None$, line 12), the priorities of all components in the current multiset are reduced by increasing the values of their exclusion weights (line 13). Otherwise, the priorities of that multiset's components are enhanced by increasing the values of their choice weights (line 16). The iteration stops once the number of synthesized programs reaches a predefined threshold (line 19).

Algorithm 1 HPF-CEGIS

```
 1: procedure PRIORITYITERATION(G: {Original instruction}, B: {Component})
 2:     PRIORITYDICT ← {comp_1:[c_1, e_1], ... ,comp_N:[c_N, e_N]} ▷ Initializing the weights of the components
 3:     R ← ∅
 4:     for each g ∈ G do
 5:         MULTISETS ← COMBINATIONSWITHREPLACEMENT(B, n)    ▷ The combinations with replacement algorithm
 6:         M ← ∅
 7:         stop ← False
 8:         while not stop do
 9:             SORTED(MULTISETS, PRIORITYDICT, g) ▷ Sorting in descending order of priority
10:             S ← MULTISETS[0]                    ▷ The highest priority first
11:             P ← CEGIS(g, S)        ▷ Generating semantically equivalent program
12:             if P == None then
13:                 Increasing the exclusion weight of components in S
14:             else
15:                 M ← M ∪ {P}
16:                 Increasing the choice weight of components in S
17:             end if
18:             if LEN(M) > k then
19:                 stop ← True
20:             end if
21:         end while
22:         R ← R ∪ {(g, M)}
23:     end for
24: end procedure
```

5 PROCESSOR VERIFICATION WITH SEPE-SQED

The correspondences between the original instructions and their semantically equivalent programs are stored in R (Algorithm 1 line 22). These correspondences (as shown in Listing 1) guide us in implementing the EDSEP-V transformation.

Following the *QED* consistency comparison principle, the input and output registers of the original instructions are mapped to corresponding registers in the semantically equivalent instruction sequences. Additionally, some intermediate inputs and outputs of semantically equivalent instruction sequences also require register allocation. To keep the triggering logic of the *QED-ready* signal simple, i.e., the number of register writebacks in the original instruction sequence is equal to the number of register writebacks in the semantically equivalent instruction sequence, we divided the register file into three parts. For a processor with 32 general-purpose registers, the segmentation is as follows:

$$O ::= \{regs[0], ..., regs[12]\}$$
$$E ::= \{regs[13], ..., regs[25]\} \quad \forall o \in O : \exists e \in E : o \mapsto e$$
$$T ::= \{regs[26], ..., regs[31]\}$$

The register allocation scheme assigns the register set O to the original instructions, while the register sets E and T are allocated to the semantically equivalent instruction sequences. Registers in O are paired one-to-one with registers in E, while registers in T serve as intermediate inputs and outputs. To maintain the data dependencies of the original instructions, the allocation of registers in T must adhere to the read-after-write principle. According to the correspondences in Listing 1, the transformation of *SUB rd rs1 rs2* is depicted in Listing 2.

Figure 2 illustrates the integration of the EDSEP-V module into the DUV's RTL during verification. The solver symbolically enumerates the original instructions under the ISA for execution, while concurrently the EDSEP-V module transforms them into corresponding

```
#Original instruction
SUB regs[1](rd) regs[2](rs1) regs[3](rs2)

#Semantically equivalent instruction sequence
XORI regs[26](t1) regs[15](rs1) 0xfff
ADD regs[27](t2) regs[26](t1) regs[16](rs2)
XORI regs[14](rd) regs[27](t2) 0xfff
```

Listing 2: EDSEP-V transformation

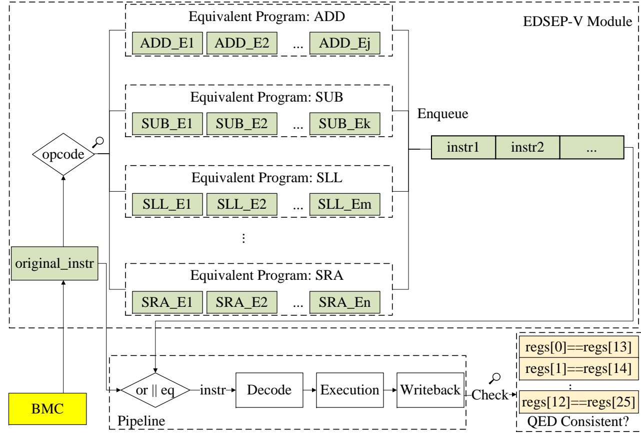

Figure 2: *SEPE-SQED* **verification model**

semantically equivalent instruction sequences. These semantically equivalent instruction sequences are stored in a queue. Based on a selection signal ($or||eq$), choose to dispatch the original instruction or semantically equivalent instruction into the pipeline. When the number of committed original instructions and their semantically equivalent counterparts is the same (determined by the number of register write-backs belonging to O or E), the model checker checks whether the state is *QED*-consistent:

$$QED\text{-ready} \Rightarrow \bigwedge_{i=0}^{12} regs[i] == regs[i+13]$$

SEPE-SQED can detect all types of logic bugs that can induce changes in processor architectural states in pre-silicon verification. On one hand, it shares the capability of *SQED* to systematically enumerate different combinations of instructions, effectively constructing various conditions that trigger *multiple-instruction bugs*. On the other hand, the original instructions and their semantically equivalent counterparts have the same functionality but different structures, thereby avoiding a single instruction bug simultaneously affecting both the original instructions and semantically equivalent sequences, leading to false positives.

6 EVALUATION

Our experimental evaluation consists of two parts. The first part involves comparing the time overhead of synthesizing the desired instruction sequences between *HPF-CEGIS* and two previous CEGIS approaches. The second part tests the bug discovery capabilities of *SEPE-SQED*. The experiments were conducted on an Intel(R) Core(TM) i9-10900K CPU with 64 GB RAM running at 3.70 GHz.

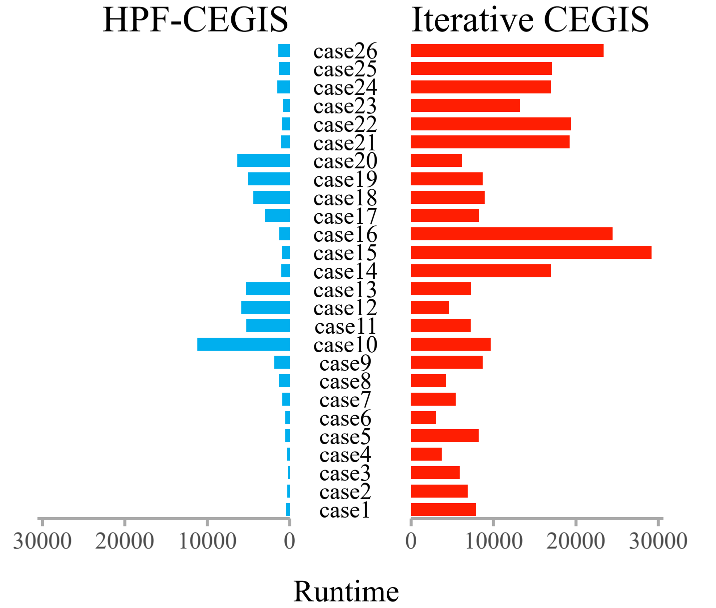

Figure 3: The time overhead of instruction synthesis

6.1 Synthesis Algorithm

We evaluated the time performance of three CEGIS algorithms, namely HPF-CEGIS, iterative CEGIS [12], and classical CEGIS [11], in synthesizing programs with equivalent semantics. Our library consists of 29 components, including 10 NICs, 10 DICs, and 9 CICs (see Section 4.1). These components collectively provide functional coverage for RV32IM instruction classes. In HPF-CEGIS, we set the initial values of each component's weight $[c_j, e_j]$ and influencing factor α to 1 and incremented the weights by 1 with each update. For each original instruction, if 20 semantically equivalent programs consisting of at least three components have been successfully synthesized, the synthesis process will terminate early. Otherwise, all possible combinations of components will be systematically enumerated.

Classical CEGIS [11] failed to synthesize a single original instruction even after several weeks of experimentation with the library of 29 components. When comparing *HPF-CEGIS* with iterative CEGIS, for the sake of fairness, we shuffle all multisets before synthesis to prevent the clustering of similar data types. Figure 3 illustrates the time overhead of synthesizing different cases using two algorithms, indicating that HPF-CEGIS significantly reduces synthesis time, exhibiting an average overall reduction in synthesis time of 50%, with synthesis time reduced by up to 90% in certain cases.

6.2 Real RTL Verification

To assess the efficacy of *SEPE-SQED* in detecting logic bugs, we conducted mutation testing on RIDECORE, an advanced superscalar and out-of-order processor core. The RTL code was converted into the BTOR2 [18] intermediate format through Yosys [19], and Pono [20] was employed as the model checking engine.

Our approach primarily focuses on enhancing *SQED* to extend its capability for checking *single-instruction bugs*. While it is also feasible to develop specialized formal properties for *single-instruction bugs* [4, 5, 9], as mentioned in the introduction (Section 1), these

Table 1: Injected single-instruction bugs

Type	Function	SEPE-SQED	SQED
ADD	Addition of two register types	3410.93s	-
SUB	Subtraction of two register types	1436.46s	-
XOR	Exclusive-OR	430.47s	-
OR	Bitwise OR of two register types	1765.66s	-
AND	Bitwise AND of two register types	777.79s	-
SLT	Set if Less Than	3306.27s	-
SLTU	Set if Less Than, Unsigned	2437.10s	-
SRA	Shift Right Arithmetic	76.50s	-
MULH	Multiply High	2837.13s	-
XORI	Exclusive-OR Immediate	627.45s	-
SLLI	Shift Left Logical Immediate	1837.11s	-
SRAI	Shift Right Arithmetic Immediate	85.44s	-
SW	Store Word	288.62s	-

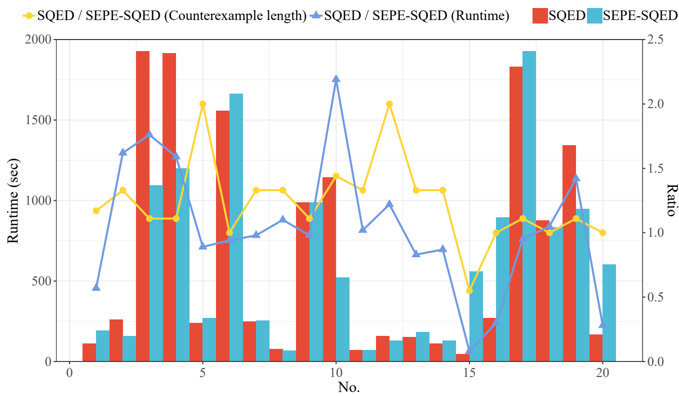

Figure 4: Detection results of multiple-instruction bugs

properties are non-universal and require much effort, we do not require a performance comparison with them. Table 1 demonstrates the detection results of *SEPE-SQED* for injected *single-instruction bugs* in RIDECORE.

The EDSEP-V module (*SEPE-SQED*) is relatively more complex compared to the EDDI-V module (*SQED*). We also conducted tests to determine if this complexity resulted in significant overhead in detecting *multiple-instruction bugs*. The x-axis of Figure 4 represents the bug identifier, while the red and blue bars depict the detection time of *SQED* and *SEPE-SQED*. The blue curve represents the detection time ratio of *SQED* to *SEPE-SQED* for the same bug, while the yellow curve represents the counterexample length ratio of *SQED* to *SEPE-SQED* for the same bug.

Both methods are capable of detecting injected *multiple-instruction bugs*. *SEPE-SQED* not only does not incur significant time overhead, but in some cases, it exhibits shorter bug detection time and counterexample traces compared to *SQED*. We attribute this to the fact that in contrast to the single pattern of matching original and duplicated instructions in *SQED*, *SEPE-SQED* can trigger a more diverse sequence of instructions that lead to bugs. As a result, in certain scenarios, the solver can find a shorter bug trace.

7 CONCLUSION

In this paper, we propose *SEPE-SQED* for processor model checking. The processor's correctness is established by comparing the consistency of its behavior with both original instructions and their semantically equivalent instruction sequences. To achieve this, program synthesis techniques are employed to discover programs that exhibit semantic equivalence to the original instructions. To improve the process of program synthesis, we present *HPF-CEGIS*, an efficient CEGIS algorithm based on a highest-priority first approach. Experimental results highlight the noteworthy enhancements in program generation speed attained by *HPF-CEGIS* and confirm *SEPE-SQED*'s ability to detect both *single-instruction* and *multiple-instruction bugs* in an open-source processor.

REFERENCES

[1] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model checking*. MIT press.

[2] Werner Damm, Amir Pnueli, and Sitvanit Ruah. 1998. Herbrand automata for hardware verification. In *CONCUR'98 Concurrency Theory: 9th International Conference Nice, France, September 8–11, 1998 Proceedings 9*. Springer, 67–83.

[3] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. 1998. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD*. Vol. 1522. Springer, 369–386.

[4] Alastair Reid et al. 2016. End-to-end verification of processors with isa-formal. In *International Conference on Computer Aided Verification*. Springer, 42–58.

[5] Clifford Wolf. 2018. Risc-v formal verification framework. https://github.com/YosysHQ/riscv-formal. (2018).

[6] Eshan Singh, David Lin, Clark Barrett, and Subhasish Mitra. 2018. Logic bug detection and localization using symbolic quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[7] Eshan Singh et al. 2019. Symbolic qed pre-silicon verification for automotive microcontroller cores: industrial case study. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1000–1005.

[8] Florian Lonsing, Karthik Ganesan, Makai Mann, Srinivasa Shashank Nuthakki, Eshan Singh, Mario Srouji, Yahan Yang, Subhasish Mitra, and Clark Barrett. 2019. Unlocking the power of formal hardware verification with cosa and symbolic qed. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[9] Keerthikumara Devarajegowda, Mohammad Rahmani Fadiheh, Eshan Singh, Clark Barrett, Subhasish Mitra, Wolfgang Ecker, Dominik Stoffel, and Wolfgang Kunz. 2020. Gap-free processor verification by s 2 qed and property generation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 526–531.

[10] Keerthikumara Devarajegowda and Wolfgang Ecker. 2018. Meta-model based automation of properties for pre-silicon verification. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 231–236.

[11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46, 6, 62–73.

[12] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 300–313.

[13] David Lin, Ted Hong, Yanjing Li, S. Eswaran, Sharad Kumar, Farzan Fallah, Nagib Hakim, Donald S. Gardner, and Subhasish Mitra. 2014. Effective post-silicon validation of system-on-chips using quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[14] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without bdds. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 193–207.

[15] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54, 9, 69–77.

[16] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 404–415.

[17] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2.

[18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2, btormc and boolector 3.0. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer, 587–595.

[19] Clifford Wolf. 2016. Yosys open synthesis suite. (2016).

[20] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark W. Barrett. 2021. Pono: A flexible and extensible smt-based model checker. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II* (Lecture Notes in Computer Science). Alexandra Silva and K. Rustan M. Leino, (Eds.) Vol. 12760. Springer, 461–474. DOI: 10.1007/978-3-030-81688-9_22.