Early Verification of ISA Extension Specifications using Deep Reinforcement Learning

Niklas Bruns
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
niklas.bruns@dfki.de

Daniel Große
Chair of Complex Systems, Johannes Kepler University Linz, Austria
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
daniel.grosse@jku.at

Rolf Drechsler
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
drechsler@informatik.uni-bremen.de

ABSTRACT

For IoT devices the demand in faster execution and at the same time lower energy consumption is a pressing problem. A very promising solution are *Application-Specific Instruction-set Processors* (ASIPs). They make use of custom instructions, which are added to the processor, forming the *Instruction-Set Extension* (ISE) of a given *Instruction Set Architecture* (ISA). While the selection process for the ISE is already challenging, an incorrect ISE specification leads to severe problems: errors and security vulnerabilities go undetected in the first formalization and in the worst case show up ultimately in the final implementation. In this paper, we propose an early verification approach for ISE specifications. Our novel approach is based on two ingredients: (i) *Virtual Prototypes* (VPs) to enable a rapid creation of an executable specification for the ISE; and (ii) *Deep Reinforcement Learning* (DRL) to search for ISE programs which violate the ISE specification intent. As case study we consider extensions of the RISC-V base ISA. We demonstrate the effectiveness of our approach for finding functional bugs in the executable specification of the ISE as well as specification gaps in the ISE leading to information leakage.

CCS CONCEPTS

• **General and reference → Verification**; • **Theory of computation → Reinforcement learning**; • **Hardware → Application specific instruction set processors**.

KEYWORDS

Early Verification, ISA Extension, Deep Reinforcement Learning

ACM Reference Format:
Niklas Bruns, Daniel Große, and Rolf Drechsler. 2020. Early Verification of ISA Extension Specifications using
Deep Reinforcement Learning. In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20), September 7–9, 2020, Virtual Event, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3386263.3406901

GLSVLSI '20, September 7–9, 2020, Virtual Event, China

ACM ISBN 978-1-4503-7944-1/20/09…$15.00
https://doi.org/10.1145/3386263.3406901

1 INTRODUCTION

The *Internet-of-Things* (IoT) is the leading cause of a technological revolution which enables many innovative products. The rapid development of new IoT products results in high demand for energy-efficient, cheap, and customizable hardware. Here, *Application-Specific Instruction-set Processors* (ASIPs) are a very promising solution. ASIPs make use of custom instructions which are added to the processor, forming the *Instruction-Set Extension* (ISE) of a given *Instruction Set Architecture* (ISA) [18]. Typical custom instructions are, for example, a *Multiply ACcumulate* (MAC) extension, a cryptographic extension or an extension for control-flow protection (e.g. [29]). There has been a lot of research on finding and selecting the specialized instructions, in particular, to speed up the execution of an application (for an overview see [11]). However, verification of the ISE for the IoT becomes even more critical as functional errors or security vulnerabilities would affect millions or even billions of devices. An important approach for ASIP design are *Architecture Description Languages* (ADLs) [20]. In essence, an ADL is a domain specific language for the description of a processor. The ADL description is then used to generate *Instruction-Set Simulators* (ISSs), HDL models and tools for the SW development. Even verification infrastructure like OVM/UVM based tests can be generated [24]. While the generator approach has several advantages, the focus of the ADL flow are clearly the generated models. More precisely, the ADL description is the first formalization of (the ISA and) the ISE specification. If functional bugs or specification gaps become part of the ADL description, the consecutive generation results are flawed. The same problems arise if classical development flows are employed, which directly implement the specification instead of using a generator.

Contribution: In this paper, we propose an early verification approach for ISE specifications. The goal of the proposed approach is to find and fix flaws in the ISE specification to prevent the propagation of errors through the whole development flow. Our novel approach is based on two ingredients: (i) *Virtual Prototypes* (VPs) to enable a rapid creation of an executable specification for the ISE. VPs are an established industry practice and typically written in SystemC [12, 17]. (ii) *Deep Reinforcement Learning* (DRL) to search for ISE programs which violate the ISE specification intent. The specification intent is thereby captured in *behavioral rules*. If during the DRL process, a violation of a behavioral rule is detected, the ISE developer can compare the violating program execution trace against the specification to determine the deviation from the specification intent. As case study we consider extensions of the RISC-V base ISA. We demonstrate the effectiveness of our approach for

finding functional bugs in the executable specification of the ISE as well as specification gaps in the textual ISE leading to information leakage.

Related Work: The specific problem of verifying specifications of instruction set extensions received only little attention so far. In [2] an approach to verify the conformity of a configurable system-model of a *Cyber-Physical System* (CPS) to its specification is presented. The authors use DRL to find a falsifying input for the CPS system-model in Matlab. In our approach we target the verification of the ISE specification using DRL and search for ISE programs which violate the ISE specification intent. For SW testing with the view on applying these concepts at the instruction level to the executable specification/ISS, constrained random techniques or fuzzing can be used. In this light closest to our work is *FuzzerGym* [8]. It combines reinforcement learning and fuzzing. The goal of the approach is to use reinforcement learning to optimize the mutation selection. Unlike FuzzerGym our reward calculation uses behavioral rules which are derived from the specification and our input space is much larger.

2 PRELIMINARIES

2.1 Deep Reinforcement Learning

Reinforcement Learning (RL) is a class of machine learning algorithms that are intended to solve sequential decision-making problems with only a reward as feedback [25, 26]. The goal of an RL algorithm is to maximize the cumulative reward by controlling the system. The theoretical formalism for RL is the *Markov Decision Process* (MDP). In this model, an environment consists of a set of states $S = \{s_1, ..., s_2\}$ and actions $A = \{a_1, ..., a_n\}$ which can be executed to control the systems state. The transaction function $T : S \times A \times S \rightarrow [0, 1]$ delivers the probability that a state action tu leads to a specific state. The algorithm/agent can choose an action and percepts the changes in the state and gets a reward $R : S \times A \times S \rightarrow \mathbb{R}$. Through these actions the agent gains knowledge about how to optimize its behavior. The argument γ is called discount factor which models the fact that future reward is worth less than immediate reward. An essential aspect is the decision whether an action leads to a positive or negative reward which cannot be done right after a single action execution. This problem is called the temporal credit assignment problem. To tackle this problem, it is beneficial to provide the agent with rewards for reaching intermediate subgoals. *Deep Reinforcement Learning* (DRL) is a combination of RL and deep neuronal networks which allows to solve more complex problems with high-dimensional state spaces [3].

2.2 RISC-V

It is very difficult to add custom instructions to traditional ISAs because they are often very complex. In contrast, as RISC-V is an open and free ISA, extendability was an important design factor of the RISC-V ISA. Hence, we use it as a case study in this paper. The RISC-V ISA emerged from UC Berkeley and recently attracted a lot of attention in industry. The ISA standard is maintained by the non-profit RISC-V foundation. The ISA consists of a mandatory base integer instruction set and various optional extensions. In this work we consider the configuration with 32 bit registers which is denoted as RV32I. For more details we refer to [27, 28]. Furthermore, as an executable specification of the RISC-V ISA we use the open source VP from [14].

Name	Source Registers	Dest. Registers	Description
MACL	rs1, rs2, rs3	rd	calculates the lower bits of the MAC operation
MACH	rs1, rs2	rd	calculates the higher bits of the MAC operation

Name	31 27	26 25	24 20	19 15	14 12	11 7	6 2	1 0
MACL	rs3	00	rs2	rs1	000	rd	01010	11
MACH	00000	01	rs2	rs1	000	rd	01010	11

$DL \leftarrow ((A \cdot B \wedge 0x0000FFFF) + C) \wedge 0x0000FFFF)$
$OF \leftarrow ((A \cdot B \wedge 0x0000FFFF) + C) \wedge 0xFFFF0000)$
$DH \leftarrow ((A \cdot B \wedge 0xFFFF0000) + OF$

Figure 1: MAC ISE Specification for RISC-V RV32I ISA

3 ISA EXTENSIONS

The *Instruction-Set Architecture* (ISA) of a processor is the interface between the HW and the SW. As motivated in the introduction, ASIPs add custom instructions to the ISA forming the so called *Instruction-Set Extension* (ISE)[1]. In general, an ISE yields several advantages [11]: (i) more dense code which reduces the code size, (ii) fewer executed instructions which can reduce power and (iii) faster execution of an application heavily using the custom instructions. Many domain-specific architectures include instances of ISEs; DSPs for example include multiply-accumulate extensions. The typical starting point for an ISE is application profiling. In this step, the computationally most demanding segments which are known as hot-spots are identified. The second step is ISE identification which uses previously determined hot-spot information and identifies instructions which should optimize the performance/power consumption. Then, the ISE is specified in detail, i.e. the functionality of each instruction is defined as a human readable document. To evaluate the potential performance gains *Instruction-Set Simulators* (ISSs) as part of an *Virtual Prototype* (VP) are used in modern design flows [7, 21]. Therefore, the existing ISA ISS is extended. However, this leads to substantial problems if the ISE specification contains bugs or gaps, as in this case the following development steps require costly long-loop iterations. Hence, we propose an early verification approach for ISE specifications. Before we describe our approach in detail, we use a simple ISE as running example.

Example 1. *Our running ISE example for RISC-V RV32I is the specification of a* Multiply ACcumulate *(MAC) custom instruction which works on integers. The ISE specification is shown in Fig. 1. The upper table lists the instructions including the used source/destination registers (2nd and 3rd column) and the description of each instruction (forth column). The lower table defines the encoding of the instructions. In addition to the instruction name, the columns show the bit positions and what is stored in the respective fields.* $rs1/rs2/rs3$ *stands for the different source registers and* rd *for the destination register. The functionality of the instructions is defined below the tables. In this definition DL stands for destination lower, OF for overflow and DH for destination higher. The semantic of the considered MAC instruction is defined as* $d \leftarrow a + (b \cdot c)$*. Because the output of the MAC instruction can be much larger than the input, the MAC instruction has been split into the two instructions MACL and MACH. The MACL instruction*

[1] As a side note, since Moore's Law is slowing down, ISEs are also very common for regular CPU ISAs, for instance Intel's x86 [4].

calculates the lower 32 bits while the MACH calculates the higher 32 bits of the result. To handle the overflow of the addition after the multiplication, MACL saves the overflow for later usage. Because of the existence of the saved overflow, MACH needs to be executed directly after the corresponding MACL.

4 EARLY VERIFICATION OF ISA EXTENSION SPECIFICATIONS USING DEEP REINFORCEMENT LEARNING

In this section we present our early verification approach for ISE specifications. The main goal of the proposed DRL approach is to search for ISE programs which violate the ISE specification intent. Fig. 2 depicts the overview. We start with the textual specification of the ISE for a given ISA. Besides the regular practice of extending the ISA ISS (see action *ISS Extension* in Fig. 2), our approach requires to capture the specification intent in form of behavioral rules that are extracted from the ISE (see action *ISE Behavior Extraction* in Fig. 2). For this, we use a logic-based language which we introduce in Section 4.1. Just like DRL, our behavioral rules work on state action tuples $S \times A \times S$. Hence, they can easily be transformed into a *Markov Decision Process* (MDP) (cf. Section 2.1). To search for ISE programs which violate the ISE specification, we transform this problem, which consists of the behavioral rules and the extended ISS, into an MDP and perform DRL. There are two possible results of the DRL process: (a) violations of the ISE specification have been generated (in form of a test vectors at the instruction level, i.e. ISE programs) or (b) no violating programs have been found. In case (a) we store these programs as *Intent Violations* (IVs). They can be used to improve the ISE specification as well as the ISS. In the following, we present more details on the behavioral rules (Section 4.1), the MDP transformation (Section 4.2) and the implementation (Section 4.3).

4.1 ISE Behavioral Rules

The behavioral rules are used to define the intended and non-intended behavior of the ISE. Formally, a behavioral rule is defined as a function $br : (S \times A \times S) \rightarrow \mathbb{B}$ where S is the set of states and A the set of actions. In the context of the considered specification verification problem this means: For the given state s_{t-1} at time point $t-1$, the to be executed instruction i_{t-1} and the successor state s_t at time point t, the value of $br(s_{t-1}, i_{t-1}, s_t)$ determines whether the instruction i_{t-1} behaves like intended. This function definition allows direct integration into DRL since the function arguments perfectly match reward function arguments. The primitive objects available in the behavioral rules are sets of constants C, variables V, instructions I, and states S. Then, the syntax of behavioral rules is defined as follows:

1 $br(s_{t-1}, i_{t-1}, s_t) ::= \neg br | br \vee br | read \circ read$
2 $\circ \in \{=, \neq, \leq, \geq, <, >\}$
3 $read ::= v | c | s_{t-1}(idx) | s_t(idx) | read \sim read$
4 $\sim \in \{+, -, \cdot, \div\}$

As defined in Line 1, every behavioral rule can be negated or connected with another behavioral rule by performing a logic operation. Moreover, every read term which is connected to another read term via a comparator (Line 2) results also in an behavioral rule (Line 1). Every read term (see Line 3) can consist of variables, constants or the value which results from reading from the state vector at index idx. A read term can be the result of a arithmetic function too (Line 4). We show a simple behavioral rule for our running example in the following. Later in the experiments we provide more behavioral rules examples.

Example 2. *We consider again the running example, i.e. Example 1. Besides the core functionality of the MAC instructions, a major verification goal is that no side effects on the general purpose registers occur. We discuss the application of our approach for this intended behavior in the following examples. As defined in the upper table of Fig. 1, there is only a single destination register per MAC ISE instruction. Hence, we capture the no side effect intent in the following behavioral rule :*

$$\forall r \in Regs\ r \neq dest(i_{t-1}) \rightarrow s_t(r) = s_{t-1}(r)$$

As can be seen we formalize that: if r is not the destination register of an ISE instruction i_{t-1}, then the value of r is the same as the time point before, i.e. remains unchanged. Note, by the use of $\forall$, the behavior is formulated as a single rule. During the transformation, the quantifiers are unrolled. Because all unrolled clauses describe a single behavior, there are no intermediate goals in this example.

4.2 Problem Transformation

In this section, we present the details about the transformation of the verification problem to an MDP. To create an MDP, a reward function $R : S \times A \times S \rightarrow \mathbb{R}$ has to be defined. As mentioned earlier, S is the set of states and A is the set of actions. The state of the MDP is the memory and the registers that are used for the instruction execution. The state must include the memory to make the functionality deterministic for an observer who only sees the actions/instructions and the state, to fulfill the Markov property. This allows us to assume that the DRL algorithm has all relevant information to make decisions. To ensure a small state space, other memory should not be included in the state. For our verification problem, set A is the set of all instructions of the ISE. To define the reward function, we transform the behavioral rules to reward rules. The first step of this transformation is to negate the behavioral rules because our verification goal is to find instruction sequences that violate the specification intent. These behavioral rules are combined with the reward value 1 to define the positive reward rules that describes the goal of the MDP. The behavioral rules which define the non-intended behavior are combined with the reward value -1 to define the reward rules that describes the negative goal of the MDP. Note, the RDL algorithm wants to omit the negative goals and only wants to find the positive goals. The combination of this two sets of behavioral rules define the reward function. Furthermore, the rewards are chosen analog to [19], a positive reward for 1 at on winning (kill enemy), and -1 on losing (suicide).

Subgoals: To accelerate the verification process, we generate so-called potential rules (see [22]). The goal of the potential rules is to define intermediate subgoals like the *positive reward for object pickup* in [19]. In the context of our verification problem this is similar to the approach of virtual coverage (see [10]). In order to create the subgoals, the reward rules are transformed into their CNF-Form. The clauses of the CNF-form reward rules are the potential rules. The reward of the individual potential rules is the multiplicative

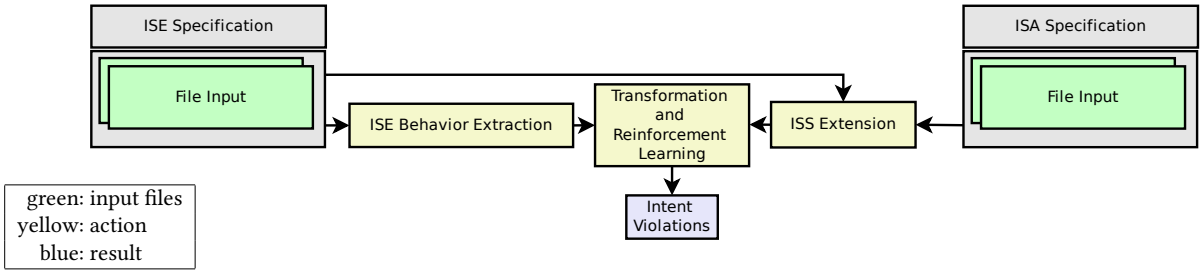

Figure 2: Overall Flow of ISE Verification using Deep Reinforcement Learning

inverse of the underlying reward. Because we choose a γ parameter smaller than 1, the DRL algorithm converges to a short solution. This convergence is a helpful for the problem at hand as short ISE programs can be analyzed more easily.

4.3 Implementation

We have implemented the proposed approach based on OpenAI Gym [5]. Besides that, we use an executable specification/ISS for the base RISC-V ISA which is the SystemC-based RISC-V VP from [14, 15]. To overcome the considerable SystemC initialization overhead, we enhanced the RISC-V VP with a checkpointing system. The checkpointing system has been implemented as a fork-based TCP server. This allows a much faster simulation of ISE programs generated during the DRL process as well as multi-threaded DRL. The pseudo-code for the DRL ISE simulator using the enhanced VP is shown in Algorithm 1. Note that the DRL algorithm connects several times to this simulator (depending on the available CPU cores). The first step of the simulator algorithm is the SystemC model initialization (see Line 1). After that, the VP parses all available instructions (including their parameters; see Line 2). Therefore, the input can be viewed as the set of all possible actions. Next, the TCP server starts (see Line 4). If a DRL client connects, the simulation is forked (see Line 5 and Line 6). After that, the VP receives the initial state and loads it (see Line 8 - Line 9). As long no error occurs, the VP receives an action (instruction) and executes it (see Line 11 - Line 13). After an action execution, the VP sends the resulting successor state to the DLR algorithm. (Line 15). If an run-time error occurs (e.g. a trap), the VP closes the TCP connection and closes its fork subprocess. Please note that the defined behavioral rules are checked as part of the DRL algorithm (due to space limitation we cannot discuss the details).

Example 3. *The implementation of our proposed DRL approach generates an instruction sequence which violates the behavioral rule (Section 4.1) of our running MAC example in 2.41s. From the resulting trace we can easily see that a general purpose register (in the trace it was t0) has been used to store the overflow (OF) of the instruction MACL. This flaw resulted from the specification, since in Fig. 1 it was not defined how the overflow should be managed. Unfortunately, during the ISS extension it was decided to use the general purpose register t0 for this task. Overall, this first example demonstrates that our approach is able to find specification flaws.*

In the next section, we present the experimental evaluation for a larger case study.

Algorithm 1: DRL ISE Simulator

```
   Input: ISA and ISE I
1  Initialization of SystemC model
2  Parse I
3  while test vector generation is running do
4      start TCP server
5      if new DRL connection accepted then
6          fork simulation
7          send list of available instructions to DRL algorithm
8          receive initial state
9          load state in VP
10         while no error do
11             receive instruction from DRL algorithm
12             if instruction is invalid then terminate connection
13             execute instruction
14             if error at execution then terminate connection
15             send successor state to DRL algorithm
```

5 EXPERIMENTAL EVALUATION

For the evaluation of our proposed approach we used in addition to OpenAI Gym the PPO2 [23] DRL algorithm from the framework *Stable Baseline* [16] with the default hyperparameters that are also used for the Atari benchmark. In the following we consider a cryptographic ISE to extend the RISC-V RV32I ISA. We detail the ISE in Section 5.1. Then, in Section 5.2 we present the obtained results for the proposed approach.

5.1 AES ISE Specification

The cryptographic ISE has the purpose of accelerating cryptographic operations for the algorithm AES128. AES is a block cipher which uses in its 128bit version 10 encryption rounds [9]. For a detailed description of the AES standard we refer to [6]. The extension at hand aims to offer a high-performance cryptographic accelerator which securely stores the keys. Secure means in this context that the extraction of the keys should be impossible.

The instructions of the extension, which are inspired by the AES-NI Extension for x86 processors [13], are listed in Table 1. The registers of the ISE are called *High Confidential Registers* (HCRs). The instruction GHMOV is defined to move data from the extension registers to the *General-Purpose Registers* (*GPR*). *GH* in the instruction name denotes that the destination of the value from a *HCR* is a *GPR*. Overall, the instruction takes three arguments. Besides the destination register and source register the third argument specifies the subword of the data which should be copied. This argument is required because the ISA uses 32bit GPRs and the extension 128bit HCRs. The instruction HGMOV is for moving data from the general-purpose registers to the extension register. It has three arguments. The arguments of the instruction are almost

the same as for the GHMOV. However, HG denotes that the destination of the value from a GPR is a HCR. The position parameter defines at which position of the word of the HCR the subword of the GPR should be copied. Furthermore, additional security flags are encoded in the position parameter. With these security flags the data can be marked as data or key. The instruction XOR realizes the exclusive-or operator for the extension registers. The instruction only operates on the extension registers. The instruction is needed to realize the preliminary round of AES. The instructions ENC, ENCLAST, DEC, and DECLAST are the core of the considered extension. They have three arguments. The first argument is the destination register which holds the result of the instructions. The second argument specifies the used round key, and the third argument the to be processed data. The instruction ENC is needed for the first nine encryption rounds, and the ENCLAST is for the tenth and therefore last encryption round. Equivalent instructions arguments apply for the instructions DEC and DECLAST. To guarantee confidentiality values that are marked as data or key can not be moved to a GPR. Values will be unmarked after the last round of the operations ENCLAST or DECLAST. The instruction KEYGEN realizes the generation of the encryption round keys. It has three arguments. The first argument is the destination register which holds the generated round key. The second argument is the previous round key. The last argument is RCON which denotes the round constant and is needed to add resistance against invariant attacks.

5.2 AES ISE Specification Verification

In this section, we present the verification of the AES ISE specification.

AES1: As defined in the AES ISE specification secret data (like the encryption key) must be stored securely. This specification intent can be found in the ISE specification in Table 1 at the instruction HGMOV in column *Source 2* in *ProtFlag* and the corresponding textual descriptions of the instructions. Hence, we captured this intent in the following behavioral rule :

$$\forall r \in HCRRegs : s_t(r) = KEY \rightarrow s_t(r.flags) = PROT$$

As can be seen, we formalize that: if a register of the ISE contains the key, then the register flag must be set to protected. Our proposed approach finds an instruction sequence which violates the behavioral rule in 0.01s seconds. This fast "unprotection" is the result of a ISE specification gap.

AES2: The specification does not contain the requirement that 10 AES rounds must be executed before the result can be marked as unprotected. After we fixed this flaw by inserting the requirement in the specification and then integrating the respective fix in the extended ISS, we check the protection of the key with the aforementioned behavioral rule again. However, our proposed approach finds again an instruction sequence which violates the behavioral rule in 795.97s seconds. The flaw resulted from the ISE specification as it was not specified that the preliminary round of the AES encryption must be executed before starting the regular AES rounds. The output of the preliminary round is a combination of the key and the data. The data input (Source 2) of the first call of the instruction *ENC* must be the combination of the key and the data from the preliminary round and not only one of them. To fix this flaw, a new instruction can be introduced, which realizes the preliminary round and the first encryption round at once.

AES3: As defined in the row for instruction GHMOV in Table 1, only non-secret values can be moved to a general-purpose register. Hence, we capture this intent in the following behavioral rule :

$$\begin{aligned}\forall r_1, r_2, r_3, r_4 \in GPRRegs : \neg(&s_t(r_1) = KEY_{subword_0} \ \wedge \\ &s_t(r_2) = KEY_{subword_1} \ \wedge \\ &s_t(r_3) = KEY_{subword_2} \ \wedge \\ &s_t(r_4) = KEY_{subword_3})\end{aligned}$$

As can be seen, we formalize that the full secret key (consisting of four subwords) can not be in GPR registers at time t. The automatic generated intermediate rewards contain the subgoal that any can hold a subword of the secret key. Our proposed approach finds a violating instruction sequence for this behavioral rule in 2.35s seconds. The shortest (found after 4.04s) contains 4 *GHMOV* instructions which move the four subwords (32bit) of the key (key length 128bit) to different GPRs:

```
GHMOV x8, h0, 3
GHMOV x13, h0, 2
GHMOV x5, h0, 1
GHMOV x9, h0, 4
```

The flaw results from the code of the reference ISS since the protection check in the instruction GHMOV was buggy.

AES4: After fixing the protection check in the code of the extended reference ISS, we run the proposed DRL approach again for the behavioral rule . As mentioned earlier, this formulation has the meaning that all GPR registers at time t can not contain the secret key. Our proposed approach results in a timeout, i.e. that no intent violation has been found within the time limit of 24 hours.

5.3 Comparison

The goal of this section is to provide a comparison to *Constrained Random Verification* (CRV), which is a well known verification technique for instruction generation [1, 30]. It is constrained to generate only valid instructions and also uses the behavioral rules for goal detection. The results for this comparison have been obtained on an Intel Xeon Gold 5122 CPU with 3.60GHz host using a time limit of 24 hours. A summary of the ISE specification verification using our proposed approach is shown in Table 2. The table also provides a comparison of our approach to a CRV approach, which however does not use DRL for searching violating ISE programs. The goal of the comparison is to show the effectiveness of our DRL approach. The first column *Name* presents the scenarios as described in the previous section. The next four columns report the results of our DRL-based approach. The last four columns list the results of the CRV approach. The result columns are structured as follows: The first column contains the computation time until a flaw has been detected (marked with ✓, otherwise ✗). The next column contains the length of the found solution (i.e. number of instructions). The next two columns contain the time until the shortest solution was found and the length of the shortest solution. *T.O.* denotes that the time limit has been reached. As can be seen AES1 shows that our approach can quickly find specification flaws. AES2 points out that our DRL based approach can be used to verify complex specification verification problems while the CRV approach fails. AES3 shows

Table 1: AES Instruction Set Extension (ISE) Specification for RISC-V RV32I ISA

Type	Opcode	Dest.	Source	Source 2	Description
I	GHMOV	GPR	HCR	Subword	moves the not secret value of the HCR register to the GPR register
I	HGMOV	HCR	GPR	Position+ProtFlag	moves the value of the GPR register to the HCR register
R	XOR	HCR	HCR	HCR	XOR operation
R	ENC	HCR	HCR (key)	HCR (data)	encrypts register data with key for one AES round
R	ENCLAST	HCR	HCR (key)	HCR (data)	encrypts register data with key for the last AES round
R	DEC	HCR	HCR (key)	HCR (data)	decrypts register data with key with one AES round
R	DECLAST	HCR	HCR (key)	HCR (data)	decrypts register data with key for the last AES round
I	KEYGEN	HCR	HCR (key)	RCON	generates the aes round keys with the round constant
R	IMC	HCR	HCR (key)	-	generates corresponding decryption round key from the encryption key

I-type: 2 register operands (with dest) and imm. R-type: 3 register operands (with dest) GPR: General Purpose Reg. HCR: High Confidential Reg. RCON: Round Constant

Table 2: Comparison of proposed DRL approach vs CRV

Name	Goal	DRL				CRV			
		first		best		first		best	
		time	len	time	len	time	len	time	len
AES1	UNPROT	0,01s ✓	2	0.01s ✓	2	2.78s ✓	3	4.16s ✓	2
AES2	UNPROT	795,97s ✓	215	795,97s ✓	215	T.O. ✗	N.A.	T.O. ✗	N.A.
AES3	MOVE	2.35s ✓	5	4.04s ✓	4	0.25s ✓	21	0.32s ✓	13
AES4	MOVE	T.O. ✗	N.A.	T.O. ✗	N.A.	T.O. ✗	N.A.	T.O. ✗	N.A.

that our approach finds better/smaller solutions in slightly more run-time compared to the CRV approach. Overall, the proposed approach was able to uncover significant flaws including non-trivial information flow.

6 CONCLUSIONS

In this paper, we have proposed an early verification approach for ISE specifications using deep reinforcement learning. In the experiments we have been able to show that the integration of an enhanced ISA simulator with OpenAI-Gym-based DRL allows to search for non-trivial specification flaws in the initial executable specification for an ISE. As a case study, we considered a complex cryptographic AES ISE of the RISC-V RV32I ISA. We have found functional bugs as well as specification gaps in the ISE leading to information leakage.

Acknowledgments: This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K and within the project VerSys under contract no. 01IW19001.

REFERENCES

[1] Allon Adir, E. Almog, L. Fournier, E. Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. 2004. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *D&T* 21 (04 2004), 84 – 93.

[2] Takumi Akazaki, Shuang Liu, Yoriyuki Yamagata, Yihai Duan, and Jianye Hao. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *Formal Methods*. 456–465.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. 2017. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine* 34, 6 (Nov 2017), 26–38. https://doi.org/10.1109/MSP.2017.2743240

[4] Andrew Baumann. 2017. Hardware is the New Software. In *Workshop on Hot Topics in Operating Systems*. 132–137.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:1606.01540.

[6] Joan Daemen and Vincent Rijmen. 2013. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

[7] Tom De Schutter. 2014. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press.

[8] William Drozd and Michael D Wagner. 2018. FuzzerGym: A Competitive Framework for Fuzzing and Learning. *arXiv preprint arXiv:1807.07490* (2018).

[9] Morris J Dworkin, Elaine B Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, and James F Dray Jr. 2001. *Advanced Encryption Standard (AES)*. Technical Report.

[10] Laurent Fournier and Avi Ziv. 2008. Using Virtual Coverage to Hit Hard-To-Reach Events. In *HVC*. 104–119.

[11] Carlo Galuzzi and Koen Bertels. 2011. The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2 (May 2011), 18:1–18:28.

[12] Daniel Große and Rolf Drechsler. 2010. *Quality-Driven SystemC Design*. Springer.

[13] Shay Gueron. 2008. Advanced encryption standard (AES) instructions set. *Intel, http://softwarecommunity.intel.com/articles/eng/3788.htm* 25 (2008).

[14] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *FDL*. 5–16.

[15] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. 2020. RISC-V based Virtual Prototype: An Extensible and Configurable Platform for the System-level. *JSA* (2020).

[16] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable Baselines. https://github.com/hill-a/stable-baselines.

[17] IEEE Std. 1666 2011. *IEEE Standard SystemC Language Reference Manual*. IEEE Std. 1666.

[18] Kingshuk Karuri, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. 2009. A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs). In *SAMOS*. 204–214.

[19] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS games with deep reinforcement learning. In *AAAI*. 2140–2146.

[20] Prabhat Mishra and Nikil Dutt. 2008. *Processor Description Languages*. Morgan Kaufmann.

[21] Lee Moore, Duncan Graham, Simon Davidmann, and Felipe Rosa. 2018. Cycle Approximate Simulation of RISC-V Processors. In *Embedded World Conference*.

[22] Andrew Y Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, Vol. 99. 278–287.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv:1707.06347* (2017).

[24] Marcela Šimková, Zdeněk Přikryl, Zdeněk Kotásek, and Tomáš Hruška. 2013. Automated functional verification of application specific instruction-set processors. In *International Embedded Systems Symposium*. 128–138.

[25] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). MIT Press, Cambridge, MA, USA.

[26] Martijn van Otterlo and Marco Wiering. 2012. *Reinforcement Learning and Markov Decision Processes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–42. https://doi.org/10.1007/978-3-642-27645-3_1

[27] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.

[28] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.

[29] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 214–226.

[30] Jun Yuan, Carl Pixley, and Adnan Aziz. 2006. *Constraint-based verification*. Springer Science & Business Media.