

Investigation of Reinforcement Learning-based Attack on Logic Locking

Jake Mellor, Allen Shelton, and Sara Tehranipoor

*Electrical and Computer Engineering Department
Santa Clara University, Santa Clara, California, USA*
{jmellor, ashelton, ftehranipoor}@scu.edu

Abstract—Logic Locking is an emerging form of hardware obfuscation that is intended to be a solution to many of the trust issues associated with the modern globalized IC supply chain. By inserting extra key-gates into a circuit, the functionality of the circuit can be locked until the correct order of bits or “key” is applied to the key gates. To assess the strength of logic locking techniques, we propose a new attack that uses “deep reinforcement learning”. This research aims to test the robustness of logic locking as well as evaluate reinforcement learning as a possible attack against logic locking. We decided to use a Deep Q-Learning Neural Network to implement logic locking. If traditional Q-Learning were to be used, a massive Q-value table would be generated for all the possible states the key could be in, which is an exponential function of the key’s size. By creating other supporting modules to enable the reinforcement learning model to run simulations with any desired key and receive a reward back based on the correctness of these outputs when compared against the unlocked circuit, or an oracle, the model can be reinforced to learn. The unpredictable nature of the outputs of a logic locked circuit creates a weak correlation between the accuracy of any key the model tries, and the correctness of the outputs. This lack of correlation makes it difficult to properly reinforce the model, making it unable to converge on a correct key, even after over 100,000 timesteps of training.

I. INTRODUCTION AND MOTIVATION

In the age of globalized supply chains, electronics producers are in need of ways to obfuscate their designs from untrusted foundries. Many design houses have shifted to outsourcing the fabrication of their integrated circuits (IC) because of the economic feasibility of using an offshore foundry. The trade-off is the decrease in security and privacy. The use of these untrusted foundries has led to many security issues such as IC piracy, cloning, and overproduction. The security flaws in ICs have also allowed attackers to extract sensitive information from these systems. The ICs at this stage are vulnerable to Trojans, physical tampering, and logic attacks. This has prompted researchers to develop security mechanisms to include circuits to counter these attacks.

Hardware obfuscation is the idea that the structure and logic of an IC are hidden to prevent adversaries from accessing it [1]. Many forms of hardware obfuscation have been developed to hide the functionality of an IC. One form of hardware obfuscation is logic locking. Logic locking aims to protect the IC by inserting key gates to hide the logic in a netlist. The inserted key gate is usually in the form of a logic component like an XOR, XNOR, or MUX. These key gates are inserted

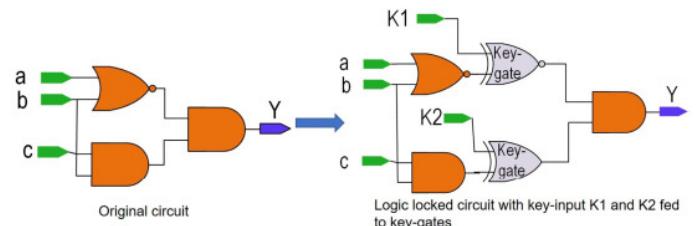


Fig. 1. Basic Logic Locking Implementation

between logic components of a netlist and have an input of a key bit. When multiple key gates are inserted into the design, only a correct key will unlock the circuit. This is shown in Figure 1. Otherwise, the key gate will change the logic of the system and generate a different output for the IC, adding an extra layer of security to logic circuits.

Since Logic Locking is a very recently proposed solution to the trust issues associated with the global supply chain, much research and experimentation is required in order to reach a position where there is a strong and robust Logic Locking scheme that can be widely used to protect IC designers from lost revenue. We plan to help push the Logic Locking research forward by evaluating a never before used attack that leverages reinforcement learning. Our research will hopefully either produce a successful attack that give insights on the strengths of Logic Locking, or will demonstrate the kind of attacks that Logic Locking is able to successfully defend against.

In this paper we review the basics of Logic Locking and reinforcement learning. We then dive into the mechanics of Deep Q-Learning, and then review our implementation of the Reinforcement Learning Attack. Our results will then be reviewed and we will cover what was successful with this attack, what was not effective, and draw some conclusions. When beginning this research project, we had the following goals: Being able to completely extract the key from any locked circuit, the attack taking only a few hours to complete, and that we would learn more about where Logic Locking can be improved.

Specifically, we have the following contributions in the paper:

- Novel deep Q-Learning model development
- An implementation of an attack that uses the new model

- Deep analysis of the strengths of Logic Locking against a reinforcement learning based attack

II. RELATED WORKS

There are many approaches to attacking Logic Locking. However, there are only a handful of attacks that use machine learning and neural networks to extract the key of a locked circuit. In 2019, Tehranipoor et al. [7] proposed BOCANet, the first logic locking attack that uses deep learning through a Deep Recurrent Neural Network. This paper notes that the amount of time and resources required to mount a SAT attack limits its effectiveness. BOCANet seeks to solve these shortcomings. BOCANet's deep learning approach to circuit deobfuscation gives it a much faster execution time than SAT attacks. The main strength of this attack is how little information is required to conduct the attack. The D-RNN model only requires less than 0.5% of possible IO pairs to train, and it's able to achieve great results. The use of a D-RNN also unlocks the ability to simply train I/O pairs to predict outputs from inputs without the need a key at all. The attack seems to scale well a circuit size and key size increases. And given the variety of possible deep learning architectures and specifications of training parameters, there is room to increase the performance of this attack even further.

Other attacks that use machine learning are SAIL [6] and SURF [2], in 2018 and 2019, respectively. Chakraborty et al. proposed SAIL as structural a structural attack that uses machine learning to deobfuscate the circuit. This attack's effectiveness comes from the fact that it uses machine learning methods to exploit the vulnerabilities of obfuscation techniques to discover design intent. SURF is a combination of a structural and functional attack that uses SAIL as a first step to deobfuscate the circuit, and then generate a seed key that is iterated upon until a correct key has been found. This allows SURF to quickly converge on the correct key rather than starting from a random seed, giving it a high accuracy. The weakness of these attacks is that they both require the unlocked circuit, or an oracle, to function correctly.

III. PRELIMINARIES

A. Reinforcement Learning

As mentioned earlier, we will be using a reinforcement learning algorithm in order to carry out our hardware attack. Reinforcement learning is an area of machine learning that seeks to train intelligent agents to take actions in an environment that maximizes its cumulative reward. At each time step, an agent receives an observation, which includes the current state and reward. Then the agent chooses an action from the set of available actions and it is sent to the environment. Then the environment changes in some way as a result of that action and moves to a new state and the next reward associated with that transition is determined. This feedback loop continues as the agent learns more information about its environment and how to make the best actions. For our project, we needed to define our problem, a logic locking hardware attack, in the context of a reinforcement learning model in order for this method to

work, since reinforcement learning has never been used before in a hardware attack. At the very least, we needed to define our agent, our environment, what constitutes an action, how we define a state, and a reward function. These definitions are shown here:

- **Agent:** Key bits (or the algorithm for finding the key bits)
- **Environment:** Circuit trying to be deobfuscated
- **Action:** Flipping 1 key bit
- **State:** The values of the key bits
- **Reward:** Output accuracy from a fixed number of inputs with the current key value compared to the correct outputs with correct key value

B. Deep Q-Learning

In traditional reinforcement learning, specifically in the type used in our project called Q-learning, how an agent learns about its environment is by using what is known as a Q-table. In a Q-table, one axis has all possible states, and the other axis has all possible actions. As an agent gains experience by taking actions from different states, a Q-value is determined for each state-action pair, where a Q-value denotes the future return of taking an action from a given state. The values in this table will be iteratively updated as the agent continues to explore its environment and find the optimal path to achieve its goal.

As the state space becomes too large, however, traditional Q-learning becomes infeasible. The benchmark circuit we're trying to attack has anywhere from 32 to 256 key bits. Even with 32 bits, that maps to 2^{32} possible states, so trying to store a Q-table for this attack, if possible, would make it extremely slow and inefficient. So we have decided to use deep reinforcement learning, which combines deep learning and reinforcement learning. Instead of using a Q-table to store and update Q-values, a neural network is used to approximate the Q-function, and this network is trained on the experiences of the agent in its environment. The network takes in the current state, which is the key value, as input, and each output neuron will be the approximate Q-value for every action that can be taken from the state that was passed into it.

Note that in traditional Q-learning, the Q-table is updated by finding the loss between the calculated Q-value from the current state-action pair and the optimal Q-value, which is given by the Bellman equation:

$$q_*(s, a) = E[R_{t+1} + \gamma \max q_*(s', a')] \quad (1)$$

This equation can be interpreted as the optimal Q-value for a given state action pair is equal to the expected value of the sum of the immediate reward from that state-action pair and the scaled maximum Q-value of the next state among all possible next actions. This scaling factor is called a discount factor, which takes into account how much the reinforcement learning algorithm will take into account future returns, which is what the Q-value represents, in relation to immediate return. In deep reinforcement learning, the loss between the current and optimal Q-values is used to train the network weights through gradient descent and backpropagation.

IV. DESIGN

Our plan for this project was to design a attack against logic locking using reinforcement learning. We would then implement this attack on various benchmark circuits using an existing logic locking technique. For our attack, we were to design and train a reinforcement learning model, as well as an algorithm around it that will be able to extract the correct key bits from benchmark logic locked circuits. The benchmark circuits we used were obfuscated using random, logic cone analysis, and SLL methods [13]. After developing this algorithm, we will need to fine tune the hyperparameters of this algorithm in order to increase its performance, both in terms of accuracy and time. We hoped to achieve 100% key accuracy with this attack, as well as have an execution time that competes with other existing state-of-the-art attack methods. All of our code was written in Python.

Our algorithm required the creation of a number of different components that would all work together. We created a netlist parser for our benchmark circuits. This netlist parser will be used to extract the input signals, output signals, and key bits from the netlist files. This parser will be used in conjunction with the simulation manager so that our algorithm can pass inputs into the benchmark circuits along with the key values that it generates and capture the outputs. The simulation manager acts as an intermediary between our attack algorithm and the netlist circuit. As our hardware attack continues to run, it will generate new key values that need to be tested on the benchmark circuit. The simulation manager will take this key value, along with a number of random inputs, and generate a Verilog testbench to pass these inputs into the circuit and capture the outputs. The oracle comparison module will take the circuit outputs that were obtained with a given set of inputs and the key generated from the algorithm and compare them to the outputs from those same set of inputs with the correct key value in order to generate a reward for our algorithm. The higher the percentage of output bits that are correct, the higher the reward will be. Since this initial reward scheme that was generated is a percentage, its value will be between 0 and 1. As you will see later, we slightly modified our reward scheme to try to improve the performance of our reinforcement learning model.

The most important component of our algorithm is our reinforcement learning model. The model will be trained by generating key values, deciding actions to take, generating rewards based on these actions, and passing these experiences into a neural network to learn how to find the correct key value. There were three main sections of our reinforcement learning model, the deep Q-network, the epsilon greedy strategy, and the replay memory. The motivation for a deep Q-Network was discussed in Section III. Both the current Q-value and the “optimal” Q-value used in calculating the loss require a pass through the Q-network in order to be calculated, since we do not actually know the optimal Q-values ahead of time. Using the same neural network to find both values will cause network optimization to be chasing its own tail, since both the current

Q-values and the optimal Q-values are being changed at each training step. We want the optimal Q-values to be somewhat fixed. To solve this, we are using two neural networks for our reinforcement learning algorithm. The policy net will have weights that are trainable based on the calculated loss. The target net will have fixed weights and are used to find the optimal Q-values, and every certain number of timesteps, the target net weights will be updated to match the policy net weights.

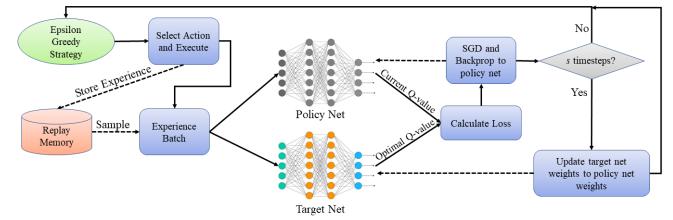


Fig. 2. Deep Q-learning Training Loop

We chose to incorporate an epsilon greedy strategy in order to mitigate how the agent chooses actions, where as mentioned earlier, an action is flipping one bit in the current key value. This strategy tries to find an optimal balance between exploration, where the agent explores the environment to discover more information about it, and exploitation, where the agent exploits the information that is already known about the environment in order to maximize its return. Both are useful for the agent to navigate its environment, and using only one of them will not lead to good results. The epsilon greedy strategy incorporates both through a parameter called an exploration rate, and the pseudo code for the strategy is as follows:

Algorithm 1: Epsilon Greedy Strategy

```

p = random();
if p < ε then
    | pull random action;
else
    | pull current best action;
end
  
```

At every timestep of the algorithm, a random number is generated between 0 and 1. If the number is less than the exploration rate, a random action is chosen (exploration), else the action that promises the highest immediate return is chosen (exploitation). In our algorithm specifically, determining the action that gives the highest immediate reward is found by passing the current state into the policy net, and choosing the action that corresponds to the output neuron with the highest value. To reiterate, each output neuron in the policy net represents the approximate Q-value, or long-term return, of taking each action from a given input state.

The last component of our reinforcement learning model is the replay memory. With a concept known as experience

Algorithm 2: Training Loop per Timestep

```

action = EpsilonGreedyStrategy(epsilon);
[reward, nextstate] = Execute(state,action);
ReplayMem.store(state,action,reward,nextstate);
if ReplayMem.size ≥ batchsize then
    samples = ReplayMem.sample(batchsize);
    current_q_values = PolicyNet(samples.states);
    next_q_values = TargetNet(samples.nextstates);
    max_q_values = Bellman(next_q_values);
    loss = MSE(max_q_values,current_q_values);
    optimizer = StochasticGradientDescent;
    optimizer.backprop(loss);
end
if (timestep % update) == 0 then
    | TargetNet.weights = PolicyNet.weights;
end

```

replay, the agent's experiences at each time step is stored in a data structure called the replay memory. An experience is defined as the tuple shown in Equation 2:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \quad (2)$$

where s_t represents the current state, a_t represents the action taken from that state, r_{t+1} represents the reward for taking that action, and s_{t+1} is the new state. A batch of these experiences will be sampled and used to train the Q-network at each timestep of the algorithm. The main advantage of using experience replay instead of providing the sequential experiences to the Q-network as they occur in the environment is to break the correlation between consecutive samples. If only consecutive samples are passed into the network, the sample would be highly correlated and lead to inefficient learning.

A flowchart of our deep Q-learning algorithm for a single timestep is shown in Figure 2, and the pseudocode for it is shown in Algorithm 2.

V. IMPLEMENTATION

In order for the reinforcement learning model to be able to get a reward from a simulation of the locked circuit, other modules were needed to allow the model to give a key to the circuit simulator and for it to get a reward back. This required the creation of a parser, simulation manager, and oracle comparison. When model began training, the parser would find the net names of the inputs and outputs and the key size, which was used to initialize the neural network and other supporting modules. Each timestep the reinforcement learning model would enter a new state by flipping a bit of the previous key, making each state the same as the key that the model was trying. This state was given to the simulation manager to be used to simulate the locked circuit with that key. The simulation manager would then generate random input stimulus, and store it in a file with the current key for a Verilog testbench module to read in the inputs and key and pass them

to the circuit, where the outputs are generated and stored in another file for the simulation manager to read out. The outputs were gathered for both the locked and unlocked circuit for the same input patterns, and then compared. In order to try to use the correctness of the locked circuit's outputs as an indicator that of the current key's correctness, multiple simulations were ran each timestep and their output correctness was averaged. This helped combat the unpredictability of using random input stimulus in a logic locked circuit. This variable, NLOOPS, was used to specify the number of simulations for the locked and unlocked circuit in each time step. If NLOOPS was set to 20, then each timestep 40 simulations would be ran, 20 simulations for the locked circuit and unlocked circuit each. From each batch of simulations, the total correctness would be averaged and that would be returned as the reward for the model to store in the reply memory. In order to more strongly reinforce the model, the average correctness of the outputs of the locked circuits for each time step was then passed through Equation 3:

$$R = e^{6T_{acc}-3} - 0.05 \quad (3)$$

This exponential reward curve gave better reinforcement for keys with accuracy on average, and poorer rewards for less accurate keys. All of the reinforcement learning model and the support modules were written in Python, and the circuits and their respective testbenches were in Verilog. Synopsys VCS was used to simulate the circuits. Below in Figure 3 the entire attacks flow and modules is depicted.

The model was trained on multiple ISCAS 85 Benchmark circuits, particularly C432 and C880A. These circuits were locked using 3 different logic locking schemes, Random Insertion, Fault Detection Insertion, and LCSB. This array of various circuits and logic locking schemes gave a healthy gauntlet of circuits to test the attack against. In addition, many of these benchmark circuits were used to evaluate other logic locking attacks, enabling comparisons between the Reinforcement Learning Attack and previously developed attacks.

VI. RESULTS AND VALIDATIONS

Once all the modules had been put together, the model was able to train for the specified number of episodes and timesteps. The model was able to change and update its weights during training, indicating that the architecture of the attack was correct. However, this implementation was unable to converge to a set of weights of each layer that would result in it being able to determine the correct key when put into evaluation mode after over 100,000 timesteps of training. This was due to the poor correlation between the correctness of the key that the model was evaluating in each time step, and the reward generated. Figure 4 shows data from a training run of 20,000 timesteps where the correctness of the current key is compared against the reward generated for that key, before the exponential reward function was added.

This data shows that a key that is only half correct can generate rewards that are almost as great as, or on average as

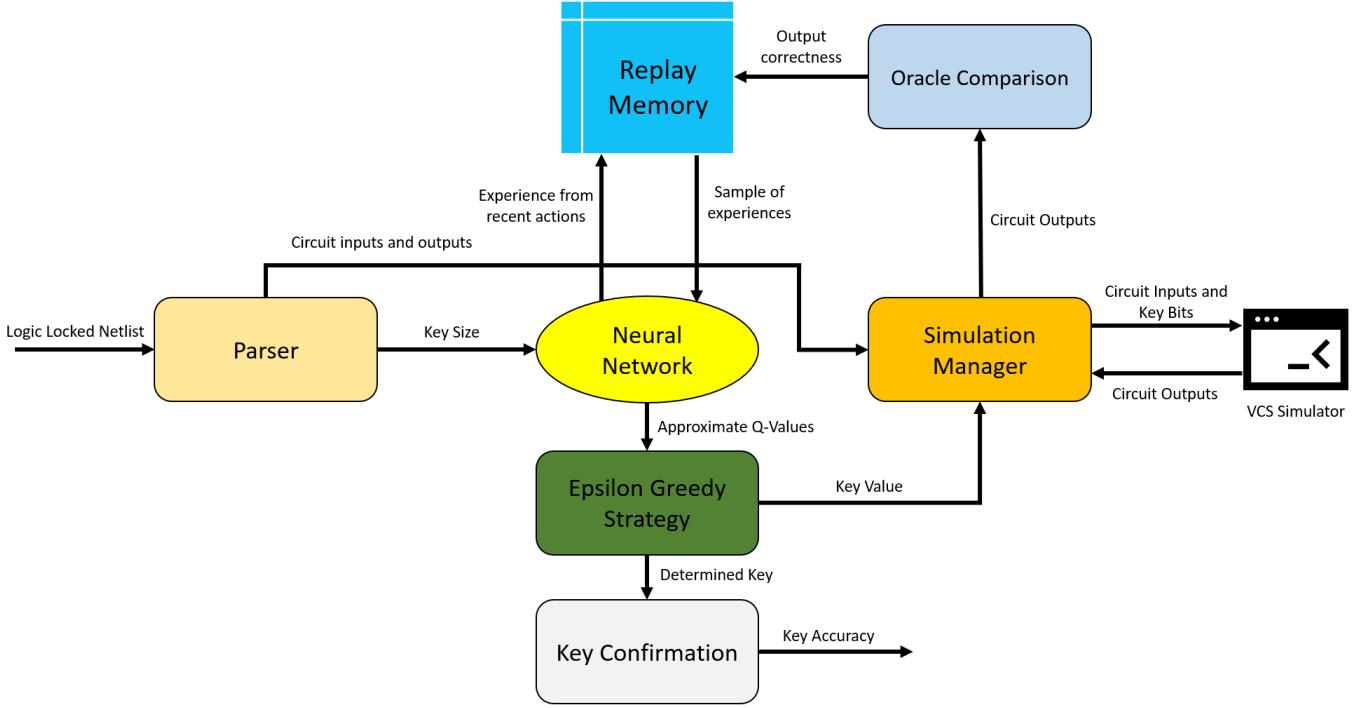


Fig. 3. Reinforcement Learning Attack Architecture

great as, a key that is almost entirely correct. Increasing the number of circuit simulations per timestep helps combat this, but quickly increases the training time of the model as more time is spent simulating the circuit. By changing the reward generation to use the equation described in Equation 2, a more favorable reward structure can be generated. The data from a training session of 25,000 timesteps is shown in Figure 5, which uses the exponential reward function. This data shows there is still not a strong enough correlation between the accuracy of the key that the model is currently trying, and the reward the model is receiving for the key. Much of the rewards generated were quite low on the scale for all key accuracy's, making it hard for the model to be properly reinforced when it is in a mostly correct state and mostly incorrect state. Rewards of higher value are much sparser. This ultimately comes down to the robustness of logic locking as a defense mechanism. The unpredictability that logic locking inserts into a circuit makes learning from the outputs difficult with random input stimulus.

Another important outcome of this attack was the training and testing time of the model. Many of the most effective logic locking attacks take somewhere from a few seconds to a minute or two to solve for the locked circuit's key. However, training the model for 100,000 timesteps took about 3 days of constant computation, even with specialized hardware acceleration. This time could be reduced by using a faster Verilog simulator, but that would not bring this implementation close to the execution time of other attacks. Once trained however, the model only takes a few seconds to be tested

and produce a key. However, the model needs to be trained for each individual circuit. If an attacker wanted to solve for the key of many identically locked circuits, a reinforcement learning model could do so quickly, once the initial training time has been completed.

VII. DISCUSSION AND CHALLENGES

There were a number of challenges faced that if resolved adequately could have resulted in better performance of our attack method. One significant issue we faced was the power of our hardware limiting how much we could train. We originally were used Santa Clara University's Linux machines to train the model, but these machines were quite slow. We decided to try using a personal PC which had a dedicated GPU with Tensor cores within it, and a higher speed processor than the Linux machines. Using GPU acceleration and overall faster PC, we saw a 47x speed up in training over the Linux workstations. This saved a lot of training time, but not enough to be competitive with other hardware attack methods. With more training the model would eventually learn, but with the current hardware being used, it would have taken far too long to be feasible.

In addition to needing faster hardware, a faster Verilog simulator could have further reducing training time. We originally started with Synopsys VCS to simulate the circuits, but this was severely limiting how fast we could train. When moving from the Linux workstations to our personal PC, we had to install a new Verilog simulator since we couldn't install VCS locally without paying for a license. We found Icarus Verilog,

which is an open-source Verilog simulator. This simulator was much faster than VCS and also helped in the massive speedup we saw over the Linux workstations. Even though Icarus Verilog did speed up the training process, it was still the limiting factor in how fast we could train. Per timestep of the algorithm, most of the time was spent waiting for the simulator to respond with circuit outputs, instead of doing operations that are training the model. Overall, this speedup was massively helpful in working on this project as it greatly reduced the amount of time we had to wait during training before we could analyze the data we logged and make improvements to try and get the model to function as desired.

Ultimately, there were issues in the design of our algorithm, such as the reward scheme, that we could not resolve. This shows that logic locking was an effective means of protecting logic circuits from our hardware attack.

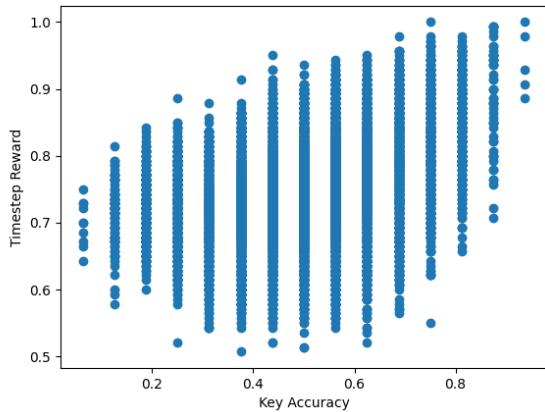


Fig. 4. Reward vs Key Accuracy Before Exponential Reward Function

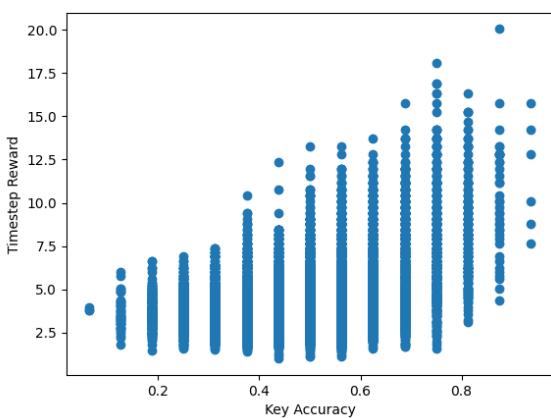


Fig. 5. Reward vs Key Accuracy After Exponential Reward Function

VIII. CONCLUSIONS

With this project, we sought to design and implement our own logic-locking circuit attack in order to assess the effec-

tiveness of reinforcement learning on this type of hardware security technique. This was the first use of reinforcement learning as a means of implementing a hardware attack.

We learned that current hardware attacks have already been developed by researchers to become more knowledgeable on the field, and we compiled that knowledge into a survey paper that was accepted to ICCE 2021 [14]. Inspired by the papers we read, we designed a new hardware attack using reinforcement learning. We obtained obfuscated benchmark circuits to test our attack on, programmed the individual components of our algorithm, and combined them into a complete system. We spent a considerable amount of time fine-tuning the hyperparameters of our systems as well as making small fundamental improvements to the algorithm with the purpose of increasing its performance in regards to time and accuracy. Ultimately, this attack was unable to accurately extract the correct key value from any of our benchmark circuits, showing that logic-locking is an effective security measure against our reinforcement learning-based hardware attack. Moreover, this paper shows the hardware security research field the infeasibility of the presented attack method against logic-locked circuits.

REFERENCES

- [1] Forte, D., Bhunia, S. and Tehranipoor, M.M. eds., 2017. Hardware protection through obfuscation. Springer International Publishing.
- [2] P. Chakraborty, J. Cruz and S. Bhunia, "SURF: Joint Structural Functional Attack on Logic Locking," 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 2019, pp. 181-190, doi: 10.1109/HST.2019.8741028.
- [3] Xu, X., Shakya, B., Tehranipoor, M. and Forte, D., 2017. "Novel Bypass Attack And BDD-Based Tradeoff Analysis Against All Known Logic Locking Attacks," Lecture Notes in Computer Science, volume 10529.
- [4] Y. Lee and N. A. Touba, "Improving logic obfuscation via logic cone analysis," 2015 16th Latin-American Test Symposium (LAT), Puerto Vallarta, 2015, pp. 1-6, doi: 10.1109/LAT.2015.7102410.
- [5] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan and Y. Jin, "AppSAT: Approximately deobfuscating integrated circuits," 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, 2017, pp. 95-100, doi: 10.1109/HST.2017.7951805.
- [6] P. Chakraborty, J. Cruz and S. Bhunia, "SAIL: Machine Learning Guided Structural Analysis Attack on Hardware Obfuscation," 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST), Hong Kong, 2018, pp. 56-61, doi: 10.1109/AsianHOST.2018.8607163.
- [7] Fatemeh Tehranipoor, Nima Karimian, Mehran Mozaffari Kermani, and Hamid Mahmoodi. 2019. Deep RNN-Oriented Paradigm Shift through BOCANet: Broken Obfuscated Circuit Attack. In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI '19). Association for Computing Machinery, New York, NY, USA, 335–338. DOI:<https://doi.org/10.1145/3299874.3318031>
- [8] Yuanqi Shen and Hai Zhou. 2017. Double DIP: Re-Evaluating Security of Logic Encryption Algorithms. In Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI '17). Association for Computing Machinery, New York, NY, USA, 179–184. DOI:<https://doi.org/10.1145/3060403.3060469>
- [9] D. Sironi and P. Subramanyan, "Functional Analysis Attacks on Logic Locking," in IEEE Transactions on Information Forensics and Security, vol. 15, pp. 2514-2527, 2020, doi: 10.1109/TIFS.2020.2968183.
- [10] F. Yang, M. Tang and O. Sinanoglu, "Stripped Functionality Logic Locking With Hamming Distance-Based Restore Unit (SFLL-hd) – Unlocked," in IEEE Transactions on Information Forensics and Security, vol. 14, no. 10, pp. 2778-2786, Oct. 2019, doi: 10.1109/TIFS.2019.2904838.
- [11] M. Yasin, J. J. Rajendran, O. Sinanoglu and R. Karri, "On Improving the Security of Logic Locking," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 9, pp. 1411-1424, Sept. 2016, doi: 10.1109/TCAD.2015.2511144.

- [12] M. Yasin, B. Mazumdar, J. J. V. Rajendran and O. Sinanoglu, "SAR-Lock: SAT attack resistant logic locking," 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, 2016, pp. 236-241, doi: 10.1109/HST.2016.7495588.
- [13] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan," in Proc. of the International Symposium on Circuits and Systems, 1985, pp. 663-698.
- [14] J. Mellor, A. Shelton, M. Yue and F. Tehranipoor, "Attacks on Logic Locking Obfuscation Techniques," 2021 IEEE International Conference on Consumer Electronics (ICCE), 2021, pp. 1-6, doi: 10.1109/ICCE50685.2021.9427730.