Development of Course Modules in Python for Hardware Security Education

Brooks Olney, Mateus Augusto Fernandes Amador, and Robert Karam
Department of Computer Science & Engineering
University of South Florida
Tampa, FL, USA
{brooksolney, mateusf1, rkaram}@usf.edu

***Abstract*—Year after year, computing systems continue to grow in complexity at an exponential rate. While this can have far-ranging positive impacts on society, it has become extremely difficult to ensure the security of these systems in the field. Hardware security - in conjunction with more traditional cybersecurity topics like software and network security - is critical for designing secure systems. Moving forward, hardware security *education* must ensure the next generation of engineers have the knowledge and tools to address this growing challenge. A good foundation in hardware security draws on concepts from several different fields, including fundamental hardware design principles, signal processing and statistics, and even machine learning for modeling complex physical processes. It can be difficult to convey the material in a manageable way, even to advanced undergraduate students. In this paper, we describe how we have leveraged Python, and its rich ecosystem of open-source libraries, and scaffolding with Jupyter notebooks, to bridge the gap between theory and implementation of hardware security topics, helping students learn through experience.**

I. Introduction

Computers are pervasive in almost every aspect of modern life, and ensuring their security is of paramount importance. This need has led to an increase in research and development in cybersecurity through national programs. As a result, the demand for cybersecurity professionals has risen, with universities offering traditional and online degree programs allowing students to specialize in software, information, and network security. In the past, hardware was thought to be inherently trustworthy; however, this view has changed in recent years, and the security of the underlying hardware is now considered crucial to ensuring the security of applications built upon it.

As a motivational example of the pervasive nature of computers and the associated security issues, we can consider the Internet of Things (IoT). This is an ongoing technology transition with the goal of connecting the unconnected. And so, network-facing sensors, controllers, and actuators are embedded in varied and diverse domains, such as transportation, healthcare, consumer electronics, public services, defense, and more [1]–[4]. These devices have long in-field lifetimes and are network-facing, occasionally receiving firmware updates, and routinely communicating with cloud services. As such, these devices are regularly exposed to potential attacks with significant economic losses and public safety risks [5], [6]. Due to the cyber-physical nature of IoT devices, their security necessarily integrates components of hardware and cybersecurity [7], [8]. In particular, hardware security has often been lacking in devices and overlooked by researchers in contrast to software security [9].

This material is based upon work supported by the National Science Foundation under Grant No. DGE-1954259. Contact: rkaram@usf.edu

To improve security assurance in all devices going forward, it is important to prioritize security as a design metric, while considering the potential threats and impact of an attack, if one is ultimately successful. But beyond that, it is necessary to address deficiencies in training the next generation of computer scientists and engineers [10], [11] - not only those who will play active roles in security assessment, but also those doing the initial design and development. Therefore, hardware security education should be more easily accessible to all students, and should be taught in a manner leveraging evidence-based pedagogical techniques to improve student learning in this area. Using hands-on, experiential learning, for example, enables students not only to learn the theory of what makes a system secure against a given attack, but also to practice the attack itself, implement a countermeasure, understand the metrics used to assess efficacy, and reflect on the impact of their work.

We have been working towards developing an introductory hands-on hardware security course that predominantly uses Python for coding tasks. The use of Python as a language for instruction has been linked to increased motivation and better student learning outcomes when compared to Java [12], due to its ease of use and deployment on different platforms. Tools such as Jupyter notebooks aid in scaffolding efforts. Course development and subsequent education research was funded by the National Science Foundation, and has resulted in the implementation of 10 hands-on hardware security experiments and associated course materials. All experiments involve hardware in some way - either through direct communication with development boards, through measurement of physical properties of the hardware, or through post-experiment analysis of data that students collected themselves in prior experiments. All experiments use Python and Jupyter notebooks designed to walk students through attacks and implementation of countermeasures, compartmentalize different aspects of the code, and encourage unit tests in subsequent cells, enabling students to check the efficacy of their attack or countermeasure as they progress through the experiments.

In this paper, we describe three example uses of Python and Jupyter notebooks in our course, along with relevant background on the hardware security topics themselves: 1) communication with hardware peripherals; 2) modeling of physical unclonable functions with neural networks; and 3) analysis of random number generators with a highly-optimized custom implementation of the NIST test suite in Python. Moreover, we provide relevant technical details on the development, optimization, and deployment strategy we used to help ensure a good course experience for the students.

II. Course Module Background

In this section, we provide background information on the hardware security primitives discussed in this paper, and their context in computer security and usage in the lab experiments.

A. Physical Unclonable Functions and Neural Networks

The first course module deals with *physical unclonable functions*, or PUFs. In semiconductor manufacturing, "process variation" refers to the fact that the individual transistors that make up integrated circuits (ICs) all have slightly different functional parameters due to tiny, uncontrollable differences in physical attributes. A difference in the dimensions (length, width, layer thickness, etc.) can impact functional properties, such as how long it takes the transistor to switch from on to off or off to on. Transistors are arranged in different ways to implement different logic functions (NAND gates, NOR gates, etc.) and that switching time manifests as *propagation delay* - the time it takes to see an output change after one or more input signals change. Typically, the variations among individual components will not be very significant, and on the whole, most of the ICs produced from the same design will end up having the same overall behavior, within some acceptable tolerance. However, some circuits can actually be designed to *amplify* the effect of these tiny variations so that they can be more easily measured. In turn, we can use this principle to create a signature that is unique to that circuit, like a digital fingerprint. Because these variations are too small to be intentional, it is physically impossible to perfectly copy them into another IC. This is the basis for PUFs, which are one of the basic "building blocks" or "security primitives" that are used in hardware security [13].

There are many different types of PUFs [14]–[16], and the particular one we introduce to students in the first experiment is the Arbiter PUF (APUF) [17], [18]. An APUF, shown in Figure 1, is a circuit consisting of two "racing paths" - a "top" path, and "bottom" path - that pass through a series of stages. At each stage, a multiplexor (MUX) can potentially swap the top and bottom paths, depending on the value of a *challenge* bit. Different paths can be selected with the application of different *challenge* bits. Each path will be comprised of different transistors, and hence, have very slightly different delay characteristics.

The *quality* of a PUF can be evaluated in two main ways: first, its *reproducibility*, and second, its *uniqueness* [13]. A PUF with good *reproducibility* will output the same value (0

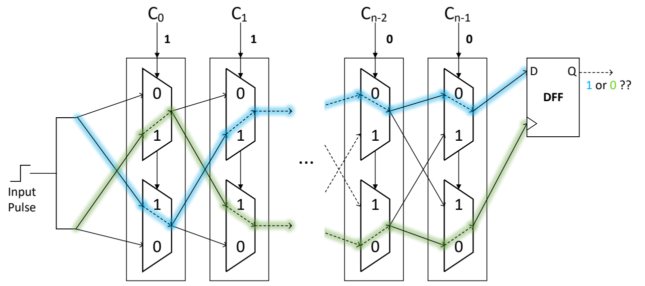

Fig. 1: Example Arbiter PUF with n stages (0 to $n-1$). Challenge bits C_0 through $C_n - 1$ control whether or not the top and bottom paths swap at a particular stage. A single pulse travels through the circuit, experiences different delays through each stage. Ultimately, the arbiter will record either a 0 or 1, depending on which signal arrived first.

or 1) for the same challenge consistently *on the same chip*. On the other hand, if you take the same PUF design on a *different chip*, a PUF with good *uniqueness* will output a 0 or 1 with close to 50% probability, which would in no way be related to the output on the original chip.

Both of these properties - reproducibility and uniqueness - can be easily measured using the Hamming distance metric. The output for the same challenge on the same chip should have a HD close to 0% - meaning it is highly reproducible. This is also known as the *intra-chip Hamming distance*. The output for the same challenge on two different chips with the same PUF design should have a HD close to 50%. This is also known as the *inter-chip Hamming distance*. A strong PUF should, at the very least, have good reproducibility and uniqueness. Students explore and evaluate these properties in the first experiment, directly interfacing with an FPGA to acquire PUF responses, and processing them offline in Python. In the next experiment, students learn that while a PUF may be considered *good* by these standards, and while it may be physically impossible to reproduce exactly on another device, its behavior may be modeled accurately using machine learning approaches.

Because of their excellence in recognizing patterns and learning features from highly complex data, ML-based approaches are well-suited to modeling attacks on PUFs. Some of the most commonly used algorithms in literature include Logistic Regression (LR), Support Vector Machines (SVMs), Evolution Strategies (ES), and DNNs [15], [16], [20]–[23]. Consequently, this has motivated hardware security researchers to devise enhanced PUF architectures that are resistant to such attacks [17], [24], [25]. In a sense, research efforts to engineer commercially usable PUFs *must* include some sort of countermeasure to mitigate threats of ML-based attacks, lest they be doomed to impracticality.

In the second experiment, students are tasked with using ML/AI to model a PUF. PUFs can be modeled using many different ML approaches, including Logistic Regression (LR),

TABLE I: All statistical tests recommended by the NIST Statistical Test Suite [19].

No.	Test Name	No.	Test Name
1	Frequency (monobit)	9	Maurer's "Universal Statistical" Test
2	Frequency within a block	10	Linear Complexity
3	Runs Test	11	Serial test
4	Longest run of 1s in a block	12	Approximate Entropy
5	Binary Matrix rank	13	Cumulative Sums
6	Discrete Fourier Transform (Spectral)	14	Random Excursions
7	Non-overlapping template matching	15	Random Excursions variant
8	Overlapping template matching	-	

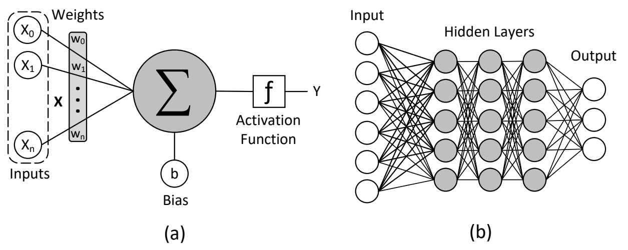

Fig. 2: (a) Diagram of simple neuron, where output is a function of the inputs and their respective weights. (b) Diagram of DNN showing the general architecture involving an input layer, multiple hidden layers of various sizes and a final output layer with a neuron for each class.

Support Vector Machines (SVMs), Evolution Strategies (ES), and Artificial Neural Networks (ANNs) [15], [16], [20]–[23]. Inspired by biology, ANNs are designed to mimic the behavior of the brain, in that "decisions" are made by an interconnected network of artificial neurons which loosely mimic the functionality of neurons in a brain. As shown in Figure 2(a), each neuron has one or more inputs – which can be from either raw data or a previous neuron – with a *weight* multiplied by each input and a single *bias* term added. The final value is then input to an *activation function*, which determines the neuron's final output. One of the most commonly used models in practice is a *deep neural network* (DNN) which has several layers of neurons between the inputs and outputs. An example of a DNN is shown in Figure 2(b). For this task, a simpler ANN network was used, though it does require some data preprocessing to succeed. Specific details on the preprocessing and ANN implementation are provided in Section III-B.

B. True Random Number Generators

The third experiment deals with randomness in hardware. Random number generation is a critical component of many aspects of computing, and especially within security domains. For instance, random number generators (RNG) are often used to generate random sequences of bits for cryptographic systems. Not all RNGs are of the same quality, especially for crypto applications, and there are important distinctions when assessing the suitability of a RNG for security applications.

First, RNGs can be categorized into a few different classes; namely, pseudo-random number generators (PRNGs), cryptographically secure pseudo-random number generators (CSPRNGs) and true random number generators (TRNGs). PRNGs typically use an algorithm to generate a sequence of random numbers. An example of a PRNG could be a linear feedback shift register (LFSR) which produces a (repeated) sequence of states that comprise the random bitstream. Consequently, they are not suitable for cryptographic applications, however enhancements may be made to make them suitable for cryptographic applications (i.e. CSPRNGs). True random number generators TRNGs are dubbed *truly* random because they generate random numbers by harnessing the inherent randomness of certain physical phenomena, rather than purely algorithmically. For example, thermal noise, clock jitter, atmospheric readings and even quantum interactions [26]–[30].

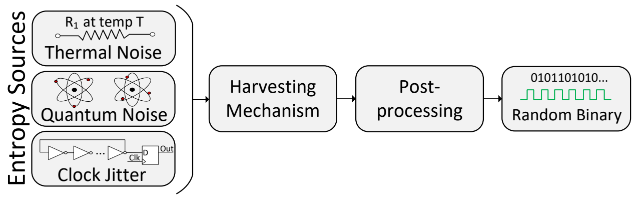

Fig. 3: Traditional structure of a TRNG. Consists of a physical source of entropy (randomness), a mechanism for harvesting that entropy, and a post-processing step to digitize the data from the harvesting mechanism which produces the final random binary data.

In general, a good TRNG consists of a few critical components [31]. First, the *entropy source* is the physical phenomena to be harnessed for generation and should be aperiodic and unpredictable. Second, the *harvesting mechanism* is the method the TRNG uses to perform the extraction of entropy from the source. The harvester should be designed in such a way that it does not disturb the process itself while collecting as much information as possible. Finally, depending on the nature of the entropy source and harvester, a post-processing step may be necessary to mask imperfections or provide tolerance to faults and tampering. A diagram of a general TRNG structure is provided in Figure 3. For certification, the TRNG should be subjected to a number of intense statistical tests to evaluate the randomness of its output. The NIST statistical test suite is one of the most widely used test suites for this purpose [19].

As seen in Table I, the NIST test suite consists of 15 statistical tests designed to test different features of binary sequences. In what follows we provide a brief description of each test: 1) *Frequency (monobit) test* focuses on the proportion of 0s to 1s in the entire sequence, 2) *Frequency test within a block* tests the proportion of ones within M-bit blocks, 3) *Runs test* focuses on the total number of runs of identical bits within the sequence, 4) *Longest run of 1s in a block* tests for the longest run of 1s within M-bit blocks, 5) *Binary Matrix rank test* focuses on the rank of disjoint (32×32) sub-matrices of the sequence, 6) *Discrete Fourier Transform (Spectral) test* focuses on the peaks in the DFT of the sequence and detects periodic features, 7) *Non-overlapping template matching test* computes the number of non-overlapping occurrences of a supplied bit template pattern, 8) *Overlapping template matching test* is similar to the non-overlapping template test, but the matches are allowed to overlap through the sequence, 9) *Maurer's "Universal Statistical" test* focuses on the number of bits between matching patterns in the sequence - essentially detecting whether the sequence can be compressed without information loss, 10) *Linear Complexity test* divides the sequence into N M-bit blocks and computes the linear complexity of each block, 11) *Serial test* focuses on the frequency of all overlapping m-bit patterns in the sequence, 12) *Approximate Entropy test* compares the frequency of overlapping blocks for block sizes of m and m+1, 13) *Cumulative Sums test* computes the cumulative sum of the adjusted sequence mapped from $0, 1$ to $-1, 1$ and examines the maximal excursion from 0 of the random walk of the cumulative sums , 14) *Random excursions test* computes the same cumulative sum metric but focuses on the number of cycles having exactly K visits in the random walk between a range of (-4, +4), and 15) *Random Excursions variant test* which is similar to the Random Excursions test except within a range of (-9, +9).

C. Student Resources and Learning Methods

In general, all experiments have a detailed instruction manual with (i) sufficient information about the importance and background of the topic(s) being addressed, (ii) a step-by-step guide for the student to follow with examples, and (iii) a clear list of all the deliverables the students are expected to submit. Furthermore. A template Jupyter notebook is also provided for each experiment. We leverage the modular design of Jupyter, these notebooks contain a series of tasks, each with the following resource elements:

1) A text heading to help students to find each task under the table of contents generated by Jupyter Notebook.
2) A text markdown cell explaining the task.
3) Example codes with which students can practice and gain experience.
4) An empty python function for the student to place their code in and encourage students to write their code in a modular fashion.
5) Test bench codes for the students to run and verify their solution themselves. These test benches verify a variety of potential inputs and outputs that may be encountered by the function written by the student.

All these elements are detailed, concise, complete, and easy for the students to understand, helping them to meet the expected outcomes. However, some experiments will have similar tasks in which they can reuse their implementation with minor modifications. Hence, the subsequent repeating tasks have fewer resources and fewer details. For instance, the first activity provides highly detailed examples and explanations on how to connect to the FPGAs, but the subsequent experiment will only have a heading with a brief description of the task. It is expected for the students to use their prior experience and reuse their code to solve the new challenges. This procedure uses a combination of scaffolding learning and experiential learning. Scaffolding is the process of "learning with assistance". Providing students with sufficient resources and guidance in order to enhance their motivational engagement and educational outcomes [32]–[34]. While experiential learning is the process of "learning by doing". Supplying students with the opportunity to work with the code and hardware directly, along with sufficient examples and test scripts can help ensure a prolonged and expanded knowledge foundation [35]–[38]. Thus, integrating both learning methods into the course modules can benefit engineering and computer science education by helping them connect knowledge learned through the lectures and guides (i.e., scaffolds) to real-world example challenges through hands-on experiences.

III. Implementation Details

In this section, we describe the ways in which Python is used in the PHS course to bridge the gaps between hardware security concepts and the associated fields applied within them, including hardware peripheral communication, modeling of PUFs, and evaluation of TRNGs.

A. Communication with Hardware Peripherals

Several experiments in this course require the gathering of data from hardware peripherals (e.g., FPGAs) for offline analysis. The Python library *Pyserial* provides a convenient wrapper around a serial port on a PC that implements the universal asynchronous receive and transmit (UART) protocol for serial communication. It implements a buffer interface so that data can be stored in real-time from the hardware device, and retrieved from the buffer. For example, on a Windows machine, the interface to a particular serial port connected to a device running at a baudrate of 9600 can be declared in Python using the syntax:

Listing 1: Usage of Pyserial for communicating with hardware.

```
import serial
interface = serial.Serial('COM6', 9600)

# Example of read/write using the interface
interface.write(bytearray.fromhex(0xC0FFEE))
bytesRead = interface.read_all()
```

Usage of Pyserial is important for several of the labs. For the PUF modeling lab, the students are required to program an FPGA with an implementation of an Arbiter PUF (Fig 1) and then obtain challenge-response-pairs (CRPs) from the FPGA. For the TRNG lab, the students have to program the FPGA with two separate TRNG implementations and gather 10 million bits from each and evaluate their quality using a Python implementation of the NIST test suite. In both cases, the Pyserial library enables the communication with these hardware modules.

B. Neural Network PUF Modeling

Before considering the use of an ANN to model the PUF behavior, formalization of the problem and PUF functionality is required. The input to the PUF is an n-bit *challenge* $C \in \{0,1\}^n$. The *delay difference* at the end of the cascaded switches between the upper and lower paths Δ_c determines the *response* r of the PUF as:

$$r = \begin{cases} 1 & if \Delta_c < 0 \\ 0 & otherwise \end{cases} \tag{1}$$

A negative Δ_c means the top signal arrived first, so a 1 is stored in the DFF, while a positive Δ_c means the bottom arrived first, latching a 0. Essentially, the ANN needs to model Δ_c to model the PUF output. Each MUX stage adds some delay $\sigma^{0/1}$, for switching/not switching, and the C controls the switching between top and bottom stages. While C controls the switching, it does not directly relate to the delay properties of the circuit - however, the *parity vector* $\overrightarrow{\Phi}$ of C does. The parity vector is a vector of -1/1 values that indicate when the internal pathing of the PUF flips. The parity vector and Δ_c can be calculated as:

$$\begin{aligned} \overrightarrow{\Phi} &= \prod_{j=0}^{n-1} (1 - 2C[j]) \\ \Delta_c &= \overrightarrow{W} \times \overrightarrow{\Phi} \end{aligned} \tag{2}$$

where $\overrightarrow{W}$ is the *weight vector*, which encapsulates the delay line properties. With the set of CRPs, $\overrightarrow{\Phi}$ can be calculated for every C, and weights in the ANN can approximate $\overrightarrow{W}$, and thus, the PUF responses can be modeled.

For the APUF given to the students, a simple ANN with a single neuron is used for modeling. For ease of implementation and use, the code leverages Tensorflow [39] for training and testing of the ANN. The students are provided a wrapper for the Tensorflow model, which contains functions for training and testing the network on different portions of the data set. Tensorflow can easily leverage a GPU for training and testing of the ANN model, or vectorized instructions in the CPU if they are available.

C. Python Implementation of NIST Test Suite

The original test suite provided by NIST is written in C [19], and provides an interface for running the tests in various configurations. Their implementation has the benefits of efficiency from being in a compiled language, but can be difficult for students to use effectively. Running C programs on a Windows machine (which is common among students) requires setting up and using a Windows-based compilation and debugging toolkit such as MinGW. This creates unnecessary setup steps that do not help the student understand the assigned problem.

Listing 1 Python code for frequency within block test in the NIST test suite.

```python
import numpy as np
import scipy.special as ss

def frequency_within_block_test(file, M=128):
    # load raw bytes
    rbytes = np.fromfile(file, dtype=np.uint8)

    # unpack bits into Numpy array
    bits = np.unpackbits(rbytes)
    nBlocks = bits.size // M

    # reshape binary into blocks of size M
    # compute frequency of bits in all blocks
    blocks = bits.reshape(nBlocks,M)
    ps = np.sum(blocks, axis=1) / M

    # compute test statistic
    chisq = np.sum(4 * M * ((ps - 0.5) ** 2))
    p = ss.gammaincc((nBlocks/2), chisq/2)
    success = (p >= 0.01)

    return [p, success]
```

Python can be an alternative to overcome these issues. However, Python is an *interpreted* language, and thus is not nearly as fast as a compiled language like C/C++. Since the NIST tests are very computationally intensive, implementing them in Python can be difficult. To implement the tests *efficiently*, the popular Python libraries *Numpy* and *Scipy* will do the heavy lifting for data processing. Numpy and Scipy are both highly optimized data analysis libraries with backend implementations written in low-level languages like C and Fortran. Numpy has many built-in functions that are very useful for implementing the tests.

First, the input bitstream $\mathcal{B}$ can be loaded as a `uint8` Numpy array. Many of the NIST tests involve splitting $\mathcal{B}$ into blocks and computing some metrics over those blocks. Numpy's functions trivialize this by enabling the reshaping and restructuring of the underlying data, and computing metrics along a given *axis* of that data. For example, consider the simplified code snippet in Listing 1 for the frequency within blocks test.

The *fromfile* function is used to read a raw byte file into an array with the given data type - *np.uint8* is used so that the data can be manipulated more easily. The default block size M for this test is 128, so the data is reshaped so that each block has 128 bits, then the *sum* function is used to compute the sum along that axis of the data. The variable *props* stores the sum of all set bits in every block in a $(nBlocks \times 1)$ array. Numpy also supports performing arithmetic and bitwise

Listing 2 Berlekamp-Massey [40] algorithm to find shortest LFSR of subsequence, vectorized using Numpy.

```
def vectorized_berlekamp_massey(blocks):
    n = len(blocks)

    # allocate polynomial coefficients
    Dc = Db = blocks
    L = np.zeros(n, dtype=np.uint16)
    M = 500

    # evaluate each bit of the LFSR
    for i in range(M):
        # compute discrepancy
        d = (Dc & 1).astype(bool)
        Dc = Dc >> 1
        mask = L <= i / 2
        T = Dc[d & mask]

        # update the appropriate coefficients
        Dc[d] ^= (~Db[d] & (2**(M-i + 1) - 1))
        L[d & mask] = i + 1 - L[d & mask]
        Db[d & mask] = T

    return L
```

operations to every element in the array at once, using similar syntax that is used for traditional Python integer variables - as is done to the *props* variable. This is useful for other tests that involve more complicated computations to perform along blocks of the data, like the binary matrix rank test and the linear complexity test.

In particular, the linear complexity test is the most computationally intensive test in the NIST test suite. It computes the smallest LFSR that can generate a M-length subsequence of $\mathcal{B}$, with $M = 500$ by default. This is done by using the Berlekamp-Massey algorithm [40], which takes a total of M iterations to compute the linear complexity of a binary sequence. By leveraging the built-in bitwise operations and other high-performance Numpy array slicing techniques, the Berlekamp-Massey algorithm can be vectorized in a highly efficient manner as shown in Listing 2.

Essentially, $\mathcal{B}$ is reshaped into an $(N \times M)$ array, and the linear complexity is computed on every block in parallel. The Berlekamp-Massey algorithm is computationally intensive - involving a loop with M iterations, with multiple if statements. Since the algorithm is vectorized, the if statements can be eliminated by performing bitwise operations on all elements at once, and using masking to modify only certain blocks in the array when appropriate.

IV. Technical Outcomes

When developing this course, we aimed to make the focus on learning about hardware security, rather than learning to program in Python. Python was chosen in part because students already have taken an introductory programming course in Python, because there are many data processing libraries readily available, it is cross-platform, and relatively easy to use. Hence, one important metric is the "usability" of the provided code in each module; whether or not the provided libraries are easy or intuitive to use, and whether or not the experiments can be run in a reasonable time on students' laptops. The latter is especially important; buggy or slow code can be frustrating, and when initially learning these topics, students will likely make errors and need to re-run the analysis. Besides summative assessment results, we also looked at changes in pre- and post-survey responses from students regarding the perceived difficulty of the labs, their familiarity with Python, and familiarity with the relevant tools/platforms and hardware security concepts.

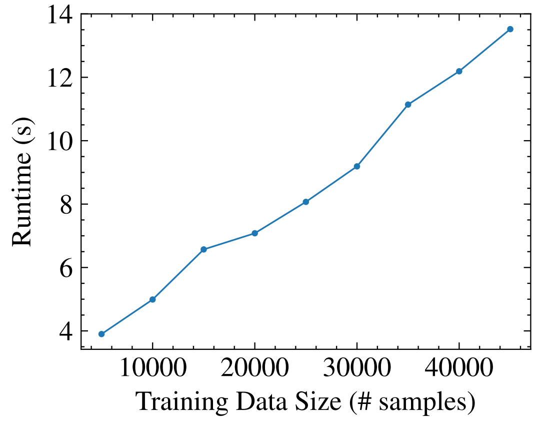

Fig. 4: Runtime of training an ANN with different amounts of the training data.

A. Runtime Results

To evaluate the runtime performance of the presented lab components, we ran tests on a laptop with an Intel i7-6600U processor and 16GB of RAM, which is comparable to the specifications of a laptop that a student may have.

1) Neural Network Training: The machine that the lab software was tested on does not have a GPU. The i7 CPU does however support AVX instructions, so, training does not take a considerable amount of time. The APUF given to the student has 16 stages of MUXes, and thus a 16-bit challenge vector. The ANN used to model the APUF has an input layer with a single neuron, and an output layer with a hyperbolic tangent activation function. The model is trained for 100 epochs using a batch size of 500. The students are given a file containing 50,000 challenges that they send to the FPGA via a serial connection, and then gather responses from the PUF. They are tasked to train the ANN on subsets of the CRPs to see how many CRPs are needed to get a decent accuracy. Runtimes for training the ANN on the i7 machine with larger and larger data subsets is shown in Figure 4.

2) NIST Test Suite: There have been a number of improvements made to the original NIST test suite, so, to give our implementation a fair comparison, we compare with a highly optimized C-based version of the test suite [1]. As shown in Figure 5, our Python implementation is able to outperform

[1]Compared NIST STS GitHub repository https://github.com/arcetri/sts.

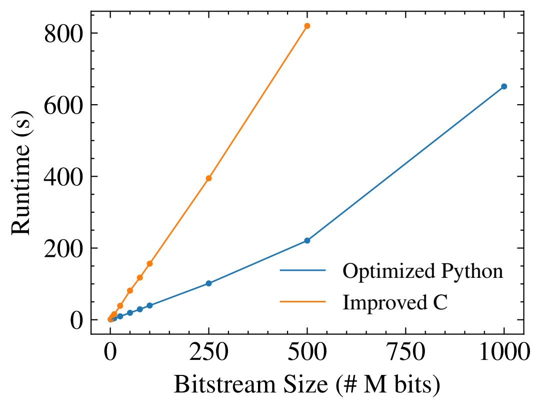

Fig. 5: Runtime comparison of our Python implementation and an improved C-based implementation of the NIST statistical test suite for different bitstreams.

the improved C implementation of the tests [2]. Moreover, the C implementation was not able to complete the tests for 1 billion bits of data, while the Python implementation was.

The NIST code that was provided to the students also includes various input handling and pre-processing code to make the interface easier to use. The interface for the C-based tests is not as straightforward and may be difficult for a student to use. The Python implementation is highly optimized, well suited to systems with lower-end processors and limited RAM availability, and easy to work with in the Jupyter Notebook interface that the class is deployed in. In the TRNG lab, the students are given two TRNG designs to program their FPGA. They are required to gather 10 million bits of data and evaluate the quality of the TRNGs using the provided NIST test suite.

B. *Survey Results*

Surveys were conducted at the start and end of the semester. A total of 37 students completed both the pre- and post-survey. Among other questions (e.g. demographics), students were asked to rate their knowledge of, or familiarity with certain topics, which we called the "*Hardware Skills Inventory*". This inventory included the following topics: mixed signal/digital oscilloscopes, logic analyzers, EM / current probes, multimeters, function generators, digital power supplies, Matlab, C/C++, Java/C#, Python, machine learning / AI, GPGPU, cryptography (theory), cryptography (implementation), Arduino/Raspberry PI or other single board computers, FPGAs, Verilog/VHDL, Synopsys or Cadence EDA tools, SPICE / circuit simulation, soldering / hardware prototyping, printed circuit board design, 3D modeling / 3D printing, and Git or other version control systems. Students are asked to rate their skills from 1 to 5, with 1 being no knowledge of or experience in the topic, and 5 being an expert on the topic. Only a fraction of these topics are covered in the class.

We hypothesized that students in the class would rate their knowledge of or familiarity with those topics we covered

[2]GitHub repository for our optimized Python NIST STS https://github.com/BrooksOlney/TRNG-Test-Suite.

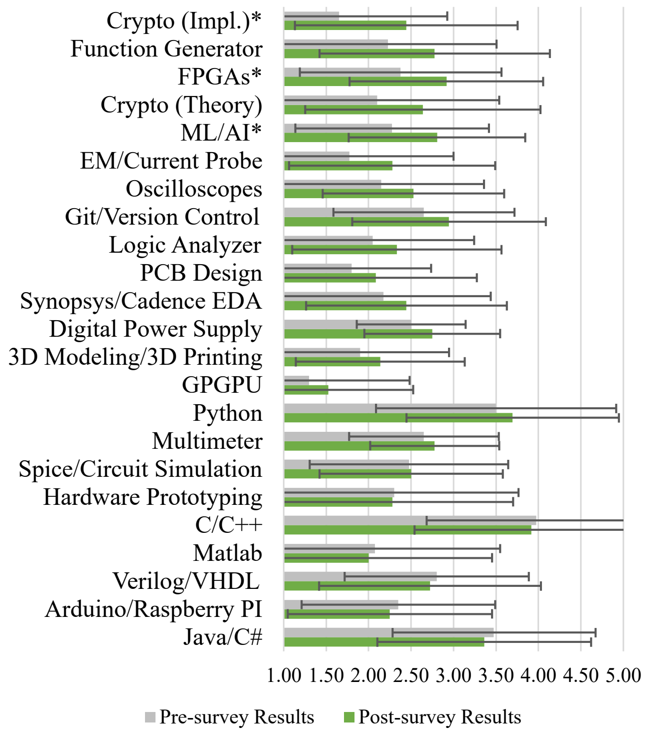

Fig. 6: Comparison showing the student's average self assessment of each topic in the Hardware Skills Inventory. Topics marked with an asterisk indicate a statistically significant improvement ($p < 0.05$).

as being significantly higher by the end of the semester. These topics include cryptography (theory), cryptography (implementation), FPGAs, and ML/AI. Figure 6 shows the baseline results from the pre-survey compared to the post-survey results at the end of the semester. We observed a statistically significant increase (T-test, $p < 0.05$) for cryptography (implementation), FPGAs, and ML/AI. While there were many topics in which students reported an increase in their knowledge, we could neither attribute these increases to the course, as they were not covered, nor were the increases significant given our sample size. While there was a slight increase for Python, it was not significant. Students came into the class feeling confident in their Python skills. In contrast, the main topics - cryptography (implementation), FPGAs, and ML/AI, increased significantly. This survey helps to confirm efficacy of the course modules, and importantly, serves as a baseline for future experiments in pedagogy which are planned in future semesters.

V. Conclusion

In this paper, we have presented three examples of how Python and Jupyter notebook can be leveraged for an introductory hands-on hardware security course. We have focused our effort on ensuring proper scaffolding throughout the course using Jupyter notebooks, as well as the usability of our custom libraries through efficient implementation and ease of use.

References

[1] S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities," *IEEE Access*, vol. 5, p. 26521–26544, 2017.

[2] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang *et al.*, "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[3] Z. Lv, L. Qiao, A. Kumar Singh, and Q. Wang, "AI-Empowered IoT Security for Smart Cities," *ACM Trans. Internet Technol.*, vol. 21, no. 4, jul 2021.

[4] I. H. Sarker, A. I. Khan, Y. B. Abushark, and F. Alsolami, "Internet of things (iot) security intelligence: a comprehensive overview, machine learning solutions and research directions," *Mobile Networks and Applications*, pp. 1–17, 2022.

[5] A. Abdullah, R. Hamad, M. Abdulrahman, H. Moala, and S. Elkhediri, "CyberSecurity: a review of internet of things (IoT) security issues, challenges and techniques," in *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*. IEEE, 2019, pp. 1–6.

[6] M. b. Mohamad Noor and W. H. Hassan, "Current research on Internet of Things (IoT) security: A survey," *Computer Networks*, vol. 148, pp. 283–294, 2019.

[7] T. Xu, J. B. Wendt, and M. Potkonjak, "Security of IoT systems: Design challenges and opportunities," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2014, p. 417–423.

[8] O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and Security in Internet of Things and Wearable Devices," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 1, no. 2, pp. 99–109, 2015.

[9] S. Sidhu, B. J. Mohd, and T. Hayajneh, "Hardware Security in IoT Devices with Emphasis on Hardware Trojans," *Journal of Sensor and Actuator Networks*, vol. 8, no. 3, 2019.

[10] D. D. Ramlowat and B. K. Pattanayak, "Exploring the Internet of Things (IoT) in Education: A Review," in *Information Systems Design and Intelligent Applications*, S. C. Satapathy, V. Bhateja, R. Somanah, X.-S. Yang, and R. Senkerik, Eds. Singapore: Springer, 2019, pp. 245–255.

[11] M. Al-Emran, S. I. Malik, and M. N. Al-Kabi, *A Survey of Internet of Things (IoT) in Education: Opportunities and Challenges*. Cham: Springer, 2020, pp. 197–209.

[12] H.-C. Ling, K.-L. Hsiao, and W.-C. Hsu, "Can students' computer programming learning motivation and effectiveness be enhanced by learning python language? a multi-group analysis," *Frontiers in Psychology*, vol. 11, p. 600814, 2021.

[13] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[14] L. Bossuet, X. T. Ngo, Z. Cherif, and V. Fischer, "A puf based on a transient effect ring oscillator and insensitive to locking phenomenon," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 30–36, 2013.

[15] U. Rührmair, J. Sölter, F. Sehnke, X. Xu, A. Mahmoud *et al.*, "Puf modeling attacks on simulated and silicon data," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 11, pp. 1876–1891, 2013.

[16] U. Ruhrmair and J. Solter, "Puf modeling attacks: An introduction and overview," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

[17] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama, "Implementation of double arbiter PUF and its performance evaluation on FPGA," in *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 6–7.

[18] R. Govindaraj, S. Ghosh, and S. Katkoori, "Design, analysis and application of embedded resistive RAM based strong arbiter PUF," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2018.

[19] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid *et al.*, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," *National Institute of Standards & Technology*, 2010.

[20] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas *et al.*, "Modeling attacks on physical unclonable functions," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 237–249.

[21] G. Hospodar, R. Maes, and I. Verbauwhede, "Machine learning attacks on 65nm arbiter pufs: Accurate modeling poses strict bounds on usability," in *2012 IEEE international workshop on Information forensics and security (WIFS)*. IEEE, 2012, pp. 37–42.

[22] D. P. Sahoo, P. H. Nguyen, D. Mukhopadhyay, and R. S. Chakraborty, "A case of lightweight puf constructions: Cryptanalysis and machine learning attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1334–1343, 2015.

[23] J. Tobisch and G. T. Becker, "On the scaling of machine learning attacks on PUFs with application to noise bifurcation," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer, 2015, pp. 17–31.

[24] A. Vijayakumar, V. C. Patil, C. B. Prado, and S. Kundu, "Machine learning resistant strong PUF: Possible or a pipe dream?" in *2016 IEEE international symposium on hardware oriented security and trust (HOST)*. IEEE, 2016, pp. 19–24.

[25] A. Vijayakumar and S. Kundu, "A novel modeling attack resistant PUF design based on non-linear voltage transfer characteristics," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 653–658.

[26] N. Bochard, F. Bernard, and V. Fischer, "Observing the randomness in RO-based TRNG," in *2009 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2009, pp. 237–242.

[27] V. Fischer, F. Bernard, N. Bochard, and M. Varchola, "Enhancing security of ring oscillator-based trng implemented in fpga," in *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 245–250.

[28] J.-L. Danger, S. Guilley, and P. Hoogvorst, "High speed true random number generator based on open loop structures in fpgas," *Microelectronics journal*, vol. 40, no. 11, pp. 1650–1656, 2009.

[29] S. Srinivasan, S. Mathew, R. Ramanarayanan, F. Sheikh, M. Anders *et al.*, "2.4 ghz 7mw all-digital pvt-variation tolerant true random number generator in 45nm cmos," in *2010 Symposium on VLSI Circuits*. IEEE, 2010, pp. 203–204.

[30] K. Yang, D. Blaauw, and D. Sylvester, "An all-digital edge racing true random number generator robust against pvt variations," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1022–1031, 2016.

[31] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[32] L. Zheng, Y. Zhen, J. Niu, and L. Zhong, "An exploratory study on fade-in versus fade-out scaffolding for novice programmers in online collaborative programming settings," *Journal of Computing in Higher Education*, pp. 1–28, 2022.

[33] E. Acosta-Gonzaga and A. Ramirez-Arellano, "Scaffolding Matters? Investigating Its Role in Motivation, Engagement and Learning Achievements in Higher Education," *Sustainability*, vol. 14, no. 20, 2022.

[34] B. R. Maxim and G. Luera, "WIP: Teaching a Knowledge Engineering Course Using Active Learning, Gamification, and Scaffolding," in *2020 ASEE Virtual Annual Conference Content Access*, 2020.

[35] F. Mantwill and V. Multhauf, *Application of Agile Experiential Learning Based on Reverse Engineering as Support in Product Development*. Cham: Springer International Publishing, 2022, pp. 65–81.

[36] P. Desai, A. Bhandiwad, and A. S. Shettar, "Impact of Experiential Learning on Students' Success in Undergraduate Engineering," in *2018 IEEE 18th Conf. Adv. Learn. Technol. (ICALT)*, 2018, pp. 46–50.

[37] K. Kaneko, Y. Ban, and K. Okamura, "A Study on Effective Instructional Design for IoT Security Education Focusing on Experiential Learning," *International Journal of Learning Technologies and Learning Environments*, vol. 2, no. 1, pp. 1–18, 2019.

[38] D. A. Kolb, *Experiential Learning: Experience as the Source of Learning and Development*. Pearson Education, 2014.

[39] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[40] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.