2024 IEEE/SICE International Symposium on System Integration (SII)
January 8-11, 2024. Ha Long, Vietnam

Encrypted Sensor and Actuator Interface for Encrypted Control Signals via Embedded FPGA Key Generation

Shane Kosieradzki[1], Saahas Yechuri[1], and Jun Ueda[1]

***Abstract*— This paper presents an investigation into the improvement of security and operation time in homomorphically encrypted systems using Field Programmable Gate Array (FPGA) technology. The primary objective is to generate keys efficiently, minimizing key sizes while maintaining security. By leveraging FPGA capabilities for key generation and key switching, smaller ciphertext sizes can be achieved, ultimately improving operation time. The paper focuses on the development of a sensor data encryption system implemented on an FPGA board. The proposed approach enables simultaneous key generation and encryption of incoming sensor data using generated keys. The developed system implemented fixed-size random number generation and prime number checking in hardware, subsequently expanding these capabilities to produce arbitrarily sized prime numbers.**

Index terms: Homomorphic Encryption, Field Programmable Gate Array, Security Parameters, Encrypted Control, Cyber-physical systems

I. Introduction

Cyber-physical systems have become an increasing target of cyber attack in recent years [1]–[6]. Recent cryptographic developments in homomorphic encryption has made the realization of real-time encrypted control systems possible by applying homomorphic encryption methods [7]. By encrypting the control and sensor signals a control system can be designed in which the signal information is protected from attacks such as falsification and sniffing. While the use of a long key is in general preferred to improve the cybersecurity of encrypted control systems, the increased computational overhead due to homomorphic encryption would degrade the real-time control performance. To resolve this trade-off, a concept of key switching has been studied in which relatively low-length keys are generated and switched at a certain frequency fast enough before a single key is theoretically identified by the adversary via a brute-force attack [8]. Since the computational burden to continuously generate keys is high, the use of a dedicated high-performance circuit, such as Field Programmable Gate Arrays (FPGAs), is favorable rather than a fully software-based (e.g., CPU) approach. This paper presents an FPGA hardware architecture that can periodically generate new keys for the class of homomorphic cyphers over integers with hardness derived from the *Approximate-Greatest Common Divisor* (AGCD) and preform encrypt and decrypt operations with the current key.

[1]G.W.W. School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332-0405. Email: {saahas, skosieradzki3, jun.ueda}@gatech.edu.

This work was supported in part by the National Science Foundation under Grant No. 2112793. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Similar investigations have been preformed into Learning With Error secured cyphers [9], however the authors believe integer based cyphers are simpler to both understand and implement, thus making them a suitable choice for wide spread adoption. Such implementation would enable not only fast key generation, but also signal encryption at the site, resulting in a sensor node with improved security.

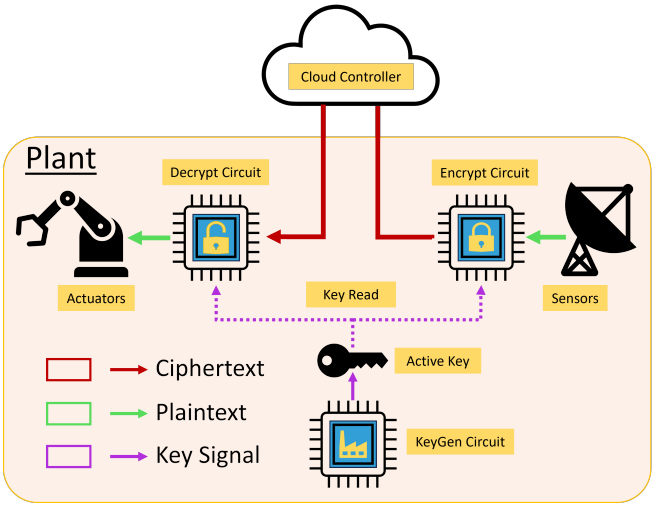

Fig. 1: Autonomous system employing dedicated cryptographic circuits with periodic key re-generation. By embedding a `Enc` circuit into the sensor node, the system can conceal the sensor signal from the cloud. The embedded `Dec` circuit in the actuator node then retrieves the control signal calculated by the cloud. The `KeyGen` circuit creates a new *active key* periodically replacing the key used by the other circuits.

II. Proposed FPGA transducer node with fast key generation

The system consists of a dedicated encryption circuit `Enc` attached to sensor equipment which immediately encrypts the sensor signal and exposes only the encrypted signal. The augmented sensor module is equipped with a communication channel (e.g. a pin) over which the module recieves the key with which it is to use to preform encryption. The encrypted signal can then be sent to the controller for encrypted calculation of the control signal. A dedicated decryption circuit `Dec` is embedded into the actuator node with a communication channel in the same way as the sensor node. The decryption circuit then decrypts the control signal, using the key it received externally.

2024 IEEE/SICE International Symposium on System Integration (SII) | 979-8-3503-1207-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/SII58957.2024.10417179

Both the sensor and actuator nodes receive their key from the same *key register*. This register is periodically overwritten by the output of a dedicated key-generation circuit `KeyGen`. Once the register is updated the `Enc`, `Dec` modules will use this updated value as their keys. This design layout can be seen in Fig. 1

By adjusting the period T_{gen} of key generation we can compensate for weaker keys. Some care is needed during the transition from one key to the next, since the signal still in the controller will become stale when the key register is overwritten. Metrics for how often a key should be switched to ensure security of a real-time encrypted control system have been explored by [10].

Homomorphic cyphers over the integers rely on strong primes to produce secure keys [11], [12]. Therefore in order to generate a key a `PrimeGen` circuit must be constructed. Primes are generated by combination of a *Random Number Generator* (RNG) and a *Primality Test*. There are several different techniques to achieve RNG in hardware [13], [14], the presented work uses a collection of Linear Feedback Shift Registers (LFSR) which each contribute a different bit to the output and are independently seeded. The RNG output is constructed by concatenation of sufficient LFSR outputs to achieve the desired bit-length.

Prime generation can be done in one of two ways, either a computationally expensive but deterministic operation can be dispatched to produce a *provable prime*, or a more computationally feasible but probabilistic method can be dispatched to produce a *probable prime* [15]. Deterministic primality tests require significant resources, in the case of prime sieves a list of all integers up to some limit is constructed and composite numbers are subsequently removed [16]. For large primes, such as those suitable for cryptographic purposes, the memory requirement of such methods become infeasible [17]. Because of the difficulties in obtaining a provable prime, we instead use probable primes with a design parameter ε which can be adjusted to control the primality probability tolerance.

The probable primes are generated by a "guess and check" method where random numbers of appropriate bit-length are constructed from the RNG module, which is sent to the Probabilistic Primality Test (PPT) module. The PPT evaluates the *candidate prime* $p_?$ and if it concludes $p_?$ is prime than successive iterations of the PPT are applied to $p_?$ until primality confidence can be brought within tolerance.

III. RNG using Feedback Shift Register

While true randomness is not achievable within deterministic devices such as Field Programmable Gate Arrays (FPGAs), pseudorandom number generators can be implemented utilizing a starting seed to produce random numbers. One such example is the Linear Feedback Shift Register (LFSR).

An LFSR consists of a clocked shift register with feedback from its constituent bits, often referred to as "taps", as depicted in Fig. 1. By applying the exclusive or (XOR) operation between bits within the shift register, a new pseudorandom bit value can be introduced into the register at every clock cycle. This feedback loop is depicted in Fig. 2.

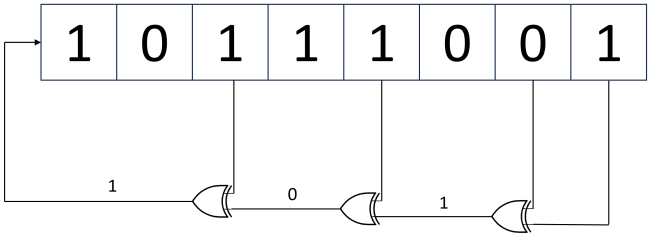

Fig. 2: Linear Feedback Shift Register with taps at its 0th, 1st, 3rd, and 5th bits

LFSRs were chosen as the source of random numbers due to their low resource utilization and ease of implementation in hardware–as they consist of only a few gates and registers. However, they are completely deterministic designs–if an attacker knew the starting seed and the design, they would be able to predict future random number outputs, which would undermine the security of the system. Multiple instances of LFSRs (See Fig. 3) may be employed to make this attack difficult, with only specific bits from each LFSR selected for the output random numbers. Furthermore, nondeterminism could be introduced by artificially introducing race conditions in the insertion of new bits into the LFSR.

Initially, a 128-bit LFSR generating 16-bit random numbers was chosen. For larger random number generation requirements, multiple instances of the LFSR module (with distinct seeds) can be combined to produce arbitrarily sized pseudorandom numbers in hardware. The resulting signal through a LFSR can be seen in Fig. 4.

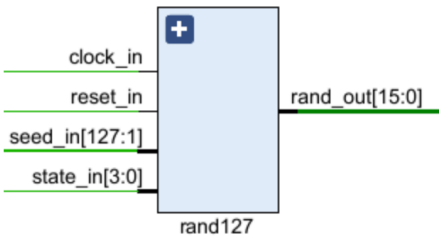

Fig. 3: Linear Feedback Shift Register Inputs and Outputs

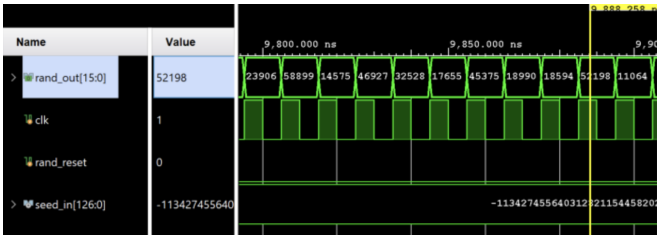

Fig. 4: Simulated Linear Feedback Shift Register random number output signal.

IV. Miller-Rabin Primality Check

The Miller-Rabin primality test is a probabilistic primality test based on Fermat's Little Theorem. It "checks" the

primality of a number n by attempting to prove it to be composite. As opposed to deterministic method such as the Sieve of Eratosthenes, the Miller-Rabin test is much faster with significantly less memory consumption [16]. This is evidenced by their respective runtime complexities– $O(\text{sqrt}(n))$ for the Sieve of Eratosthenes and $O(k * \log^3(n))$ for the Miller-Rabin test, where k is the number of tests run [18].

Algorithm 1 Miller-Rabin Test

```
1: procedure MR(n, s)
2:     for j ← 1 to s do
3:         a ← RandomInteger(1, n − 1)
4:         if Witness(a, n) then
5:             return COMPOSITE
6:         else if
7:             thenreturn PRIME
8:         end if
9:     end for
10: end procedure
```

The `Witness` procedure of Alg. 1 can be described as follows. Let a be a random number which is said to be a witness of n being composite, if

$$a^{n-1} \not\equiv 1 \bmod n$$

These relations are precisely the negation of the equivalence relations of Fermat's Little Theorem, giving strong evidence that n is composite [19]. However, even if n passes the above test, there is a chance that it is a strong pseudoprime, for which the corresponding value of a would be a strong liar. To remedy this, the Miller-Rabin test is performed several times on a potential prime, reducing the chances that it is a strong pseudoprime with so many strong liars [20], [21].

Let ε be the probability that n is a strong pseudoprime after k checks. Then, an upper bound can be placed on the probability that n is a strong pseudoprime as follows [18]:

$$\varepsilon < \left(\frac{1}{4}\right)^k. \tag{1}$$

Therefore, a corresponding lower bound can be placed on the probability that n is prime after k checks

$$P(n \text{ is prime after } k \text{ checks}) \approx 1 - \left(\frac{1}{4}\right)^k. \tag{2}$$

A. State Machine

The FPGA based Miller-Rabin circuit is implemented as a finite state machine (See Fig. 5) with 6 states that systematically checks an input 32-bit prime number up to a specified number of checks (k, determined by setting system parameter ε) and determines whether it is prime or not.

The state machine begins with the transition to the **start** state, which takes in a potential prime n and a number of checks to do k, and resets other intermediate registers. It

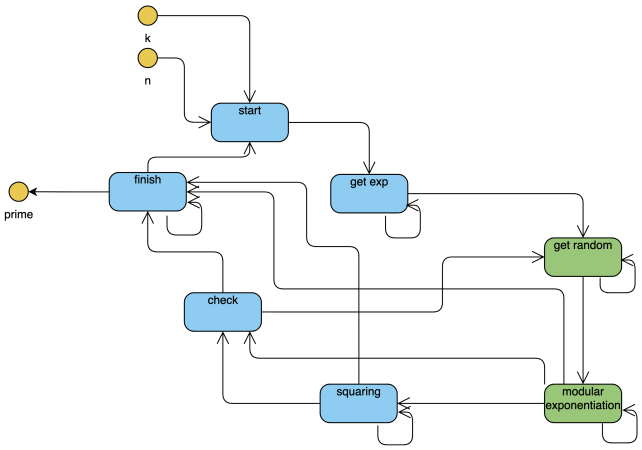

Fig. 5: Miller-Rabin State Machine

then transitions to the **get exp** stage, which prepares the exponent that will go into the miller rabin check. Next, in the **get random** state the random number a is retrieved. The **modular exponentiation** step then follows, finding the value of the base to the prepared exponent, and either continuing, or going to the finish state if n is composite. Next the exponent is continually squared and reduced in the **squaring** step - going to the finish state if n is composite, and only going to the check state if the congruence holds. The **check** state keeps track of the number of loops (new values of a) that have been checked, and if it is under the specified k value, the number of checks is incremented and the loop is restarted. If the number of checks is equal to k, n has gone through enough Miller Rabin checks to be deemed prime, and the transition to the **finish** state is made.

A modular exponentiation module was designed to compute $a^n \mod p$ for arbitrary $a, n, p \in \mathbb{Z}$. The authors identified some similar projects conducted in research labs; however, to our best knowledge, there is a limited number of this type of system integration work reported in the literature.

It repeatedly bit shifts n to the right (divides by 2) while squaring a and taking its modulus ($a_{new} = a_{old}^2 \mod p$) – storing any additional a terms in an intermediate register ($intermediate_{new} = a_{old} \times intermediate_{old} \mod p$ if n is odd). This process is repeated until n is 0 or 1, after which the result is calculated directly from the values of a and $intermediate$. With this modular exponentiation module, 32-bit primes could be tested – but to generate arbitrary size prime numbers, further modification was required to ensure the functionality of all steps of the module for arbitrary input sizes. To allow for this, the module registers and wires were parametrized to a desired prime size (size of n), and the logic for getting random numbers (a in the above equations) for each iteration of Miller Rabin was changed from being fixed (to 16 bits) to building a random number from 16-bit random numbers up to the specified prime size. This logic was kept clocked/sequential – to limit the resource utilization of the miller rabin modules – as the speed increase would not have been worth the extra utilization.

The Miller-Rabin circuit behavior can be seen in Fig. 6

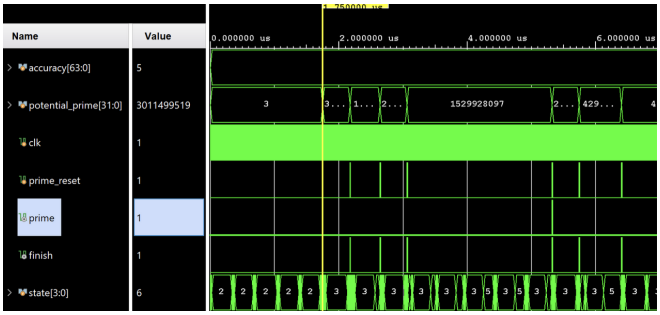

Fig. 6: Simulated Miller Rabin primality outputs

V. Constructing Prime Generation Module From RNG and Miller-Rabin Modules

With these building blocks, higher level modules for the generation of keys could be generated for Dyer's HE1N scheme. The random number generation and miller rabin modules were next combined to create a prime gen module – again parametrized to allow for arbitrary prime size generation. For this module, a potential prime number is built up from 16-bit prime numbers similar to how values for a were generated in the miller rabin module. These potential primes are then fed into the miller rabin module, where the number is checked. This continues until a prime number is found, after which the module holds until it is reset. Tricks can be used to speed up the rate of prime number generation - for example, ensuring that no potential primes are even (which can be easily implemented by ensuring that their LSB is not 0).

VI. Cryptographic Key Generation

Using the prime gen module, a key gen module was then created – which syncs up 3 prime gen modules to generate a key and public modulus (κ, p, and q). This is where the strengths of the FPGA show – as while a CPU based design would have to generate κ, p, and q in series, an FPGA generates them in parallel, and is only limited by the largest prime among them (See Fig. 7).

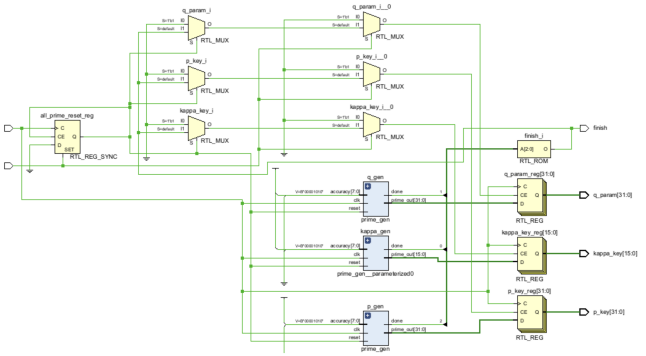

Fig. 7: Key Gen Inputs and Outputs - including 3 Prime Gen modules

VII. Encrypt Module

An encrypt module was created to take keys from the key gen module and generate appropriately sized noise terms for r and s (See Fig. 8). For this module, speed was crucial – much more so than everything involved with the key gen – as new data to encrypt will be coming in each clock cycle, so if the encryption process takes multiple clock cycles, incoming data will be missed, and the analog signal will have more discretization error. To prevent this, this module was designed to be completely combinatorial – so the encryption process for a given plaintext value only takes 1 clock cycle and no data is missed. This was a time/space trade off done by instantiating as many LFSRs were needed based on the desired noise term size rather than building up the noise terms (which would take multiple clock cycles). This entails the random numbers being generated combinatorically using multiple rand modules as opposed to using a single rand module and building random numbers over multiple clock cycles as was done in the prime gen and miller rabin modules. Furthermore, to ensure the appropriate size of the generated noise terms without bit shifting that would take multiple clock cycles, a bitwise and is taken between the keys and rand module outputs to ensure that the noise terms are smaller than the keys.

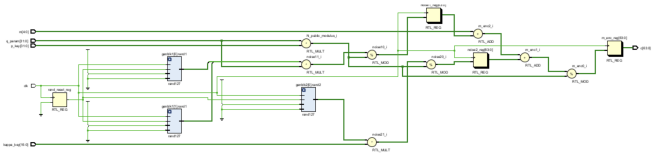

Fig. 8: Encrypt module inputs and outputs – the random number generation modules are instantiated parametrically – using a Verilog generate block

VIII. Final Block Design

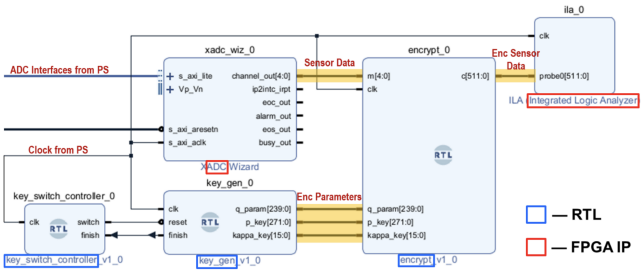

Fig. 9: Final block diagram with all of the top level modules – security parameters are $\lambda = \eta = 32, \nu = 16$

The final experimental setup consisted of an the Zynq-7000 development board with a function generator plugged to its analog input, see Fig. 10. The Internal Logic Analyzer (ILA) IP was used to view outputs connected to internal wires in the FPGA. Keys were continuously generated, and information continuously flowed to the analog input to the encryption module, and was encrypted in real time by it. The

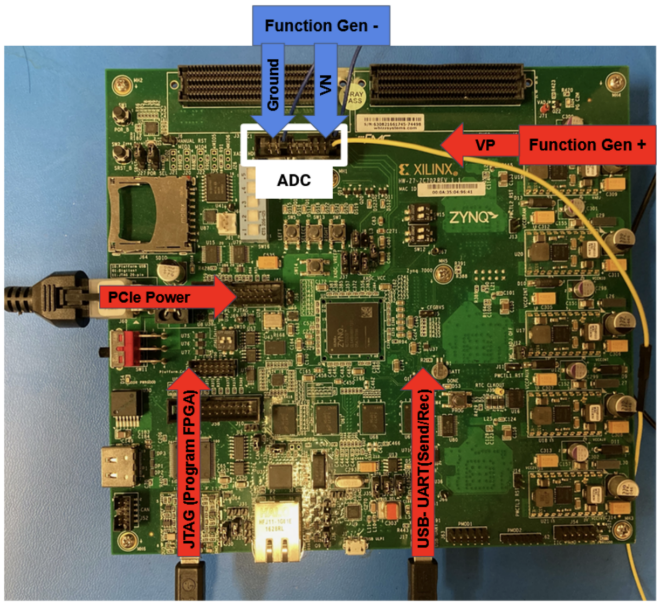

Fig. 10: Experimental setup with function generator emulating sensor data

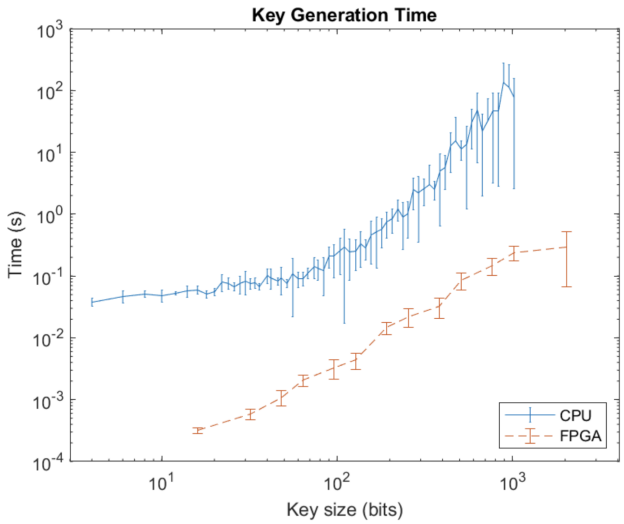

Fig. 11: Key Generation Time on a CPU and FPGA – FPGA performance is approximately 2 orders of magnitude better at minimum

final block design can be seen in Fig. 9. Though a decryption module was not designed yet, using the ILA to view the values of the keys and ciphertext showed that the FPGA was correctly encrypting the input discretized numbers.

IX. Results

Using these modules, an experiment was conducted to compare the prime generation time between a CPU and an FPGA. Keys from 16 bits to 2048 bits were generated on an FPGA and a CPU. Initially, only time to the first prime is recorded, which resulted in a much larger spread of results due to randomness in prime number distribution. To better characterize steady state performance of both implementations, multiple samples were taken for each bit size (5 for CPU, 30-60 for FPGA – CPU tests took very long, so more than 5 samples for each size would have significantly increased experiment time).

The times were then averaged and plotted, with their standard deviations being used for error bars. A log-log scale was used to show the trends of bit size and time over larger magnitudes. These preformance behaviors can be seen in Fig. 11. The FPGA based architecture is shown to have a consistently lower key generation time in all tested key size ranges–with key generation times between 10-100× faster–allowing for much stronger key production for the same amount of time.

X. Conclusion

This paper presented a realization of fast cryptographic key generation on a FPGA board. Both random number generation with feedback shift registers and the Miller-Rabin primality check were implemented. The realized FPGA key generation was confirmed to be 2 orders of magnitude faster than that with a CPU. Future work includes the management of key switching and further performance validation.

References

[1] S. Amin, A. A. Cárdenas, and S. S. Sastry, "Safe and Secure Networked Control Systems under Denial-of-Service Attacks," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, R. Majumdar and P. Tabuada, Eds. Springer, pp. 31–45.

[2] Y. Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, and B. Sinopoli, "Cyber–Physical Security of a Smart Grid Infrastructure," vol. 100, no. 1, pp. 195–209.

[3] Y. Mo and R. M. Murray, "Privacy Preserving Average Consensus," vol. 62, no. 2, pp. 753–765.

[4] F. Pasqualetti, F. Dorfler, and F. Bullo, "Control-Theoretic Methods for Cyberphysical Security: Geometric Principles for Optimal Cross-Layer Resilient Control Systems," vol. 35, no. 1, pp. 110–127.

[5] H. Sandberg, S. Amin, and K. H. Johansson, "Cyberphysical Security in Networked Control Systems: An Introduction to the Issue," vol. 35, no. 1, pp. 20–23.

[6] A. Teixeira, K. C. Sou, H. Sandberg, and K. H. Johansson, "Secure Control Systems: A Quantitative Risk Management Approach," vol. 35, no. 1, pp. 24–45.

[7] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 26–39.

[8] W. Xu, Y. Zhan, Z. Wang, B. Wang, and Y. Ping, "Attack and improvement on a symmetric fully homomorphic encryption scheme," *IEEE Access*, vol. 7, pp. 68 373–68 379, 2019.

[9] S. Behera and J. R. Prathuri, "Design of novel hardware architecture for fully homomorphic encryption algorithms in fpga for real-time data in cloud computing," *IEEE Access*, vol. 10, pp. 131 406–131 418, 2022.

[10] K. Kogiso, "Attack detection and prevention for encrypted control systems by application of switching-key management," in *2018 IEEE Conference on Decision and Control (CDC)*, 2018, pp. 5032–5037.

[11] Y. Chen and P. Q. Nguyen, "Faster algorithms for approximate common divisors: Breaking fully-homomorphic-encryption challenges over the integers," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 502–519.

[12] H. Cohn and N. Heninger, "Approximate common divisors via lattices," vol. 1, no. 1, pp. 271–293. [Online]. Available: https://msp.org/obs/2013/1-1/p14.xhtml

[13] M. D. Gupta and R. K. Chauhan, "Recent development of hardware-based random number generators on fpga for cryptography," in *VLSI, Microwave and Wireless Technologies*, B. Mishra and M. Tiwari, Eds. Singapore: Springer Nature Singapore, 2023, pp. 489–500.
[14] H. B. Meitei and M. Kumar, "Fpga implementation of true random number generator architecture using all digital phase-locked loop," *IETE Journal of Research*, vol. 68, no. 3, pp. 1561–1570, 2022.
[15] J. Rajput and A. Bajpai. Study on Deterministic and Probabilistic Computation of Primality Test. [Online]. Available: https://papers.ssrn.com/abstract=3358737
[16] A. K. Tarafder and T. Chakroborty, "A comparative analysis of general, sieve-of-eratosthenes and rabin-miller approach for prime number generation," in *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, 2019, pp. 1–4.
[17] H. M. Bahig, M. A. G. Hazber, K. Al-Utaibi, D. I. Nassr, and H. M. Bahig, "Efficient sequential and parallel prime sieve algorithms," *Symmetry*, vol. 14, no. 12, 2022. [Online]. Available: https://www.mdpi.com/2073-8994/14/12/2527
[18] E. Bach, "Explicit bounds for primality testing and related problems," vol. 55, no. 191, pp. 355–380. [Online]. Available: https://www.ams.org/mcom/1990-55-191/S0025-5718-1990-1023756-8/
[19] U. Daepp and P. Gorkin, *Fermat's Little Theorem*. New York, NY: Springer New York, 2011, pp. 315–323. [Online]. Available: https://doi.org/10.1007/978-1-4419-9479-0-28
[20] R. Cheung, A. Brown, W. Luk, and P. Cheung, "A scalable hardware architecture for prime number validation," in *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, 2004, pp. 177–184.
[21] J. Sorenson and J. Webster, "Strong pseudoprimes to twelve prime bases," *Mathematics of Computation*, vol. 86, no. 304, pp. 985–1003, jun 2016. [Online]. Available: