Physically Unclonable Functions with Confidential Computing for Enhanced Encryption of EHRs

Aditya B. Sood

Abstract

Continual exploitation of Electronic Health Records (EHRs) has led to increasing amounts of ransomware and identity theft in recent years. Existing cryptosystems protecting these EHRs are weak due to their inherently transparent software that allows adversaries to extract encryption keys with relative ease. I designed a novel cryptosystem that employs Physically Unclonable Functions (PUFs) to securely encrypt user EHRs in a protected SGX enclave. The CPU-attached PUF provides a secret, device-unique value or a 'digital fingerprint' which is used to derive a symmetric key for subsequent AES-NI hardware encryption. Since the cryptographic operations, from key derivation to encryption, transpire in a confidential SGX enclave, the keys are always protected from OS-privileged attacks- a capability lacking in most existing systems. I used my system APIs to evaluate the performance of various hash and encryption schemes across multiple EHR block sizes. SHA512 and AES-NI-256-GCM were selected for cryptosystem implementation because they demonstrated high performance without compromising on security.

Keywords

Cybersecurity — Cryptography — Physically Unclonable Function — Confidential Computing — SGX

Contents

1. Introduction

Accompanying the explosion in popularity of electronic biometric tracking devices, from FitBits to Apple Watches, consumer Electronic Health Records (EHRs) are being harvested at unprecedented rates [1]. Upon being collected, these EHRs leave the user's possession and often fall into the wrong hands or are used for unauthorized purposes. In fact, this private data has quickly become the very essence of black market exchanges, being traded and exploited all without the user's consent or even their knowledge [2]. As the prevalence of smart technologies increases, so does the severity and frequency of these fraudulent exchanges. In the first half of 2019 alone, 4.1 billion records were estimated to have

been breached, indicating the presence of explicit gaps in existing cryptosystems tasked with protecting EHRs [3].

Many modern cryptosystems are highly reliant on the software for purposes like key derivation, encryption, and key protection, which is in itself sub-optimal. Adversaries are able to extract relevant information about cryptographic operations from the innately transparent software with relative ease (such as through software side-channel attacks) [4]. If the encryption key is ever betrayed to the adversary, they will have full, undisputed access to the EHRs, which makes it all the more essential that the key remains securely protected- preferably by the robust hardware. The focus of my research is to develop and test a novel hardware cryptosystem that effectively protects the keys, giving users complete control over their medical data records.

1.1 Encryption

Encryption is the process of encoding information such that only authorized parties possessing the encryption key can decrypt and read that data. There are two primary types of encryption: symmetric and asymmetric. In symmetric encryption, a single private key has the power to both encrypt and decrypt data [5]. The most prevalent symmetric encryption algorithm is Advanced Encryption Standard (AES), which has modes such as Cipher-Block Chaining (CBC) and Galois/Counter Mode (GCM) along with various key sizes.

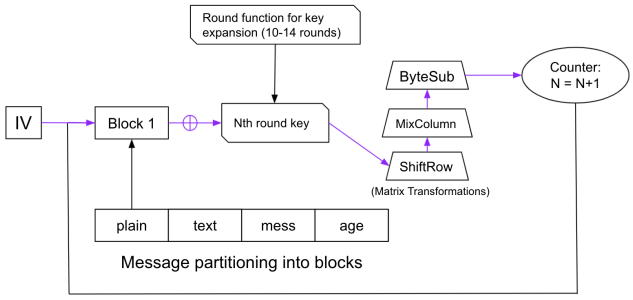

Figure 1. This is a typical AES hardware core, prior to any specifications like GCM or CBC

The other subset of encryption, asymmetric, involves the use of a public and private key pair [6]. The former can only be used to encrypt the data and is thus released to the public. The private key, however, stays only with authorized parties and is what is used to decrypt the data that was encrypted by the public key. RSA is considered an intractable asymmetric encryption algorithm that uses relatively prime numbers to generate this pair of encryption keys [7].

Regardless of which algorithm or subset of encryption is being implemented, what remains essential is always the protection of the encryption key. This involves everything from its generation by a key derivation function (KDF) to encryption to key disposal or storage. With small key sizes, adversaries can easily run exhaustive key searches- the brute force process of testing different key possibilities to find the correct one.

Since the key possibilities vs. key length graph demonstrates an exponential relationship, one of the best defenses against these attacks is to merely increase the key size [8]. For example, a 256-bit key has 2^{256} key combinations, which is estimated to be the total number of atoms in our universe [9]. Hence, an adversary would

have to run through magnitudes of combinations -a process that would elapse multiple centuries- before they found the right key. Other factors that contribute to the key strength include the salt(s) that went into making the key and their availability to external parties.

1.2 Hash

Hash functions are one-way functions that take an arbitrary n-bit input and always output a seemingly random sequence of bits of predetermined length. For example, the digest of the SHA512 algorithm will always consist of 512-bits, irrespective of the input size. Unlike encryption, the hash pre-image property prevents hash functions from traversing backward from the output digest to what was initially supplied in the input [10]. These properties are why many KDFs rely on internal hash-based mechanisms to generate their secure encryption keys [11].

Every secure hash function must also be collision-resistant. A collision is displayed below:

Hash function $H()$

Message space M

Tag space $T = 2^{tag\ length}$

For $m1$ and $m2 \in M$ where $m1 \neq m2$

$H(m1) = H(m2)$

As modeled above, a collision transpires when two unique inputs produce the same hash digest. Since the message space is vastly larger than the tag space, there will invariably be collisions due to the pigeonhole principle [12]. A good hash function ensures that it minimizes these collisions and makes them non-generalizable such that an adversary cannot print or predict a collision at will.

1.3 PUFs

Due to inherently unpredictable silicon fluctuations during the manufacturing process, each Physically Unclonable Function is a random, unique, and immutable 'digital fingerprint.' The PUF can employ its underlying physical characteristics to generate a secret 256-bit value [13].

A PUF could be created, for example, if someone sprinkled reflective flakes on a melting gold brick and let the gold solidify. Shining a flashlight on this resulting contraption would result in a light pattern that has the same properties as a PUF: it cannot be feasibly replicated as it would be nearly impossible to get the exact positioning and angle of each flake in the gold bar, it is random, and it is immutable. I propose that the PUF be embedded on the CPU, where its 256-bit secret contributes towards generating a secure, hardware-confined AES key for subsequent EHR encryption.

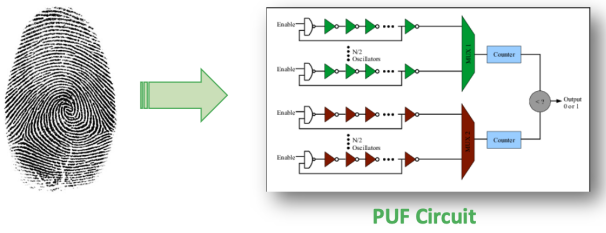

Figure 2. The underlying silicon properties of a PUF make it akin to the device's digital fingerprint.

1.4 SGX

Software Guard Extensions (SGX) is Intel's instruction set for implementing confidential computing on Intel CPU [14]. SGX ensures that upon calibration, the BIOS will set aside a portion of the device's memory for trusted operations that are only accessible to the CPU. The CPU performs access control and encryption on the secure computing enclave to prevent higher privileged software like the OS and BIOS from accessing the contents of this memory. These security measures make SGX a prime location for key storage and secret provisioning that doesn't involve third parties (unlike key escrowing) [15]. However, it is important to note that while it keeps adversaries out in many ways, SGX itself does not protect against side-channel attacks if the code executed within the enclave is not software side-channel resistant. Hence, it is typically advised to only execute small portions of trusted code in each enclave lest everything become compromised from an intrinsic code vulnerability.

2. System Architecture

My hardware encryption system employs PUFs to securely encrypt user EHRs, giving consumers complete control over their data. Upon receiving the user's EHRs from medical tracking devices, the Health application needs to secure these records by encrypting them. Rather than participating in traditional software encryption methods that are inadequate for providing high security, I propose that the EHRs are sent to the hardware using my set of APIs. Upon arriving, the records undergo secure PUF-based encryption, and the resulting ciphertext is sent back to the application for local storage. Alternatively, if the application wants to send the EHRs to the cloud, the PUF will be used to negotiate an RSA key with the cloud. After authentication (via CAs), the PUF can securely send the EHR ciphertext to the cloud in this privacy-preserving manner. Regardless of the type of encryption being used, the entire life cycle and implementation of the key occurs not only within the hardware but in a secure SGX enclave, ensuring the key is always protected.

2.1 Threat Model

The trusted computing base for EHRs consists of Health app, OS, PUF, Software Guard Extensions (SGX), and CPU. For encryption keys, only the CPU, PUF, and SGX are trusted. This cryptosystem possesses valuable security properties lacking in many existing systems including its resilience against side-channels, local malware, and OS-privileged attacks.

2.2 Novel PUF Key Derivation

The first step in the key derivation process is to hash the PUF secret using the SHA-512 hash primitive. The resulting digest, along with the Health app's ID and Salt, is inputted to a KDF to generate a secure AES key. Intel AES-NI then uses the newly minted symmetric encryption key to securely encipher the consumer's EHRs. In addition to maintaining the confidentiality of the records, the cryptosystem can also perform integrity protection

via digital key signing.

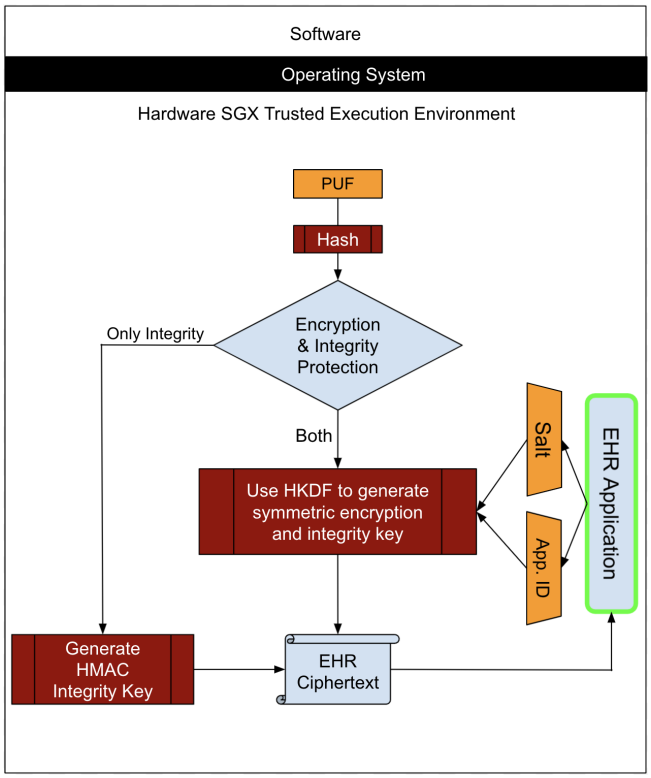

Figure 3. The key derivation transpires in an SGX enclave to protect keys from OS-compromised attacks.

Although the application's seed components are essential to ensuring the uniqueness of each key (using one key for all encryption tasks makes it easier to break the encryption), the key strength is largely provided by the PUF. This is because, as mentioned in my trusted computing base, the Health application cannot be trusted with the key, so we cannot rely on its inputs as a source of the key's security. Since KDFs require the same inputs to produce the same key (which may be needed for purposes like decryption), each PUF must maintain its physical properties -and hence its secret- throughout its lifetime [16].

Because the PUF utilizes its hardware structure to generate the secret, an alteration to the PUF would result in a significant change to the digital fingerprint it provides. Existing protocols for PUF error correction are able to significantly minimize this, resulting in negligible risk posed toward the PUF-based key derivation [17]. As discussed in the system architecture, even if the PUF breaks or the phone is lost (hence losing the encryption keys forever), there is a cloud-based system in place to ensure the records are still accessible by your new device.

Listed below is an overview of the encryption/decryption steps for the EHRs.

1. The key is securely derived in the enclave using a trusted built-in hardware path from the PUF that contains the 256-bit secret
2. The plaintext EHRs are delivered to the enclave via a hardware path from the Health App
3. AES-NI encryption & integrity protection are performed on EHRs using the PUF-derived key
4. EHR ciphertext is sent to the application for local storage

For decryption:

1. EHR ciphertext is sent from the Health app to the enclave
2. The KDF is seeded with the same inputs, resulting in the rederivation of the encryption key that was used to encrypt the records
3. EHRs are decrypted and sent to the Health app through the trusted hardware path, arriving in the clear text.[1]

[1] Hence, the OS and Health app are trusted for EHRs and not keys

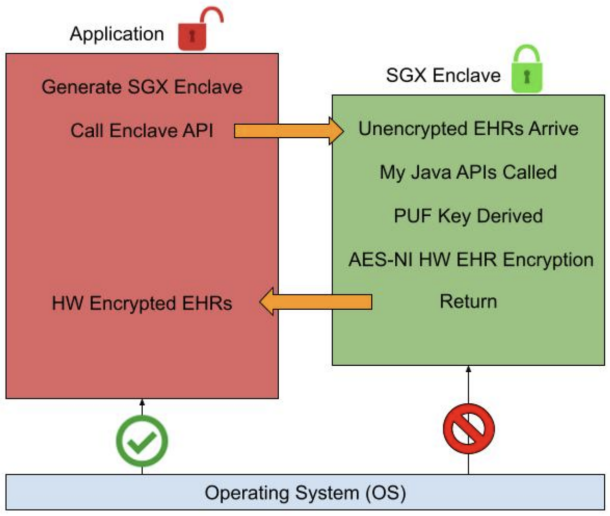

Figure 4. In the event that an adversary manages to corrupt the OS or implant malware on the device, they will find that they cannot breach the key from the software since the key's entire life-cycle is restricted to the CPU-protected SGX enclave.

3. Methods

Due to a commercial unavailability of CPU-embedded PUFs, the secret was replicated by generating a random 256-bit value. Using Oracle JRE and Eclipse IDE, I developed a prototype that followed the same structural format as the system architecture. I also created a set of Java APIs that enable software applications to interact with and use hardware PUFs for secure key derivation, encryption, and hash operations. These APIs are directly executed in a trusted enclave for enhanced security.

To determine which encryption algorithms (and modes if applicable) and hash functions were most computationally optimal for implementation in my design, I tested various cryptographic algorithms on my prototype using Java Cryptography Extensions (JCE) and OpenSSL.

Real-world EHR data sizes from 16 bytes to 8k bytes were inputted for each of these algorithms to test them under real-world conditions, and the throughput was calculated for each trial. In addition to evaluating their performance, the encryption and hash algorithms were also evaluated under a security rubric.

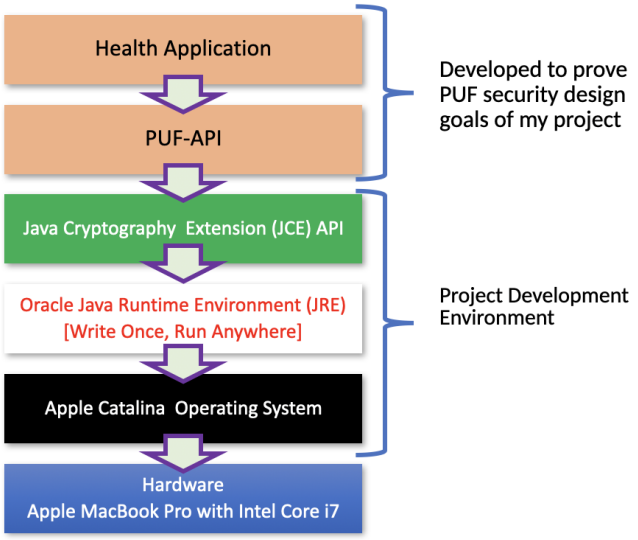

Figure 5. Program software stack

4. Data and Analysis

I conducted three different cryptography comparisons and performed two primary analyses (performance and security) on each comparison.

4.1 Software Encryption Algorithms

In this particular evaluation, I conducted performance and security tests on different software encryption algorithms. This largely included subtypes of the predominant AES scheme with varying modes and key sizes.

AES-GCM had much higher throughput than AES-CBC and ChaCha20-Poly1305 for all data input sizes. Data shows that at the 8k Bytes EHR size, AES-GCM had a throughput about 15 times that of AES-CBC. In addition to its high level of performance, AES-GCM

offers a reliable mode of authenticated encryption using GHASH.

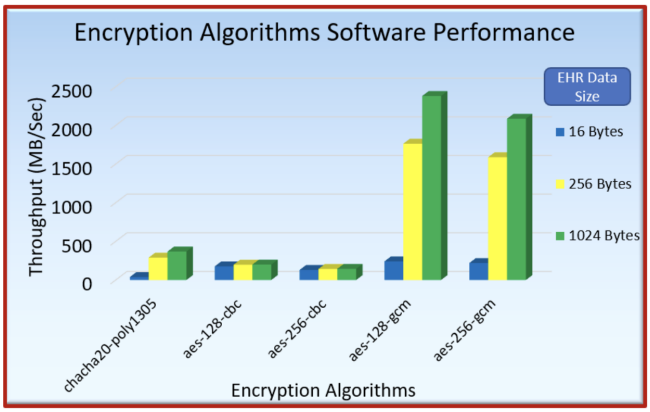

Figure 6. This performance chart for software encryption algorithms indicates the highest performers were both subtypes of AES-GCM.

Authenticated encryption serves a dual purpose by simultaneously encrypting data and confirms its authenticity, removing the need to include a separate key-signing algorithm. If a different encryption algorithm was chosen without authenticated encryption, that would further reduce performance by necessitating a separate form of authentication like HMAC-SHA256. Due to its extreme inefficiency in comparison to other encryption algorithms tested, ChaCha20-Poly1305 was eliminated.

For algorithms such as AES that are semantically secure, it is important to analyze their security against exhaustive key searches: standard dictionary attacks that attempt billions of different keys at a rapid pace to find the correct key. Strong algorithms naturally have a large keyspace, thus exponentially increasing the total possible key combinations. For example, 256-bit key lengths (such as AES-256) have so many possible key combinations that it is approximated to be the total number of atoms in the universe. Due to the additional security it provides and in accordance with the US NIST Post-Quantum Guidelines, a key length of 256 bits was preferred over that of 128 bits [18]. The need for a 256-bit key length, high performance, and authenticated encryption make AES-256-GCM an attractive candidate for design implementation.

4.2 Software vs. Hardware Encryption

To determine the feasibility of hardware encryption, I evaluated the performance and security of hardware (Intel AES-NI) encryption schemes against the control values provided by software encryption.

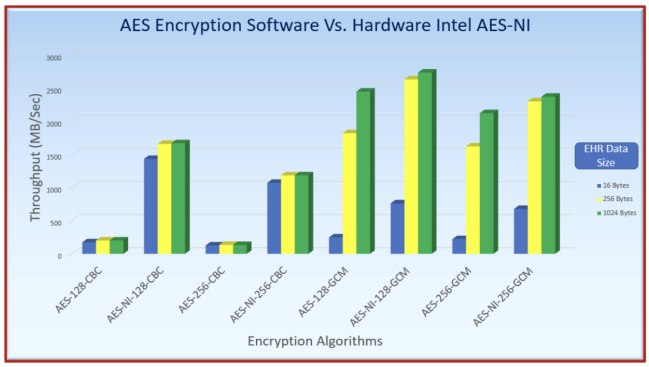

Figure 7. This performance chart depicts the throughput increase provided by hardware encryption.

The throughput tests between AES-NI and AES indicate that hardware encryption is significantly more performant than its software counterpart. As seen in the comparison at the 16 Byte EHR size between AES-NI-128-CBC and AES-128-CBC, the respective encryption rates (MB/sec) are 1431.075 and 172.896. This demonstrates a nearly 700% performance jump with hardware encryption. AES-NI was consistently 5-7X faster for AES-CBC across all key and EHR sizes and 5-8% faster

for AES-GCM.

4.3 Hash Functions

There are no existing hardware hash instruction sets which is why my analysis had to be conducted on a software level for hash functions. Primitives from the SHA2 family were chosen along with other algorithms like GHASH, all of which were evaluated on their throughput and collision resistance for performance and security, respectively.

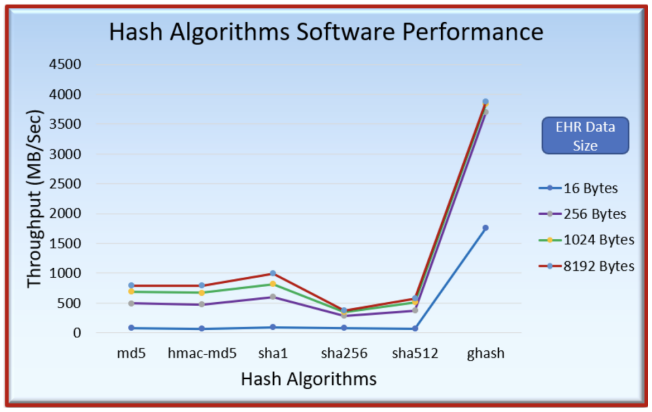

Figure 8. The performance chart for software hash algorithms shows that GHASH outperformed the other hash functions by a significant margin. With the exception of GHASH, the remaining functions had very similar throughputs for each EHR size.

Hash	64B	256B	1024B	8096B
MD5	223.8	492.8	685.3	791.9
HMAC-MD5	205.3	474.4	670.9	794.7
SHA1	279.0	595.3	818.6	998.8
GHASH	2073	3698	3839	3876
SHA-256	172.6	289.9	346.5	373.5
SHA-512	248.3	370.2	506.3	577.3

All collision-resistant hash functions must have:

1. No efficient algorithm, "A," which can print collisions at will (no generalizable collisions).
2. Large tagspace with a preferable size of at least 256 bits to decrease collisions.

MD5 failed both of the criteria for collision resistance, indicating that it severely lacked security and was not feasible for my design. These results are consistent with its current NIST classification that states it is considered deprecated for all practical implementations [19]. Even though it consistently had throughput 2-3 times the other hash algorithms, GHASH was also eliminated due to its relatively small digest size of 128 bits. This was expected since its most prevalent function is in AES-GCM for authentication.

Since the remaining hash algorithms had little variance (nearly negligible) to one another in their rate of hashing for all byte sizes, SHA-512 was selected due to its strong collision resistance (large output size of 512 bits is greater than the other digests).

5. Discussion

My data indicated that for software encryption, the most performant schemes were AES-256-GCM and AES-128-GCM. With regards to the intrinsic security of the algorithm, I used the exhaustive key search metric to determine their relative strengths. With semantic security and a key size of 256 bits, AES-256-GCM is exponentially stronger in defending exhaustive key searches than other schemes (like AES-128-GCM). Due to its stronger defense against brute force attacks on the key, high performance rate, and additional bonus of authenticated encryption, AES-256-GCM was selected for software

encryption.

However, this was only part one of two in the encryption evaluation process. The other analysis sought to evaluate the effectiveness of hardware encryption on my design prototype. Results indicated that incorporating AES-NI in my system accelerated the rate of encryption for all modes and EHR block sizes. In addition, AES-NI directly runs the encryption rounds on the CPU in constant time, defending against various side-channel attacks (namely timing and cache side-channels). As outlined in the threat model, my cryptosystem is designed to protect the key from side-channels, local malware, and OS-privileged root attacks. The latter two are defended against by my specific usage of Intel SGX while the former is addressed by AES-NI. My results indicate that the optimal cryptographic schemes for this design are AES-NI-256-GCM and SHA512.

Societal Impact 1: Users Control their EHRs

Users have an inherent right to be in control of who has access to their personal data, anything from health records to financial data to transaction logs. In an increasingly data-driven world, artificial intelligence and neural networks enable data-hungry companies to not only monetize from user data but also predict users' behavioral traits. This is often done without user consent, and in fact, most users fail to realize how their information is being used to map their traits.

In my cryptosystem, the only way that EHRs can be decrypted is by first accessing the PUF for key derivation and subsequent decryption. The importance of this is seen when contrasted with existing encryption. In these systems, the key can be hacked with relative ease since the attackers has an arsenal of attacks that can compromise the relatively weak software holding the key. This design addresses this security vulnerability by confining the key not only to the hardware but to a trusted execution environment (SGX) in which all the important cryptography processes transpire. By effectively protection cryptographic operations, users can be assured that their data is not being exploited and immorally processed without their consent.

Societal Impact 2: Enhances Security & Performance

- Hardware and silicon-rooted security greatly enhance the security and performance of the overall system. Moreover, hardware security is vastly more resilient against software bootkits, rootkits, malware, and remote hacking.
- Greater availability: silicon vendors (Intel, AMD, ARM) are directly building security into their CPUs and processors/SoCs, rather than relying on software-based security.
- Higher performance and reliability in hardware encryption, as shown by the comparison in throughput between AES-NI and AES.

This design utilizes the benefits of hardware security while addressing the prevalent issue regarding key storage in existing forms of encryption.

Societal Impact 3: Open-Sourced Protocols

By designing the first-ever Java PUF APIs, cryptography interactions could occur between PUFs and

software level applications that were otherwise not possible. Moreover, these APIs blueprinted how PUFs would be used for processes such as key derivation and encryption. Lastly, I developed a hierarchical class prototype of the system architecture, open-sourcing the Java code at https://github.com/soodadityab/PUF-User-Java-API. I plan on improving and diversifying these APIs so they can be reliably used on various platforms to encrypt user data securely.

Since many of the core technologies in my cryptosystem, like SGX and PUF, are still in their early stages of research, a key impact of my project was that it identified certain silicon enhancements that should be made for stronger security.

1. Hardware-protected path from the SGX to the PUF that is accessible by SGX enclaves
2. Hardware-protected path from the IO (display, keyboard, etc) to the SGX enclave, removing the Health app from the EHR trusted computing base
3. The development of SHA-512 New Instructions that permits hardware hashing

References

[1] Mark P Jarrett. Cybersecurity—a serious patient care concern. *Jama*, 318(14):1319–1320, 2017.

[2] Mariya Yao. Your electronic medical records could be worth $1000 to hackers, Apr 2017.

[3] Davey Winder. Data breaches expose 4.1 billion records in first six months of 2019, Aug 2019.

[4] Bonnie Baker. Iot security: hardware vs software, Jan 2020.

[5] CSRC Content Editor. symmetric key algorithm - glossary.

[6] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.

[7] Chris K. Caldwell. Rsa cryptosystem.

[8] Michael J. Wiener. *Exhaustive Key Search*, pages 206–209. Springer US, Boston, MA, 2005.

[9] Jennifer Seberry Professor of Computer Security. World's toughest encryption scheme is 'vulnerable' ... so what about you?, Jul 2021.

[10] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.

[11] Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (hkdf). Technical report, RFC 5869, May, 2010.

[12] *Stanford Cryptography 1*. Coursera.

[13] Yansong Gao, Said F Al-Sarawi, and Derek Abbott. Physical unclonable functions. *Nature Electronics*, 3(2):81–91, 2020.

[14] Intel® software guard extensions.

[15] Somnath Chakrabarti, Brandon Baker, and Mona Vij. Intel sgx enabled key manager service with openstack barbican. *arXiv preprint arXiv:1712.07694*, 2017.

[16] Ihtesham Haider, Michael Höberl, and Bernhard Rinner. Trusted sensors for participatory sensing and iot applications based on physically unclonable functions. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, pages 14–21, 2016.

[17] Meng-Day Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Design & Test of Computers*, 27(1):48–65, 2010.

[18] Information Technology Laboratory Computer Security Division. Minimum acceptability requirements - post-quantum cryptography: Csrc, Jun 2021.

[19] Loganaden Velvindron and Kathleen Moriarty. Deprecating md5 and sha1 in tls 1.2, May 2019.