

Smartphone and Cloud Security: Creating a secure cloud-based continuous implicit authentication application for mobile devices

1. Introduction

Smartphones today require users to provide some form of authentication to access their device. Many Android users elect to use some sort of swiping pattern, and many iPhone users use a 4 digit pin code. On modern iPhones, the home button reads the user's fingerprint in order to allow the user access to the device (however, this approach is often supplemented by a 4 digit pin in case the fingerprint reader is temporarily malfunctioning).

Requiring user authentication before empowering the user to interact with the smartphone device is effective at addressing the threat model of an attacker gaining physical access to the device because the phone was stolen or lost. However, if the attacker gains access to a device that is already authenticated, the attacker can launch an impersonation attack which threatens the victim and the other users in his network.

Lee *et al.* have already done some significant work to develop an application that addresses the above situation [1]. Their approach utilizes the sensor data of the smartphone to train a support vector machine (SVM) of the typical physical phone usages. Once the machine has been trained, and an attacker who steals a logged-in device can be locked out since his physical handling of the phone would differ from the rightful owner. Lee's research demonstrates that by using the accelerometer, orientation sensor, and magnetometer the application can have an intruder detection accuracy of 90% [1]. Similar approaches use other sensor inputs including swiping gestures and light sensors to accurately detect malicious users [2, 3].

In order for this application to be used in a secure manner, the overall system design must be architected. It is important that each user can only see their sensor data, which cannot be accessed by anyone else. Additionally, the user must be sure that their data is not being leaked. The only time the sensor data should be in plain text is during the machine learning algorithms. This paper focuses on designing the security framework in order for this application to run on the user's smartphone and transmit all of the data to the machine learning backend in a secure fashion.

In addition to the security design, we have designed a prototype for this application. The prototype includes an Android Application that can collect sensor data and send it to the backend server, which stores the data securely in the backend database.

This paper will begin by covering the threat model in Section 2. Section 3 contains the security features on the smartphone and on the server. Section 4 presents how the system is designed as a whole, including our communications protocol. Section 5 discusses the work done on the prototype to this point.

2. Threat Model

The threat model for this application must address many different attack vectors. It is possible for the attacker to reside on the smartphone, on the network, or on the server. We address each of the major areas where an attack can take place, and specify the threat model for each part of the system.

2.1 Client

The two main types of attacks considered for the smartphone are physical attacks where an attacker attempts to physically manipulate the phone to gain access to its contents, and software attacks where a

malicious piece of software attempts to access sensitive information. For the physical attacks, we assume that the attacker has access to the unlocked phone either through finding the device in an unlocked state or by knowing the user's lock-screen password. We further assume that the attacker does not have any other information including the user's account password for the continuous authentication application. We consider attacks on confidentiality of the user's sensor data and machine learning model. We also consider availability attacks where the user attempts to disable or remove the continuous authentication application from the phone. We further consider hardware attacks where an attacker is able to physically probe the phone's memory to compromise the confidentiality of sensitive data.

For software attacks, we consider malicious software that attempts to breach the confidentiality and integrity of the data stored on the phone by performing memory access requests. We also consider availability attacks where malicious software attempts to delete data stored by the continuous authentication application.

In this threat model, we do not consider side channel attacks. Our model does not trust the operating system, the hypervisor, or the other applications running on the phone. The TCB is comprised of the trusted features of ARM's TrustZone and CryptoCell. This includes the secure monitor, trusted OS, secure firmware, and the hardware processor that TrustZone uses for execution. TrustZone's CryptoCell is trusted to provide secure and accurate encryption [11].

2.2 Server

Since we are running this application in the cloud, there are many possible attack surfaces. Research has been done by the PALMS lab at Princeton to provide an analysis of the security attacks possible in the cloud computing environment, and systemized which architectures can provide support against different types of attacks. The following threat model extends from their research [15].

There are VM level attacks where it could be attacked by the OS of the VM or by applications also running inside of the VM. The OS has complete view of the VM's memory and other applications could share physical memory allowing for the attack to take place. Furthermore, it is possible the same application becomes compromised and leaks sensitive data to third parties. We will consider all of these VM level attacks.

Beneath the VM, there are several other components that could launch attacks. Other VMs on the same machine could launch an attack through analyzing the shared caches or registers. Furthermore, they could share the same machine memory which could be exploited. The hypervisor also could attack the VM as it is responsible for all the resources allocated to each VM. Finally, there are hardware attacks that can be executed by an attacker with physical access to the machine allowing the attacker to execute a probing attack. We will also consider these server level attacks.

Beyond the machine that hosts the VM, attacks can be performed by other machines in the cloud network's subnet, or machines outside of the network. For example, machines in the network could make false requests to the database and machines outside the network could send invalid requests to the public facing API. One example of this kind of attack would be a user sending sensor data and claiming to be a different device. We need to validate the origin of each request and the data that was sent along with the request. We will consider these attacks as well. One attack we are not considering is a malicious cloud manager provisioning the incorrect resources.

From a high level, we are concerned with the confidentiality and integrity of the data at all times. Furthermore, we are concerned with the integrity of some pieces of our application code that must not be tampered with. In order to address all of these attacks, some components must be trusted. An important aspect of our design is utilizing SGX (Software Guard Extension) from Intel [5] on the server, and we must trust the SGX hardware and software resources are provided by SGX. A diagram of the hardware and software resources provided by SGX is provided in Figure 1. More details on SGX are provided in Section 3.3.3.

One of the most important aspects of any secure system is key management. In order for our key management protocol to take place, we must rely upon the security of one master key. For our application, we will trust the key management provided by SGX. Each SGX device has a unique key that is stored in hardware at manufacturing time, which can be used to derive an enclave specific key [5].

2.3 Communications

We consider an active attacker that can both watch all network traffic, and manipulate packets by either reordering them (splicing attack), changing their contents (spoofing attack), or inserting an old packet (replay attack). Therefore, we are concerned with confidentiality of the data in transit and integrity of the data. We will not consider an attacker that is causing a DoS attack by dropping all of our packets causing none of the user data to arrive at the backend server. Furthermore, we will not consider side channel attacks such as timing analysis attacks or fingerprinting attacks.

3. TrustZone and SGX

The security of our application is dependent upon the security features provided by TrustZone and SGX on the smartphone and server respectively. While TrustZone is currently available on mobile devices with ARM architectures, this paper assume SGX will be available on Amazon EC2 instances in the future.

3.1 TrustZone

The client smartphone device makes use of secure hardware architectures in order to provide the appropriate defenses against attackers who attempt to misuse the smartphone or load malicious software onto it. For our system, we will assume the client uses the ARMv8-A instruction set architecture and includes TrustZone with CryptoCell as a part of its processor architecture. Though other secure architectures like Intel's Trusted Execution Technology (TXT) could be used to implement a similar system, we will focus on ARM and TrustZone since they are much more prevalent in the mobile field.

ARM has recently added to its security features by introducing CryptoCell. CryptoCell provides a mechanism that implements a hardware anchor of trust. CryptoCell is comprised of four main components: two encryption modules for symmetric and asymmetric encryption, a module for securely transmitting and receiving data, and a module for providing cryptography-based hardware security mechanisms, as shown in Figure 2 [11]. Contained within this last module is secure boot. By computing the hashes of the secure firmware and software at different stages of the bootup process and comparing them to a golden hash, CryptoCell is able to ensure the runtime environment is both valid and secure. It is able to instantiate a

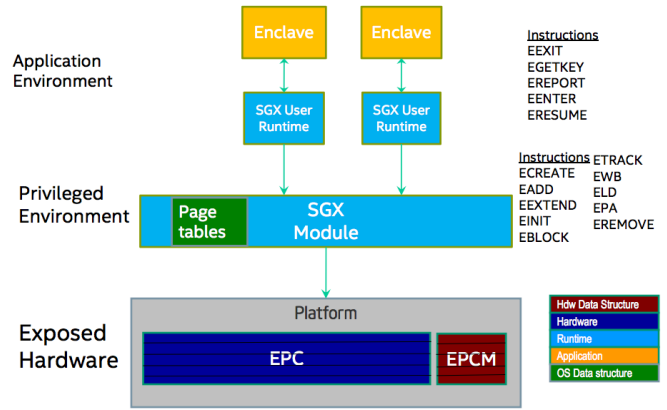


Figure 1: On overview of the hardware and software components available in SGX [5].

trusted execution environment in TrustZone, which our application leverages to maintain the confidentiality of user data. CryptoCell also provides hardware anchors of trust to allow for secure storage. Much like Bastion [17] and other secure architectures, CryptoCell contains a device key and a storage root hash kept in secure memory to allow for secure storage of sensitive data [11].

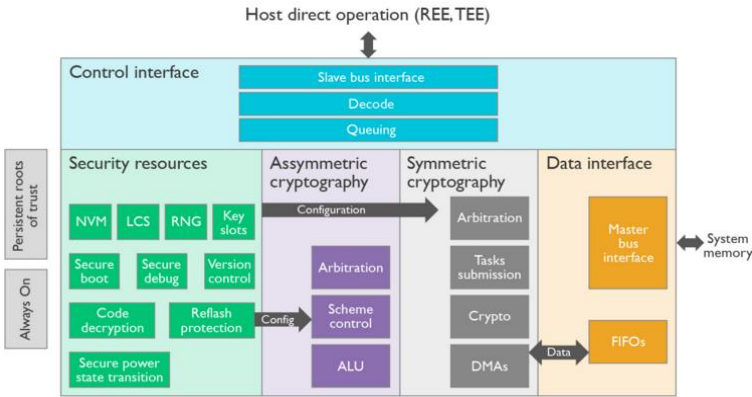


Figure 2: Block diagram of CryptoCell containing critical resources including secure boot, RNG, and key storage. Image taken from [11].

Thanks to secure boot, TrustZone is able to set up a trusted execution environment (TEE) where trusted applications can operate securely. The TrustZone architecture does this by separating the regular rich execution environment from the TEE. Only trusted applications can access the trusted execution environment and secure storage within TrustZone. This is controlled through the different exception levels in the ARMv8-A ISA.

Standard applications run at the lowest exception level. Applications with root-level permissions operate at the next lowest level, and the hypervisor operates at the third level. TrustZone introduces a fourth exception level in which the secure monitor operates. The secure monitor is able to change operating modes between the standard rich environment and TrustZone’s TEE. A diagram of this provided in Appendix C.

Another useful feature of TrustZone and the ARM architecture is the ability to connect use secure peripherals in the computing environment. By interfacing with an ARM AMBA Advanced High-Performance Bus, the TEE is able to securely send and receive signals from I/O peripherals [11]. Having secure peripherals enables the ability to have trusted screen display and keyboard input, which is critical for operations like entering user passwords for authentication purposes.

3.2 SGX

The backend server will execute within the Amazon Web Services (AWS) infrastructure on an EC2 instance (Appendix D). In order to have an execution environment that is trusted, we will assume we can run our backend code on top of an Intel SGX enabled platform. Even though there are no options within the AWS infrastructure to do this currently, SGX is ready to be deployed and we assume it will be available through AWS soon.

SGX is a new set of CPU instructions that allow developers to create secure memory regions for sensitive sections of code and data [5]. This is critical for our application because we have software components that must be trusted. Since we do not trust the other VMs running on the EC2 instance, we need to keep some sections of our code secure. Without SGX, the attack surface would include the entire EC2 stack. Therefore, an attacker could exploit a flaw in the application, guest OS, Xen hypervisor, or the host OS. This attack surface is very large, and SGX reduces the attack surface to only the application and the processor. A summary of some of the new instructions is provided in Appendix E.

The key component to SGX is the ability to create an enclave or a TEE for the application, demonstrated Figure 4. Code running inside of the enclave has access to the plaintext data; however, external access to the data is denied. Furthermore, the enclave supports protection against snooping attacks

of the memory in the enclave. Data and code in the enclave are encrypted in memory and on buses. The data is only in the clear inside of the cores and caches. This security feature is provided by a memory encryption engine (MEE), which runs as an extension to the memory controller. If the memory controller is accessing a memory address inside of MEE region, it forwards the transaction to the MEE to handle cryptographic processing. The MEE is responsible for encrypting the message and generating an authentication tag. SGX uses separate keys for authentication and encryption. Furthermore, the MAC tag uses a counter value in order to prevent replay attacks. There is a counter value for each cache line, which is incremented on each transaction. Lastly, the confidentiality and integrity keys are stored in the MEE and are never stored in memory. These keys are generated at boot and destroyed on restart. By never storing the keys in memory, the system is protected against a cold boot attack. This provides confidentiality and integrity of the data during execution inside of the enclave. It is still the responsibility of the developer to sign and encrypt the data that will be stored outside of the enclave on the database.

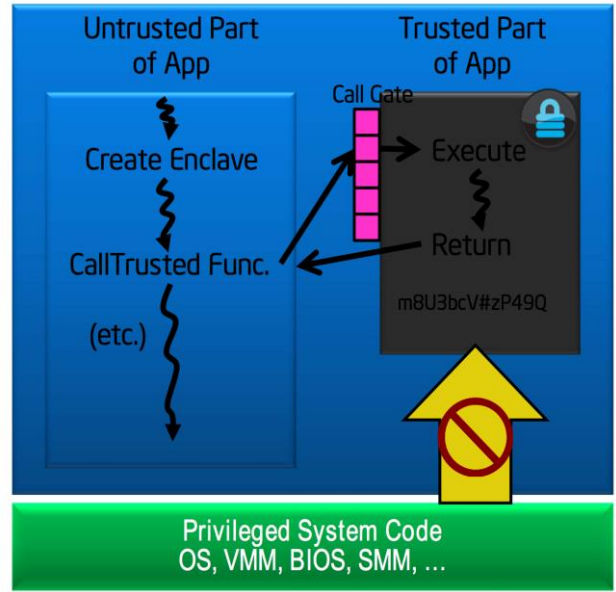


Figure 3: The fundamentals of SGX allows the developer to execute certain sections of code in a trust environment which can only be accessed from specific entry points [5].

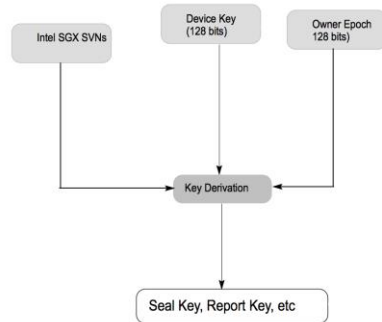


Figure 4: The inputs to the key derivation scheme for the enclave keys [9].

Another important feature of SGX is key management. There is a device key that is at the root of the key hierarchy. This device key is fused into the processor at manufacturing time. In addition to the root key, keys are derived based on Owner Epoch, which is an extra 128 bits of entropy, which the user can specify in order to add some personal entropy to the derivation process. Finally, the derivation process is also dependent upon enclave identity and measurements, and SGX security version numbers (SVNs). SVNs provide insight into the veracity of the current SGX environment. This is demonstrated in Figure 4.

4. System Overview



Figure 5: The overall system design. The security of each step in this process must be ensured.

This section will cover how the system works beginning with enrollment and attestation and ending with the user utilizing the model created by the machine learning algorithms for continuous authentication. This process is provided in Figure 5.

4.1 Software Modules

On the client side all software modules will be run in TrustZone's TEE. This includes the code responsible for sensor data collection, user configuration, machine learning, and crypto libraries. All of these modules must be kept secure because they will interact with the user's sensitive data in plaintext.

Unlike the client side code, not all code on the server will be executed in a trusted environment. Furthermore, we plan to use multiple programming languages for different components on the server since it does not have the same Java restriction enforced by Android app development.

1. **REST API:** This is a NodeJS [19] application that is responsible for receiving the network request. A popular library used for creating API handlers is ExpressJS [20]. Details on REST and our API can be found in appendices A and B.
2. **Encryption and Decryption:** This software component is written in Java and must execute in a secure environment. This software module will be responsible for verifying the integrity of the request and the data sent along with the request. Since this module will have access to the user's plaintext information, it is important this module is executed in a secure environment. There are many libraries available in Java for symmetric and asymmetric encryption. It is important to use Java for this module because Java will be used on the client for encryption and decryption. Matching the encryption and decryption libraries between client and server reduces design complexity
3. **Machine Learning:** This component will be written using Python and must execute in a secure environment because it operates on users' plaintext data. Python has strong machine learning support, particularly through LibSVM and the scikit-learn libraries.
4. **Databases:** We are currently using MongoDB as our database. The databases do not need to be kept secure because they will only store the encrypted information. Furthermore, we will store each database entry with a hash over the entry to protect against integrity attacks.

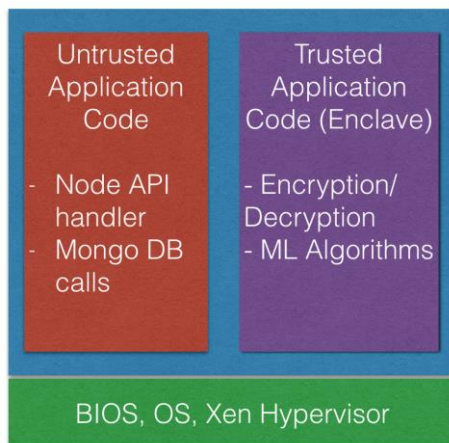


Figure 6: The separation of trusted and untrusted parts of the server code.

In order to create a secure environment for the Encryption/Decryption module and the ML model, we will create an enclave with SGX. Details on creating an enclave with SGX are provided in Appendix F and the division of code on the server is provided in Figure 6. With an enclave created, the critical components of the server application can be protected against attacks from within the VM and from other VM's on the same server [5].

4.2 Android Application Marketplace

One of the worst attacks that could be launched is a phishing attack where a different malicious application pretends to be our application, and the user installs the incorrect application onto their device. Our application needs to run as a device administrator in order to have the privilege of locking the device, so if a malicious application were to be downloaded, the user could be tricked into giving a malicious

application administrator access, which could be very bad for the user. This could lead to the user being locked out incorrectly (false positives), leaking user secrets, or wiping the device. Therefore, it is important for our application to have an identity that is secure, which allows us to mitigate the threat of an impersonation attack.

Android applications are typically signed directly by the user, rather than a CA of the application marketplace [16]. Therefore, the origin of the application is determined by the application signature. The signature will be based on the developer's private signing key. This signing key must be kept secure in order to prevent impersonation attacks. In order to keep the signing key secure, we plan to store the key on a flash drive that is kept safe amongst the application developers. Access to the secret key should be limited to when the application is signed, and the key should never be saved onto a device with network access. The key should only exist on the flash drive. With application signing, our application in the marketplace will be tied to our identity. This will help mitigate phishing attacks, because an attacker cannot impersonate our identity, even if he can create an application that mimics our application.

4.3 Enrollment and Attestation

Upon downloading the application, the first thing that must occur is mutual attestation. It is important for the client to know that the server code has not been tampered with. Otherwise, the user cannot trust the validity of the backend server. Similarly the server must know the application is in the correct state, in order to make sure the sensitive ML model that the server sends to the client will not be leaked. Therefore, upon opening the application for the first time, the application will begin the attestation protocol by making a POST call to /register. The server will respond to this call with a challenge, r , and the client will be responsible for attesting the current version of the application code along with the given challenge.

In order for the client to successfully attest the code on the smartphone device, a hash must be computed of the application code. In addition to the application code being verified, the environment that the application is running in must be verified. This includes proof that the application is running in the secure world of the TrustZone and the boot process was secure. CryptoCell provides secure boot and these measurements can be sent to the server to verify the environment of the application. It is important for CryptoCell to also sign these measurements in order to demonstrate the measurements came from a trusted entity.

In addition to secure boot and verification of the TrustZone secure world environment, it is important for the application code to be verified. It seems that CryptoCell should be able to perform attestation of the application code; however, documentation on CryptoCell never describes how this would be completed. Therefore, we will assume we also have access to a TPM chip inside of Android in order to demonstrate how an attestation protocol would be implemented on the device. The protocol is similar to that described in [18]:

Client → Server: POST /register; it makes an HTTPS call to begin the attestation protocol

Server → Client: r ; the challenge

Client → TPM: PK_C, r ; the client generates a public/private key pair and sends the public key along with the challenge to the TPM. This private key must be stored securely on the Android device. This is accomplished by storing the private key encrypted with the trusted device key provided by CryptoCell. The integrity of the data is protected by the secure root hash.

TPM → Client: $[\text{hash}((\text{hash}(\text{code}) \parallel r), PK_C)]_{TPMSK}, \text{hash}(\text{hash}(\text{code}) \parallel r), PK_C$; The TPM computes a hash of the application code and then takes another hash in order to include the challenge in the attestation report. It is important to include a challenge during the attestation protocol to avoid replay attacks. An interesting

attack that is not easily handled is when an attacker requests the TPM chip to compute the hash of the legitimate code that is stored elsewhere in memory. The TPM chip must be able to verify the caller, and take the hash of the correct code. Otherwise, an attacker can tell the TPM chip to sign the legitimate code that is also sitting on the device inactive, and the attacker could forge the attestation report.

Client → Server: $[\text{hash}((\text{hash}(\text{code}) \parallel r), \text{PK}_C)_{\text{TPMSK}}, \text{hash}(\text{hash}(\text{code}) \parallel r), \text{PK}_C, [\text{DEVID}]_{\text{SKC}}, \text{DEVID}, q]$; The client forwards the attestation report to the server by calling PUT /register. In addition to sending the attestation report, the client sends the server a challenge q , which will be used for server attestation. DEVID is the IMEI, which acts as the device ID.

Attestation Verification: The API request is handled in the Node API layer and forwards the request to the Enclave to perform the attestation verification. In order to verify the attestation report the public key of the TPM must be known from a TPM certificate signed by a CA. Given the TPM public key, the server can verify the integrity of the message and that it was signed by a TPM. Then the server must calculate $\text{hash}(\text{golden_hash} \parallel r)$, where golden_hash is the valid hash of the client side code. The golden hash must be stored securely, on the server. The confidentiality and integrity of the golden hash will be protected within the SGX enclave. As long as the attestation report is valid, the DEVID and PK_C are stored in the user database, and the server begins constructing an attestation report to deliver to the client, based on the challenge q .

Application Enclave → Quote Enclave: SGX directly supports remote attestation through a call to EREPORT. The REPORT provided by SGX is a hardware assertion that contains the enclave's attributes and measurements, which include a hash of the enclave's code. The full reporting structure is provided in Appendix G. This report will contain a value that is specified by the user at the time of the report. We set this value to be q . Similar to the need for TPM in the Android device to perform remote attestation, SGX handles this issue by implementing a quoting enclave (QE). The quoting enclave verifies the report produced by the application enclave, and signs the report using QE's private key. QE's public key will be known, and valid signature by QE indicates the application enclave is running in a secure environment. Given the quote from QE, the application is ready to send the attestation report to the client. This is demonstrated by Figure 7.

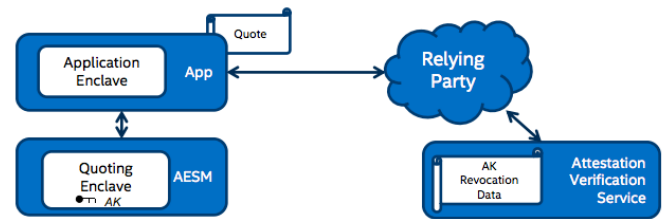


Figure 7: Remote attestation with SGX.

Server → Client: $[\text{hash}(\text{REPORT}), \text{PK}_S]_{\text{QESK}}, \text{hash}(\text{REPORT}), \text{PK}_S$; The server is sending its public key to the client and it's important for the corresponding secret key to never be leaked. In order to handle this, the server's secret key will be stored encrypted with the enclave's secret key. The client will be responsible for validating this attestation report by first setting the REPORTDATA value in the golden report to q , and then must compute $\text{hash}(\text{golden_report})$ and verify the signature by the QE by getting QE's public key from a CA. Once the signature validation is complete, both the client and server have attested one another, and they know each other's public keys, which will be important for future communications.

Once again we have the issue of storing the golden hash securely (this is a different golden hash form before, this golden hash is verifying the server code, the one on the server is verifying the client code). The golden hash could be encrypted with the CryptoCell device key, and kept secure with CryptoCell's secure memory.

4.4 Password and Application Configuration

Once the app is downloaded and the client and server have mutually authenticated one another, the user is prompted to configure their account settings. It is critical for this process to be completely secure and confidential so that an attacker cannot read or modify these settings and prevent the app from behaving correctly. The user first creates a unique, strong password that is associated with their user account. Each user account is tied to the IMEI of the mobile device. We assume only one user is the primary owner of the mobile device at any one time and thus only provide continuous authentication for that user. As described below, we have put mechanisms in place if the primary owner wishes to temporarily share the mobile device with a trusted friend or give primary ownership of the device to another person.

We include multiple security defenses to ensure password the password creation process is secure. First, we take advantage of secure peripherals provided by TrustZone to guarantee no keylogging app can read the password from the keyboard. We also ensure that all user passwords are of a certain strength by specifying minimal lengths and complexities (e.g. the inclusion of special characters) in the password. We also perform a dictionary check to make sure the password is not easily guessable through dictionary attacks. Once entered and confirmed by the user, the hash of the password is calculated and stored in secure memory. A salt is not needed in this case since there is only one password stored on the phone and because salts simply prevent correlating multiple passwords in the same database. Storing the hash of the password is also preferred since the password would not be compromised even if an attacker were able to access and decrypt the secure storage provided by TrustZone and CryptoCell. The application-specific user password is used throughout our app to authenticate the user. During these authentication checks, the user simply enters the password through the trusted keyboard, and the hash of the input value is compared to the reference hash in secure storage.

After creating the application-specific user password, the user is prompted to configure other system settings. The first changeable setting is the training period duration. Though we do propose a training period of one day as shown below, the user has the option to specify longer training periods in the hope of collecting a wider variety of behaviors. This may be desired if the user's phone handling habits differ on different days of the week. The next setting to change is determining what the application does when it loses administrative privileges without the primary user's authorization, which is the first step to uninstalling the application. Since the authentication service can no longer lock out the user from accessing the device after its administrative privileges are revoked, we treat this action as the security boundary for our system. Further discussion of the uninstallation process is discussed below in section 4.9.

4.5 The Training Process

Now that the application has been configured, the client can begin sending data to the server. The first step in this process is sending an update to the current user's mode. Before training begins, the user's mode is set to OFF. This information is stored in the user table in the database. In order to update the current mode of the user the client will make a simple call to PUT /mode with the new mode in the JSON body of the request. In order to securely communicate with the server, we will use a dedicated protocol rather than rely upon the security of HTTPS. We will use HTTPS in addition to this protocol; however, our secure communications is dependent upon the following communications protocol and not HTTPS. This is important because even with HTTPS, the data is in plaintext in the API handler, and then we will have to trust the reverse proxy on the server to not snoop on the user's plaintext when handling the TLS decryption. In order to avoid adding the reverse proxy for HTTPS and the API handlers to the trusted zone of the application, we will implement the following communications protocol:

Client → Server: GET /nonce; Header: DEVID. This returns a nonce to the user that will be used for the next message. It is important for the nonce to be determined by the server in order to make sure each nonce is only used once across all users. The server will keep track of which nonce is sent to which clients by checking the DEVID in the header of the request.

Server → Client: n ; return to the user the next nonce.

Client → Server: PUT /mode. Header: DEVID. Body: $E_{\text{Session Key}}(\text{Mode})$, $E_{\text{SKC}}[h(\text{Mode} || n || \text{DEVID})]$, $E_{\text{PKS}}\{\text{Session Key}\}$; this protocol will provide non-repudiation, integrity, confidentiality, and prevent against replay attacks. We want to make sure an attacker cannot pollute the database by sending this exact message. This is protected by the nonce. We want to make sure only the server can have access to the message, and this is ensured by the symmetric encryption of the message with a session key that is only made available to the server. The session key will be computed using the random number generator that is available in CryptoCell [11]. The message is verified by checking the client's signature. First the server decrypts the session key and the message. Lastly, the server computes a hash of the message, nonce, and DEVID. If this hash is equal to the public key decryption of the signature, then the request is valid. A valid request results in a 200 response, otherwise, the server responds with a 400 response.

Once this message has been sent successfully, the user will have successfully enrolled and its mode will be set to training, and the training process begins. To learn the behavior of a user, the app regularly polls the smartphone's sensors for their reading values. New data is only recorded when the phone is actively in use, as determined by when the screen is on or not. We do not record the data values for when the display is off because we are only concerned with comparing how an attacker would hold the phone when using it, not when storing it. In other words, the phone does not care who is holding it when the screen is off because the data cannot be accessed since the phone is not in use. The application needs to record many data points to develop a strong model for legitimate user behavior. Measurements are taken 10 times per second over the course of a day. Since the average American mobile phone user uses their phones for about 4.7 hours each day [4], it should be easy to collect thousands of data points for each user. Having an extended training period of an entire day also has the added benefit of capturing different user behaviors as the phone is used differently throughout the day. Smaller training periods might only capture a subset of the user's common behaviors and thus cause unnecessary lockouts when these behaviors are encountered in the continuous authentication mode. When the data is collected, it is immediately encrypted and stored in secure storage by CryptoCell and TrustZone. This prevents an attacker from accessing this data as it is recorded. At the end of the training period, the data is sent to the server, where it is stored in the database. We have measured each entry into the database to be around 0.5 kB in size. Thus, data that would need to be uploaded to the cloud would maximally be around 432 MB, but only average around 85 MB. This is a very reasonable size for a file upload, especially if users choose a Wi-Fi connection instead of cellular data to perform the upload.

Once the data is ready to be uploaded to the server, the client will use the same communications protocol as above in order to send the sensor data to the server.

Client → Server: GET /nonce; Header: DEVID.

Server → Client: n

Client → Server: POST /sensor. Header: DEVID. Body: $E_{\text{Session Key}}(\text{Sensor Data})$, $E_{\text{SKC}}[h(\text{Sensor Data} || n || \text{DEVID})]$, $E_{\text{PKS}}\{\text{Session Key}\}$

This request arrives at the API handler on the server. The request must be first delivered to the encryption/decryption software module running in the secure enclave. The encryption/decryption will validate the signature of the sensor data, and be responsible for encrypting the sensor data before saving it to the database. In order to store the sensor data securely on the server, the user's sensor data is re-encrypted with a user specific key. The user specific key is determined by using a key based key diversification function (KBKDF). The master key is the enclave key which is securely derived using SGX key derivation, and the input keying material used to diversify the master key is the user's device ID. A simple KBKDF is HMAC Key diversification, which uses an HMAC to take a keyed hash of input keying material with an optional salt using the master key [10]. Therefore, a user encryption key is defined by $\text{HMAC}_{\text{MASTERKEY}}(\text{DEVID})$, where the master key is the enclave specific key. This key is used to encrypt the each sensor data entry. We plan to use AES-128 for our encryption algorithm. The user authentication key will be $\text{HMAC}(\text{HMAC}(\text{DEVID}))$ where the key to HMAC is the enclave's master key. This key will be used to compute a signature over the plaintext data, which will be stored along with the encrypted data for each database entry. In order to save the data in the database, each database entry will have a unique database id. The database ID will be stored in plaintext in the database, but will be used in the authentication tag. This will prevent an attacker from directly copying data in the database. For instance, if an attacker attempts to copy ten entries from a user and write them back to the database to make those entries appear more frequently in the database, the authentication tags for those entries will fail because the database ID will be different on the new entries, and then the authentication tag fail verification. Once the data is ready to be stored on the server, the code exits the enclave and delivers the authenticated and encrypted data to the database module to make the database request to write the data to the database. The session key, user encryption key, and user authentication key are deleted after this process is complete. By utilizing our secure communications protocol, SGX enclaves, and SGX key management, we are able to provide confidentiality and integrity of the data in transit and on the server.

4.6 Creating the Machine Learning Model

Once collected, the data is used to train a one-class support vector machine (SVM). This SVM essentially analyzes the cluster of the data in many-dimensional space and constructs a boundary. Values located within the boundary exceed a certain probability of belonging to the user and values outside the boundary are likely to be coming from an illegitimate user or a rare outlier for the real user. When creating the SVM model, we can specify whether the training set contains only inliers or if a certain percentage of the data contains outliers. For our purposes, we assume that a small portion of the most extreme or isolated values from the training data represent outliers.

We choose to use one-class SVMs primarily because they only require data from a single user. Using a multi-class SVM could possibly leak information about how multiple users hold their phones because the machine learning model performs a comparison between users to determine which class the test data value fits in better. Furthermore, our system should work with as few as one users, whereas other models may need to create a library of multiple users in order to create a useful classification model. Using one-class SVMs is also preferred to other anomaly detection techniques because they are able to characterize more complex patterns in their model.

The user sends a request to create the SVM model from the data already collected by using the following protocol:

Client → Server: GET /nonce; Header: DEVID

Server → Client: n;

Client → Server: POST /model; Header: DEVID, Body: $E\{E[\text{hash}(\text{POST /model} \parallel n)]_{SK, C}\}_{PK, S}$; This message contains the request to generate the model and the information needed to authenticate the request. Since the nonce is encrypted using the client's private key, it is not forgeable by attackers who wish to impersonate the client. Furthermore, it is not subject to replay attacks due to the nonce.

Once this occurs, the cloud server will start the training of the one-class SVM model. The client may then periodically send request to the server to retrieve the model once it finishes training. The request is sent using the same protocol as used to send POST /model requests.

Client → Server: GET /nonce; Header: DEVID

Server → Client: n;

Client → Server: GET/model; Header: DEVID, Body: $E\{E[\text{hash}(\text{GET /model} \parallel n)]_{SK, C}\}_{PK, S}$

If the model is fully trained, the server responds to this request according to the following:

Server → Client: 200 OK; Body: $E_{\text{Session Key}}(\text{model}), E_{SKS}[\text{h}(\text{model} \parallel n)], E_{PKC}\{\text{Session Key}\}$

Otherwise, the server responds with an acknowledgement saying the data isn't ready via a "204 No Content" response.

4.7 Performing Continuous Implicit Authentication

When the model completes training, it is sent back to the mobile device securely, as described in the communications protocol in section 4.6. There, it is put into secure storage on the mobile device and accessible only to the trusted portion of our application operating in TrustZone's TEE. At this point, the application continuously authenticates whoever is handling the device. The application runs in the background and at regular intervals, polls the mobile device's sensors for their data in a similar fashion to the data collection mode. The polling frequency can be adjustable. With a higher polling frequency comes a higher performance overhead of continuous authentication. We also predict the app may lock out the user more frequently if the polling happens more often but the model maintains the same false positive rate. The sensor readings are input to the one-class SVM for classification. If multiple readings are classified as outliers or anomalous given the legitimate user's model, the application determines that there is an intruder and restricts access to the phone.

Since we assume the attacker has the ability to bypass the device's default lock screen, the application will need to provide a second level of restriction that prevents the intruder from accessing the device's apps and contents. One possible way to implement this is using application pinning, a security feature introduced to the Android operating system in Android 5.0 [8]. Using application pinning, we can force our authentication app to the foreground of the device and prevent the user from accessing any other application on the device. The user must authenticate himself through our application before our application is unpinned from the foreground and the user regains access to the mobile device. The user authenticates himself by entering the password used in registering his account. This password is unknown to the attacker and thus prevents him from bypassing the lockout screen.

4.8 Retraining

Over long periods of time, a user's behavior may change and thus the SVM model used to perform user authentication may become invalid. Thus, it is critical to be able to re-train the SVM model. At the same time, attackers must not be able to reset the SVM model to match their biometrics. A secondary form of user authentication must be performed to ensure the retrain request was sent by the legitimate user. A common method for doing this is by using out-of-band communications like sending a verification email. However, if the attacker is able to access the mobile device, a verification email provides no help since the

attacker can simply open the email app to perform the verification. Thus, each request to retrain the SVM model must be verified through password authentication using the legitimate user's application-specific secret password. Once verified, an API call will be made to PUT /mode with the JSON body set to TRAINING. This will set the user's mode to be retrained and another retraining period will occur over the next 24 hours. Until the new model has been successfully retrained and sent to the client device, the old SVM model is used to keep providing continuous authentication services. We choose to handle retraining in this fashion for two reasons. Continuously recording and transmitting data to the cloud service is likely have a greater impact on device performance as well as including a constant need for a data connection to the cloud server. Furthermore, the SVM library we use does not support online retraining of the model. Thus, there would be a higher resource constraint on our database since it would need to store all collected information from all users.

4.9 Disabling Service and Uninstalling

Much like with retraining requests, we must be wary of requests to disable the continuous authentication service. Once again, an additional layer of authentication is needed. As before, the user must enter their application-specific secret password to successfully disable the authentication service. Implementing a secure, yet simple way to disable the continuous authentication service improves the usability of our app. For example, if the legitimate user wishes to hand his phone to a friend, he can disable the continuous authentication in a matter of seconds before giving the device to the trusted friend. When re-enabling the continuous authentication service, the same SVM model is used as before. No re-training is done unless manually initiated by the user as described above.

Similarly, defense mechanisms are put in place to prevent an attacker from easily uninstalling the app. Upon installation of the app, the user must grant it administrative privileges. This is necessary for the app to perform critical tasks like pinning the application to the foreground on a lockout and locking the phone. Having administrative privileges allows us to set a flag making the program uninstalleable. However, this feature is nullified if the phone user ever removes the administrative privileges through the Android security settings. To account for this, the application can contain a method that is called when the user revokes administrative privileges. The actions taken in this method can vary depending on the user's preferences. One possibility that emphasizes convenience over total security would be to set the phone's lock screen password to a value set when initially configuring user settings and lock the phone screen. This would hinder an attacker, but if they were able to guess the new password, there would be no further defense since the authentication program is no longer running and could be uninstalled easily. A more extreme possibility would be to wipe all phone data upon removal of administrative privileges. This would protect the confidentiality of the sensitive data on the phone, but could significantly inconvenience the legitimate user when the phone is recovered from the attacker. In the case where the legitimate user wishes to disable the administrative privileges of the application without triggering any of these security mechanisms, the app can prompt one last time for the user's application-specific password to prevent them from occurring.

When administrative privileges are revoked, the user's local copy of the SVM model is deleted. Thus, if the application is ever re-installed on the same device, the device will need to re-establish communications with the cloud and request a fresh copy of the model. If the primary user of the device has not changed, the owner can authenticate himself using his application specific password and retrieve the same SVM model used before the app was previously uninstalled. In the case of a new owner, the new user will be unable to supply this password and would need to create a new account using the methods above. Since the device's IMEI is no longer connected to the old account, the server deletes all of the data it stores

pertaining to the unique device ID and stores the new account data connected to the device's new primary user.

5. Prototype

Our team collaborated with David Gilhooley and Tony Jin to create an Android application and corresponding cloud server using an Amazon EC2 instance. Jin and Gilhooley principally designed the client side of our prototype system and we focused on the server side of the system. We were able to successfully construct a system which interfaces with the smartphone and implements most of the REST API specified in Appendix B. We created a database through MongoDB where the cloud server can store incoming sensor data from multiple smartphones. For the client/server interface, we did not implement the specific communications protocols as discussed above, but we were still able to ensure secure communications through SSL communication.

Additionally, we used the data collection application to collect sample data that should be representative of data collected by our system. The data was sampled at a default rate of 1 sample per second. Results were uploaded to the server at the end of the day or by user request through the data collection app. Secure storage in the MongoDB database has been implemented as outlined in Section 4.5.

We also explored the construction of one-class SVMs using the libSVM library. These SVMs used the GCU Dataset [3], a popular dataset containing very similar information to the data we collected using our application. This dataset has been shown to support the idea of continuous authentication by showing how one's sensor data can be used to authenticate the current user of the phone. Unfortunately, due to difficulties with the database we used, we were unable to construct SVMs using the data we collected ourselves. We leave this for future work, but expect to replicate the results from [1].

6. Division of Work

We collaborated extensively throughout all aspects of the project. Such collaboration included proposing attacks and improvements to each other's protocols and system design choices. We split primary authorship of this report in the following manner:

Mike - Prototype server and API, discussion of SGX and server-side system design, attestation protocol, high-level software system design.

Paul - Machine learning, discussion of TrustZone and client-side system design, data transfer protocols, life cycle of authentication app.

7. Conclusion

Modern smartphones contain many preventative measures from allowing an intruder access to the sensitive files and information stored on the device. Fortunately, the sensor subsystem present within mobile devices can provide unique identifying information about how the phone is being interacted with and manipulated. In this report, we present an application which records and analyzes this sensor data to continuously and implicitly authenticate the user of a mobile device. We show how features of ARM architectures including TrustZone and CryptoCell can provide a trusted execution environment to ensure data can be collected safely on the device without worrying about many classes of attackers. We show how Intel's SGX and Amazon Web Services can provide secure execution environments for our cloud server. We also design the communications protocols needed to verify the identities of the clients and server as well as how they transfer sensitive data while ensuring confidentiality and integrity. We detail the lifecycle of the application from downloading the application through uninstallation of the app. At each stage we

discuss which features of Android implement the security features we desire and what a full implementation of our app might incorporate. Finally, we describe our prototype system including a cloud server which collects sensor information from participating Android devices.

Sources

- [1] Wei-Han Lee and Ruby B. Lee, “Multi-sensor Authentication to Improve Smartphone Security”, in Proceedings of the International Conference on Information Systems Security and Privacy, 2015.
- [2] L. Li, X. Zhao, and G. Xue. Unobservable reauthentication for smartphones. In Network and Distributed System Security Symposium. 2013.
- [3] H. G. Kayacık, M. Just, L. Baillie, D. Aspinall, and N. Micallef. Data driven authentication: On the effectiveness of user behaviour modelling with mobile device sensors. Mobile Security Technologies. 2014.
- [4] Dan Branley. Informate Mobile Intelligence First to Measure Smartphone Usage Internationally, Report currently Tracks 12 Countries and Will Expand to 25 by the End of 2015. Informate Mobile Intelligence. February 2015. <http://www.informatemi.com/newsletter10022015.html>
- [5] Intel® Software Guard Extensions (Intel® SGX). Intel Corporation. June 2015. <https://software.intel.com/sites/default/files/332680-002.pdf>
- [6] Matthew Hoekstra. Intel SGX for Dummies (Intel SGX Design Objectives). Intel Developer Zone. November 2015. <https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx>
- [7] B. Schölkopf, J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. Neural Computation. 2001.
- [8] Android 5.0 APIs. Google. <http://developer.android.com/about/versions/android-5.0.html#ScreenPinning>
- [9] Intel® Software Guard Extensions Programming Reference. Intel Corporation. October 2014. <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>
- [10] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). <http://tools.ietf.org/html/rfc5869>
- [11] TrustZone. ARM. 2015. <http://www.arm.com/products/processors/technologies/trustzone/>
- [12] Fundamentals of ARMv8. ARM Cortex-A Series Programmer’s Guide for ARMv8-A. 2015. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CIHGCIDG.html>
- [13] Steven J. Vaughan-Nichols. Amazon EC2 cloud is made up of almost half-a-million Linux servers. ZDNet. March 2012. <http://www.zdnet.com/article/amazon-ec2-cloud-is-made-up-of-almost-half-a-million-linux-servers/>




- [14] Amazon Web Services: Overview of Security Processes. August 2015.
<https://d0.awsstatic.com/whitepapers/aws-security-whitepaper.pdf>
- [15] Pramod Jamkhedkar, Jakub Szefer, Diego Perez-Botero, Tianwei Zhang, Gina Triolo and Ruby B. Lee. A Framework for Realizing Security on Demand in Cloud Computing. IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bristol, UK. 2013.
- [16] Signing Your Applications. <http://developer.android.com/tools/publishing/app-signing.html>
- [17] David Champagne and Ruby B. Lee. Scalable Architectural Support for Trusted Software. The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA). January 2010.
- [18] J. Crhistopher Bare. Attestation and Trusted Computing. CSEP590: Practical Aspects of Modern Cryptography. March 2006.
- [19] Node.js Foundation. Node.js. <https://nodejs.org/en/>
- [20] Express – Node.js web application framework. <http://expressjs.com/>

Appendix

A. What is a REST API?

REST stands for REpresentational State Tranfer and is a very popular way to handle client server communications. REST systems use the same HTTP verbs (GET, POST, PUT, DELETE, etc.) and typically runs of HTTP or HTTPS. The REST API will specify which HTTP verbs are available on different URIs (Uniform Resource Identifiers). For instance, in our system we allow the POST verb on the /sensor URI for the user to upload sensor data for his device. An important implementation detail of the REST architecture is that the server code is stateless. Therefore, all of the information necessary to handle a client's request should be available in the current request. If there is information that is necessary to too be stored across sessions, such as information for authenticating a user, it is typically stored in a database rather than in memory.

B. Our REST API

| METHOD | URI | NOTES | RESPONSES | Implemented in Prototype? |
|--------|-----------|--|--|---|
| POST | /register | This method is used to begin the attestation protocol. This protocol is outlined in Section 4.1. | 200: Responds with r used for the client attestation |  |
| PUT | /register | This method is used to respond to the attestation request. For more details on the response refer to Section 4.1. | 200: If the attestation report has been verified. Sends server attestation report to the client. 403: If the attestation report was invalid. | |
| GET | /nonce | This method is is used by the client to get a nonce from the server that can be used by the client for a subsequent request. | 200: Responds with r for the client to use in its next request. |  |
| POST | /sensor | This method is used to upload sensor data. The sensor data is sent along with the request as JSON encoded data. | 200: If the data integrity and device are verified. 400: If the data integrity failed or the device did not match the signature. |  |
| POST | /model | This method is used to create a new model given the user data that is already present in the database. | 200: If the data integrity and device is verified. Creating the model takes ~1hr so this just responds with ok. 400: If the device did match the signature. | |

| | | | | |
|-----|--------|--|--|---|
| GET | /model | This method will return the model if it is ready for the user. | 200: If the model is ready and the device is verified, return the model. 204: If the model is not ready and the device is verified. 400: If the device did not match the signature. | |
| PUT | /mode | This route will change the current mode of the application. The three modes are Off, Training, and Continuous Authentication. The new mode will be sent in JSON. | 200: If the data integrity and device are verified. 400: If the data integrity failed or the device did not match the signature. | ✓ |

C. Exception levels in TrustZone

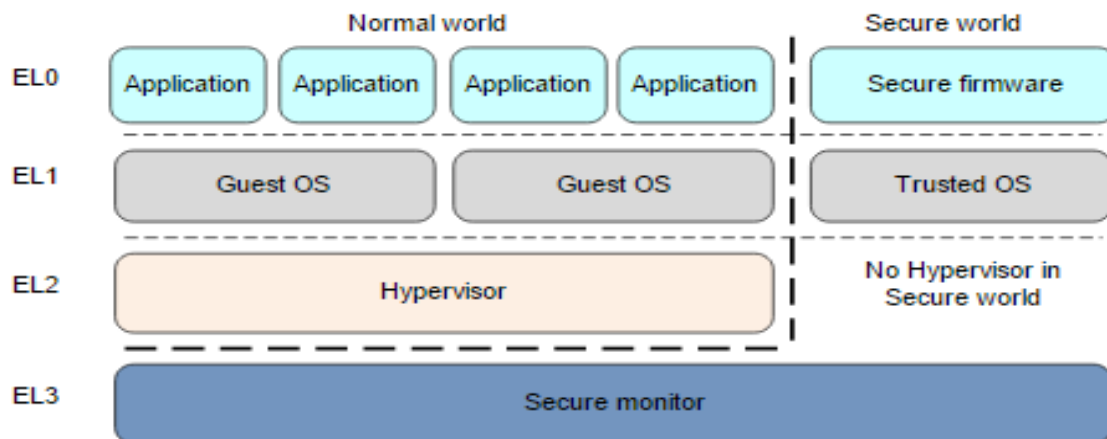


Figure 8: ARMv8 Exception levels in the normal and secure worlds [12].

D. Amazon AWS

The AWS infrastructure hosts virtual machines that users can purchase with pricing depending on their storage or CPU requirements. The machines utilize a Xen hypervisor and it is probable that is running a customized version of Red Hat Enterprise Linux (RHEL) as its base operating system [13]. Therefore, Xen runs as a Type 2 VMM on top of RHEL and the user can create use a guest OS that is running Linux, Solaris, Windows or BSD on top of Xen.

AWS holds themselves to high security standards and provides a lot of information on their networking security in their security white paper [14]. For instance, with AWS you can limit access to your device to only certain IP subnets or other Amazon instances. One can set up an access control policy to limit access to their

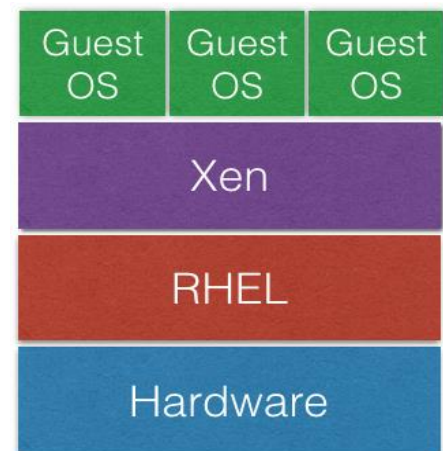


Figure 9: Amazon's exact system stack has never been released; however, researchers believe this is Amazon's System Stack.

Amazon S3 instance, which is used for data storage, to only a certain Amazon EC2 instance. Therefore, these policies will only allow the backend application to query the database. Furthermore, Amazon assures their user that virtual machines running on the same hardware cannot receive traffic that is not meant for them. However, Amazon does not provide protection against another VM viewing the traffic to a different VM. Beyond networking security, Amazon EC2 instances are vulnerable to many attacks within the VM and from other VM's on the machine. While our threat model assumes an attacker can bypass the networking security provided by Amazon, it is important to know the security that is currently offered by the AWS infrastructure

E. Important SGX Instructions [5] [9]

| Method | Description | Function Type |
|---------|--|---------------|
| ENCLS | Execute an Enclave System Function of Specified Leaf Number | N/A |
| ENCLU | Execute an Enclave User Function of Specified Leaf Number | N/A |
| EADD | Add a page to an uninitialized Enclave | System |
| EAUG | Add a page to an initialized Enclave | System |
| ECREATE | Create an SECS page in the Enclave Page Cache. SECS structures include base address, size, attributes, and other identifying features of the enclave. | System |
| EEXTEND | Extend uninitialized enclave measurement by 256 bytes. Used for cryptographically measuring the enclave during creation in order to verify the enclave was created successfully during enclave creation. | System |
| EINIT | Initialize an enclave for execution. This is the final instruction executed in the build process. | System |
| EREMOVE | Removes a 4K page from the enclave. | System |
| EENTER | Enters a enclave | User |
| EEXIT | Exit an enclave | User |
| EGETKEY | Retrieves a 128 bit secret key from the processor based on a processor specific value and KEYREQUEST structure. | User |
| EREPORT | Creates a cryptographic report the describes the contents of the enclave | User |
| ERESUME | Re-enter an enclave after an interrupt | User |

F. Creating an Enclave [5] [9]

1. ECREATE: This is responsible for creating the unique enclave. This call will establish the address range for the enclave. This will also create the Secure Enclaves Control Structure (SECS), where enclave specific information is stored such as the mode (32/64) and debug mode.
2. EADD: The sensitive code is initially stored in untrusted memory, and must be added to the enclave. The system software is responsible for selecting a free enclave page and setting the attributes and content of the page.
3. EEXTEND: Since SGX relies on system software to add the pages to the enclave, the EADD instructions must be protected against integrity attacks. This method does just that by generating a cryptographic hash of the content of the enclave. It preforms the hash over 256 bytes chunks, so this must be repeated 16 times in order to take a hash over the entire 4K page. Each call to EEXTEND adds the cryptographic log with the section being measured and measurement itself. These entries in the cryptographic log are critical to determine if the system software which is not trusted created the enclave correctly.
4. EINIT: As long as the measurements taken during initialization are match the expected measurements, this will mark the enclave as ready to be used. This involves validating the SIGSTRUCT is signed, the measurements in the cryptographic log match the expected measurement, and that the attributes are correct. This SIGSTRUCT is an enclave signature structure which specified the enclaves expected measurement, attributes and the independent software vendor (ISV) information.
5. EENTER: Now that the enclave has been set up, this instruction enters the enclave for execution. This will transfer control to a pre-determined location inside the enclave.
6. EEXIT: This will exit the enclave and return to normal execution. It is important to clear the TLB entries for enclave addresses upon exiting.

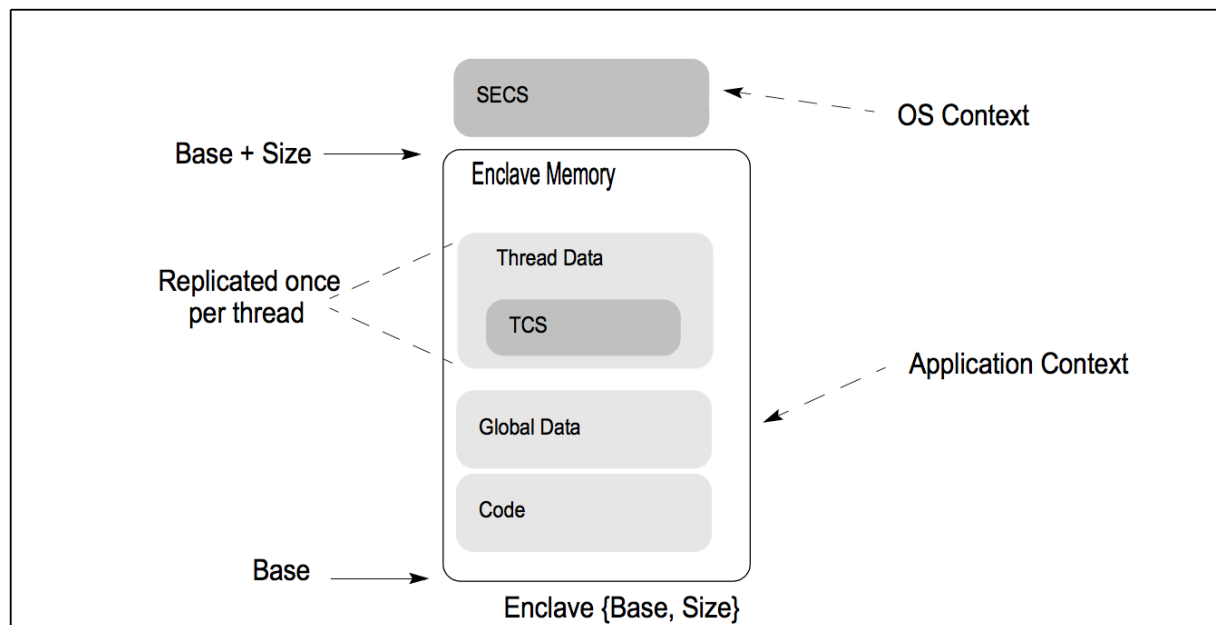


Figure 10: Memory layout of an enclave. [9]

G. Report Layout in SGX

Table 2-21. Layout of REPORT

| Field | OFFSET (Bytes) | Size (Bytes) | Description |
|------------|----------------|--------------|---|
| CPUSVN | 0 | 16 | The security version number of the processor. |
| MISCSELECT | 16 | 4 | SSA Frame specified extended feature set bit vector |
| RESERVED | 20 | 28 | Must be zero |
| ATTRIBUTES | 48 | 16 | The values of the attributes flags for the enclave. See Section 2.7.1 (ATTRIBUTES Bits) for the definitions of these flags. |
| MRENCLAVE | 64 | 32 | The value of SECS.MRENCLAVE |
| RESERVED | 96 | 32 | Reserved |
| MRSIGNER | 128 | 32 | The value of SECS.MRSIGNER |
| RESERVED | 160 | 96 | Zero |
| ISVPRODID | 256 | 02 | Enclave PRODUCT ID |
| ISVSVN | 258 | 02 | The security version number of the Enclave |
| RESERVED | 260 | 60 | Zero |
| REPORTDATA | 320 | 64 | A set of data used for communication between the enclave and the target enclave. This value is provided by the EREPORT call in RCX. |
| KEYID | 384 | 32 | Value for key wear-out protection |
| MAC | 416 | 16 | The CMAC on the report using report key |

Figure 11: The report layout for an attestation report used by SGX. The REPORTDATA is a user specified 64 byte buffer that the EREPORT instruction will use when generating the cryptographic report [9].

H. Code

```
'use strict';

//mongoose set up
require( './db' );

var debug      = require('debug')('app');
var express    = require('express');
var path       = require('path');
var logger     = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var apiApp = require('./apiApp');

//set up the express app
var app = express();

app.use(logger(app.get('env') === 'production' ? 'combined' : 'dev'));

//user body parser to handle json inputs
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());

//register the api application on /
app.use('/', function(req, res, next) {
  apiApp(req, res, next);
});

// error pages
app.use(function (err, req, res, next) {
  res.status(500);
  res.send('<pre>' + err.stack + '</pre>');
});

//listen on port 3000
var port = 3000;

app.set('port', port);

app.listen(app.get('port'), "0.0.0.0", function () {
  debug('Express ' + app.get('env') + ' server listening on port ' + this.address().port);
});
```

Figure 12: app.js is used as to set up the server listening on port 3000.

```

var mongoose = require( 'mongoose' );
var Schema   = mongoose.Schema;

var Sensor = new Schema({
  DEVID: String,
  TIMESTAMP : String,
  GPS1 : String,
  GPS2 : String,
  GPS3 : String,
  ACCEL1: String,
  ACCEL2: String,
  ACCEL3: String,
  GYRO1: String,
  GYRO2: String,
  GYRO3: String,
  MAGNET1: String,
  MAGNET2: String,
  MAGNET3: String,
  LIGHT: String,
  PROXIMITY: String,
  TEMP: String,
  PRESSURE: String,
  SIG: String
});

var User = new Schema({
  DEVID: String,
  Model: String,
  Mode: String,
  PK: String,
});

mongoose.model( 'Sensor', Sensor );
mongoose.model( 'User', User );
mongoose.connect( 'mongodb://admin:' + process.env.MONGO_PASS + '@ds039135.mongolab.com:39135/ele-final-project' );

```

Figure 13: db.js is used to set up the different tables in the database.

```

var app = require('express')();
var mongoose = require( 'mongoose' );
var Sensor    = mongoose.model( 'Sensor' );
var User      = mongoose.model( 'User' );
var crypto    = require('crypto');
var bigInt    = require("big-integer");

//use a big integer for the nonce
//the nonce is use to prevent replay attacks
var nonce = bigInt(1);

//create the user encryption key using the master key and devid
var createEncKey = function(devid) {
  var masterKey = process.env.ENCRYPT;
  return crypto.createHmac('sha256', masterKey).update(devid).digest('hex');
}

//create the authentication key using the encryption key and devid
var createAuthKey = function(encKey) {
  var masterKey = process.env.ENCRYPT;
  return crypto.createHmac('sha256', masterKey).update(encKey).digest('hex');
}

//use AES256 CBC Mode encryption
var encrypt = function(key, text){
  var cipher = crypto.createCipher('aes-256-cbc', key);
  var crypted = cipher.update(text,'utf8','hex')
  crypted += cipher.final('hex');
  return crypted;
}

//use HMAC SHA256 for the signature
var signature = function(authKey, entry) {
  return crypto.createHmac('sha256', authKey).update(entry).digest('hex');
}

```

Figure 14: apiApp.js uses the crypto libraries defined by NodeJS. For the real application we plan to use a Java module for encryption and decryption.

```

//respond with a challenge
app.post('/register', function(req, res) {
  res.status(200);
  res.send({'challenge' : Math.floor(Math.random() * 999999999)});
});

//this method will eventually have to assert an attestation report
//currently this route is just checking the dev_id has already
//been registered
app.put('/register', function(req, res) {
  var data = req.body;
  var dev_id = req.get('userid');
  console.log(dev_id);
  console.log(data);
  User.find({'DEVID': dev_id}, function(err, curUser) {
    if (curUser.length > 0) {
      res.status(400);
      res.send({'message' : 'This device is already registered'});
      return;
    }
    else {
      new User({
        DEVID: dev_id,
        Model: '',
        Mode: 'OFF',
        PK: ''
      }).save( function ( err, user, count ){
        if (err) console.log(err);
      });
      res.status(200);
      res.send({'id': dev_id});
    }
  });
});

//respond with a nonce and increment the nonce
//this method should also place the current nonce
//associated with the user in the user table
app.get('/nonce', function(req, res) {
  res.status(200);
  res.send({'nonce' : nonce++});
});

```

Figure 15: The implementation of POST /register, PUT /register, and GET /nonce. PUT /register will be responsible for handling attestation in the future. Currently it only checks if the user has already registered.


```

//store the sensor data securely
app.post('/sensor', function(req, res) {
  var data = req.body;
  var dev_id = req.get('userid');
  console.log(dev_id);
  console.log(data);

  var encKey = createEncKey(dev_id);
  var authKey = createAuthKey(encKey);

  for (entry in data) {
    var sensorData = data[entry];
    new Sensor({
      DEVID: dev_id,
      TIMESTAMP : encrypt(encKey, sensorData.TIMESTAMP),
      GPS1 : encrypt(encKey, sensorData.GPS1),
      GPS2 : encrypt(encKey, sensorData.GPS2),
      GPS3 : encrypt(encKey, sensorData.GPS3),
      ACCEL1: encrypt(encKey, sensorData.ACCEL1),
      ACCEL2: encrypt(encKey, sensorData.ACCEL2),
      ACCEL3: encrypt(encKey, sensorData.ACCEL3),
      GYRO1: encrypt(encKey, sensorData.GYRO1),
      GYRO2: encrypt(encKey, sensorData.GYRO2),
      GYRO3: encrypt(encKey, sensorData.GYRO3),
      MAGNET1: encrypt(encKey, sensorData.MAGNET1),
      MAGNET2: encrypt(encKey, sensorData.MAGNET2),
      MAGNET3: encrypt(encKey, sensorData.MAGNET3),
      LIGHT: encrypt(encKey, sensorData.LIGHT),
      PROXIMITY: encrypt(encKey, sensorData.PROXIMITY),
      TEMP: encrypt(encKey, sensorData.TEMP),
      PRESSURE: encrypt(encKey, sensorData.PRESSURE),
      SIG: signature(authKey, entry)
    }).save( function ( err, sensor, count ){
      if (err) console.log(err);
    });
  }
  res.status = 200;
  res.send({'code': 'ok'});
});

//handling the ML model is not implement
app.post('/model', function(req, res) {
  res.status(501);
  res.send();
});
app.get('/model', function(req, res) {
  res.status(501);
  res.send();
});

```

Figure 16: The implementation of POST /sensor using user derived keys based on the master key. POST /model and GET /model have not been implemented in the prototype.

```

//update the mode of the current user
app.put('/mode', function(req, res) {
  var data = req.body;
  var dev_id = req.get('userid');
  console.log(dev_id);
  console.log(data.mode);
  var mode = data.mode;
  var user = new User({
    DEVID: dev_id,
    Model: '',
    Mode: mode,
    PK: ''
  });

  var updateData = user.toObject();

  // Delete the _id property, otherwise Mongo will return a "Mod on _id not allowed" error
  delete updateData._id;

  //must implement mode validation and sanitization
  User.findOneAndUpdate({'DEVID' : dev_id}, updateData, function(err, raw) {
    if (raw == null) {
      res.status(400);
      res.send({'message': 'Invalid device id'});
    }
    else {
      res.status(200);
      res.send({'message': 'Updated mode to ' + mode});
    }
  });
});

module.exports = app;

```

Figure 17: The implementation of PUT /mode used to update the user's mode. This method still needs to make sure the mode transition is valid.