622.755 – Crypto Engineering Winter Term, 2023/24

Project: SuperSAM Security Analysis

Alessandro Castelli

1 Executive Summary

The provided code and documentation exhibit significant deficiencies that compromise the system's security. These issues stem from flawed algorithm implementations, non-compliance with standards, and insufficient attention to the critical aspects of certain algorithms that require specific adjustments for secure operation.
The Implementation Note contains various flaws, with often superficial explanations that make it challenging to understand the rationale behind certain design choices. In particular, serious vulnerabilities arise from the use of AES in ECB mode for encoding certain token information. These vulnerabilities will be examined in detail later. Other errors involve inadequate security assumptions for certain algorithms, the criticalities of which will be illustrated.
The majority of errors in the Implementation Note focus on Diffie-Hellman, employed in an insecure manner. Specific errors include the absence of authentication, inappropriate generator selection, incorrect output size, and others. Adopting these approaches in the actual implementation could result in severe security breaches.
To ensure a robust and secure system, it is imperative to rectify erroneous implementations, adopt secure algorithms, and adhere to best design practices. Documentation should be improved to provide detailed explanations of design choices and clearly highlight the security measures implemented. Specifically, abandoning the use of AES in ECB mode, correcting Diffie-Hellman-related errors (which are critical), and developing and understanding customized best practices for SuperSAM are essential for ensuring robust system security.
It is also crucial to emphasize that significant improvements have not been made regarding the use of RFID. While this technology indirectly poses numerous security issues, they have not been addressed as they were not requested. A comprehensive analysis of this technology should be considered for a more thorough management of potential vulnerabilities.
The implementation section also presents significant issues, including one of the most serious: the initialization of seeds to zero. This practice renders the masking completely useless, leaving the system vulnerable to potential Differential Power Analysis (DPA) attacks. Addressing this problem promptly is essential to preserve the system's security.
Furthermore, the choice of the System Master Key appears too obvious, potentially facilitating attackers. The system's robustness largely depends on the secrecy and complexity of the keys used. It is advisable to review the selection of the System Master Key to make it less predictable and enhance resistance to attacks.

In addition to these severe problems, other minor errors can be identified that, while compromising the system to a lesser extent, should not be overlooked. Correcting these minor flaws will contribute to ensuring an overall more secure system.
Moreover, a comprehensive review of the implementation section is recommended, following security best practices and adopting additional measures to protect the system from potential threats.
These improvements will not only address current issues but also contribute to making the entire system more resilient and dependable.

2 Specific points

2.1 Implementation Note - AES in ECB Mode: Section 2.3, Page 3

Problem Description: In Section 2.3, midway through Page 3, it specifies the use of AES-128 in ECB mode for encrypting tickets. This choice introduces a significant vulnerability due to the way ECB mode handles encryption.
The main issue arising from this implementation is that each plaintext block is encrypted independently of others. Consequently, identical blocks of plaintext will produce identical blocks of ciphertext. Additionally, input blocks always have the same size.
This approach poses various risks in the context of SuperSAM:

1 **Copy and Paste of the Ticket:** In cases where multiple users share the same System Master Key (SMK), it is possible to copy and paste the encrypted ticket from one token to another. This poses a risk of gaining access to buildings for which the user should not be authorized.

2 **Possibility of Pattern Detection:** ECB mode is vulnerable to pattern detection. Since identical blocks of plaintext generate identical blocks of ciphertext, an attacker could identify recurring structures and deduce information about the nature of the encrypted data.

Demonstration of Vulnerabilities:

1. In the context of SuperSAM, the `ticket` is encoded with AES-128 using the SMK. If multiple employees share the same SMK (as admitted in the implementation notes), an attacker could acquire access to the `EncTicket = AES.ECB.128(Ticket, SMK)` of a specific device. Since ECB mode encrypts blocks independently, the attacker can copy specific 128-bit blocks. Even without knowing the actual content of `EncTicket`, they could use this encoded block (or part of it) on other devices with the same SMK, potentially gaining access to unauthorized buildings.

2. The implementation notes indicate that `ticket = buildID1, buildID2, ...`, where each `building` is an unsigned 32-bit integer `uint32`. Suppose building IDs are unique and arranged in ascending order. This could lead to a potential attack based on the ECB mode of AES. If, during encoding, a common pattern is observed among the encodings of $M = m_1, m_2, m_3, ..., m_n$ where M is the plaintext, an attacker could, by decoding the pattern, fraudulently add permissions to a device. For example, if the identified pattern is $ENC(m_1) = 00100, ENC(m_2) = 00101, ENC(m_3) = 00110, ...$, the attacker could attempt to add access to m_4 by adding to the ticket $ENC(m_4)$ presumably equal to 00111. Recognizing a complex pattern might be challenging, but the possibility remains.

Proposed Improvement To address both mentioned issues, it is necessary to adopt a different encryption mode. One protocol that tackles these challenges is AES-CBC.
In AES-CBC, each encrypted text block depends not only on the corresponding plaintext block but also on all previously encrypted text blocks. This interdependence between blocks enhances the security of the system.
In AES-CBC, the block interdependence prevents the exploitation of the vulnerability described in 1. Let's assume we have two tickets encrypted with AES-CBC: $Enc_{ticket_1} = AES.CBC.128(ticket_1, SMK)$ and $Enc_{ticket_2} = AES.CBC.128(ticket_2, SMK)$. These produce respective ciphertexts $C_1 = c_1^1, c_1^2, \ldots, c_1^n$ and $C_2 = c_2^1, c_2^2, \ldots, c_2^m$. It will not be possible to copy a ciphertext block, such as c_1^2, and paste it at the end or in the middle of C_2. This is because c_1^2 depends on c_1^1 and the IV used in encoding. Copying and pasting it into another context would create errors and generate meaningless ciphertext.
Furthermore, the use of AES-CBC prevents the exploitation of the vulnerability described in 2. This is because even modifying a single bit in the encrypted block would have a significant impact on all subsequent blocks during decryption. Moreover, the XOR chain introduces randomness into the process, making it challenging to analyze patterns within the encrypted data.

2.2 Implementation Note - Wrong Generator: Section 2.3 on page 3

Description of the Problem: In Section 2.3, page 3, it is indicated that a value equal to 3 is used for Diffie-Hellman, however, this value is incorrect. The DH parameters provided are sourced from RFC 3526 (you can find them on page 3, Section 2.3), and there the value of the generator is specified as 2 and not 3.

Demonstration of Vulnerabilities: A generator g has the property of generating the entire prime group, and this characteristic is of fundamental importance as it makes the discrete logarithm problem computationally difficult to solve with modern computers.
If an element r is chosen that is not a generator, it is possible that it generates only a much smaller subgroup. This circumstance could make the discrete logarithm problem

solvable in reasonable time.
Let's consider a prime number p, and suppose the generator is chosen to be 3. In the case where 3 is a valid generator, then every element of the group generated by $3 \mod p$ must be coprime with p. However, the error lies in the fact that we cannot guarantee that the generator equal to 3 has these properties since it is not the number specified in the standard. Therefore, choosing 3 poses the risk that it may not be a good generator, making the discrete logarithm problem easy to solve.

Proposed Improvement The best solution would be to use the generator proposed by RFC 3526, which is 2. Alternatively, it is possible to search for another generator. To find an optimal generator, one must find a number such that: g generates all elements of the cyclic group associated with the prime modulo p, the order of g is large, all elements generated by g are coprime with p, and there are no cycles of small order. Therefore, in our case, any generator that is itself a prime number, including 3 or 5, would theoretically work well. It is important that the chosen generator is not too large to avoid excessive memory and computational burden.

2.3 Implementation Note - CBC-MAC: Section 2.3 on page 3

Description of the Problem: In Section 2.3, page 3 towards the end, it is stated that

> To convert the outcome of a DH key exchange to a symmetric session key we will use AES as a block cipher to implement CBC-MAC with a fixed key equal to zero. This takes the 1536 bit outcome of DH and produces a 128 bit AES session key.

The use of CBC-MAC as a method to derive the key obtained with Diffie-Hellman (DH) presents potential security issues for several reasons:

1. **Improper Use of CBC-MAC:** CBC-MAC is commonly used as a message authentication protocol, generating a Message Authentication Code (MAC) from a variable-length message. In this case, it is used as a key derivation method, employing AES as the encryption protocol. This improper use could compromise the security of key derivation.

2. **Fixed Key Equal to Zero in CBC-MAC:** The use of CBC-MAC with a fixed key equal to zero is problematic. This choice of a fixed key makes the procedure susceptible to attacks since the key can be easily deduced by an attacker.

3. **Lack of Authentication in DH Exchange:** The Diffie-Hellman key exchange, by itself, does not provide any authentication of the involved parties. The use of CBC-MAC with a fixed key equal to zero does not address this issue. In a Man-in-the-Middle attack context, CBC-MAC does not add additional layers of protection.

Demonstration of Vulnerabilities Considering the context in which a Diffie-Hellman (DH) key exchange occurred with a subsequent Man-in-the-Middle (MITM) attack, the key reduction process represents a critical point in the system's security. The analysis of this phase, which will be further explored in the subsequent section 2.5, reveals some significant vulnerabilities.
After successfully obtaining the result of the DH exchange, the attacker is now faced with the need to perform key reduction to initiate fraudulent communication. In this phase, the reduction is executed using CBC-MAC, a protocol that, in this case, leverages AES with a fixed key set to zero.
The choice of a fixed key equal to zero within CBC-MAC poses a serious security risk. Such a key is easily identifiable and susceptible to attacks, such as a dictionary attack, which could allow the attacker to determine the key without excessive difficulty.
The direct implication of this vulnerability is that the attacker, having successfully reduced the key, is capable of initiating unwanted communication without encountering significant obstacles. This raises concerns about the confidentiality and integrity of the exchanged information, as the attacker might be able to decipher or manipulate the transmitted data.
There is also the additional problem that SuperSAM is using CBC-MAC to derive a key, while CBC-MAC is an algorithm originally designed and used to authenticate messages employing block ciphers. The use of CBC-MAC for a purpose different from its original intent implies the possibility of vulnerabilities that would not emerge if CBC-MAC were used in its original context.
Furthermore, since a Diffie-Hellman (DH) agreement lacks an intrinsic authentication mechanism, one might consider using the key reduction step to add a minimal level of authentication. However, this opportunity is not exploited due to the imprudent choice of the key, as adopting a key fixed to zero makes the system susceptible to compromise relatively easily.

Proposed Improvement One key aspect to enhance security is to avoid the use of a fixed key equal to 0, which is vulnerable even to simple brute-force attacks. In this context, a proposed solution is the adoption of the System Master Key (SMK) instead of a fixed key. The SMK is known to both the Reader and the Token and undergoes regular updates, as indicated in the Implementation Note in the paragraph `SuperSAM System Master Key is updated`, making it a more secure and dynamic choice.
In the case of a potential Man-in-the-Middle (MITM) attack, using the SMK as the secret key for CBC-MAC reduction represents an additional layer of security. Unless the attacker can obtain the SMK through other means, they would not be able to acquire the session key for subsequent communications, as they do not know the key required to perform the CBC-MAC.
However, it is suggested to consider alternatives to CBC-MAC, such as using a protocol specifically designed for hash generation, like SHA-3. SHA-3 is known for its resistance to various cryptographic attacks and could be a secure choice for hash generation in this

context. It is important to note that SHA-3 is a one-way hashing algorithm and is not specifically designed for authentication, which should be treated as a separate issue and will be extensively discussed in Section 2.5.

2.4 Implementation Note - Section 2.3, Page 3. Length of Diffie-Hellman Output: 1536 bits

Problem Description In the implementation notes of SuperSAM (in section 2.3), it is specified that the output of the Diffie-Hellman (DH) phase before undergoing CBC-MAC is 1536 bits. However, this length is not considered sufficient to ensure security during the DH exchange.

Demonstration of Vulnerability The issue lies in the fact that the output of the SuperSAM DH process is only 1536 bits, a length that raises concerns about the robustness of the generated shared key. In accordance with the guidelines of the National Institute of Standards and Technology (NIST), expressed in the document SP 800-56A Rev. 3, specific indications are provided regarding the security of schemes based on the discrete logarithm problem.
The NIST document suggests that to obtain a shared key robust enough to resist known attacks, it is necessary to use a key of at least 2048 bits. This length is considered suitable to ensure a level of security adequate for most known attacks against DH key exchange schemes.
Using keys shorter than 2048 bits could make the shared key more vulnerable to known cryptographic attacks, as the key length directly influences the computational complexity required to solve the discrete logarithm problem. Attacks such as factorization of large prime numbers or the use of advanced algorithms might become more effective with key lengths below 2048 bits. Therefore, the use of shorter keys could compromise the robustness of the shared key and the overall security of the Diffie-Hellman key exchange system.

Proposed Improvement The proposed solution is to increase the length of the Diffie-Hellman output to 2048 bits. To implement this change, it is necessary to update the key generation parameters in SuperSAM.
This modification is essential to ensure robust resistance against known attacks based on the discrete logarithm problem.
Furthermore, it is recommended to conduct a detailed analysis to assess the impact of this change on the overall system performance. Further optimization of the implementation may be necessary to ensure that the increased key length does not significantly compromise performance during the DH exchange.
In conclusion, to mitigate the identified vulnerability, it is advisable to implement the described modification to ensure that the shared key generated during the DH exchange

has a length of at least 2048 bits, thereby providing a level of security in line with current cryptography best practices.

2.5 Implementation Note - Authenticate: Section 2.4 on page 4, authenticate paragraph

Description of the Problem: In section 2.4, page 4, in the paragraph related to authentication, it is mentioned that parties obtain a session key through a Diffie-Hellman exchange. However, it is important to note that the Diffie-Hellman protocol is susceptible to Man-in-the-Middle attacks, where an attacker can impersonate one of the two involved parties without the other party being aware. In clearer terms, this means that during the key exchange, a malicious actor could intercept the communication between the two parties and impersonate both the sender and the receiver. This opens the door to potential security threats, as the attacker could manipulate or intercept sensitive information without the legitimate parties being aware of such interference. In the implementation notes, it is specified that the token proves its legitimacy by sending its SecureID to the reader (in section 2.4.1, paragraph `SuperSAM Token is used to gain building access`). Still, this is not sufficient for effective token authentication, both because the SecureID can be taken and used by the attacker in subsequent DH sessions (thus not sufficient for token authentication) and because the reader does not authenticate itself.

Demonstration as a Vulnerability: The Diffie-Hellman (DH) protocol is secure for key exchange over an insecure public channel, as it relies on the discrete logarithm problem, computationally infeasible to calculate if parameters are chosen correctly. However, this protocol is subject to Man-In-The-Middle attacks, where an attacker intercepts communication between two subjects and impersonates one of them. In the SuperSAM implementation notes, during an attack, it could happen that when the token initiates communication with `Hello`, the legitimate reader does not respond, and the attacker sends its DH value to the token. At that point, the token responds with the `SecureID` and its DH value. At this stage, the attacker is recognized by the token as the legitimate reader and can interact with the token, receiving the encrypted ticket and sending a new ticket to grant or deny access to other buildings. If, on the other hand, the attacker pretends to be the token (having already obtained the token's Secure ID in the previous step), the attacker can receive a new valid System Master Key (as indicated in the implementation note, paragraph 2.4.1, section "SuperSam Ticket is updated").

Proposed Improvement: To address the authentication issues of Diffie-Hellman (DH) and implement a secure and efficient authentication protocol, the use of digital signatures is proposed. However, it is essential to consider that implementing this protocol might require a significant amount of memory, as digital signatures involve public-key cryptography. Since both the token and the reader must store their respective private keys and all necessary public keys, it becomes evident that this form of authentication is limited by the memory capacity of the token. Compact and versatile, token devices may not have a large storage capacity. Therefore, it is crucial to consider the scalability of this solution, as the number of public keys to store may limit the protocol's effectiveness. In summary, this solution is practical as long as the number of involved devices is limited. However, it becomes less scalable as the number of public keys to manage increases, as devices with limited memory may struggle to handle an excessive amount of information. To make authentication more scalable, exploring alternatives may be necessary. A possible alternative is to use a pre-shared System Master Key (SMK) between the two involved parties. A recommended practice is to encrypt the Diffie-Hellman (DH) value with a highly secure algorithm, such as AES-128, both on the token and the reader using the SMK. This approach is designed to mitigate Man-in-the-Middle attacks, as even in case of interception, the attacker would only see the encrypted DH value and would not be able to extract the session key required for subsequent communications. Knowledge of the SMK is limited to the legitimate reader and token, making it difficult for the attacker to discover the base key needed to decipher the data. In implementing this method, choosing a robust System Master Key becomes crucial. Some best practices for building a secure SMK include:

- Ensure that the key length is at least 128 bits.
- Create the key using a random number generator to ensure the absence of predictable patterns.
- Regularly change the key to prevent the accumulation of potential long-term vulnerabilities.
- Securely store the key to prevent unauthorized access.
- Implement a key rotation policy to limit continuous exposure of the same key.
- Use Hardware Security Modules (HSMs) for key generation and management, if possible.
- Monitor and log activities related to the SMK to detect potential anomalies or unauthorized access.

In the implementation note provided to us, it is not explained how the System Master Key is created and shared, but assuming that it is created and distributed securely, the above solution could be a good way to ensure that the token and the reader are legitimate and thus prevent Man-In-The-Middle attacks.

2.6 Implementation Note - Send Ticket - UpdateTicket - UpdateSystemMasterKey: Section 2.4 on page 4

Description of the problem: In the presence of a Man-In-The-Middle attack, a potential risk is the attacker's ability to update or discover the system's System Master Key,

read the ticket, or send a new valid ticket to a token. This problem is closely related to the lack of authentication in the Diffie-Hellman protocol, as previously illustrated. In the case where authentication in DH occurs correctly, such issues would not exist.
Let's now consider a situation where the authentication problem has not been resolved and examine fraudulent communication.

Demonstration of the vulnerability: Suppose the attacker is impersonating either the reader or the token. In the case where the attacker pretends to be the reader, they could read the ticket sent by the token because it is encoded with the session key obtained by the attacker. Additionally, the attacker might be able to update the `System Master Key` (encoded with the session key) and the `SecureTicket`.
In the scenario where the attacker pretends to be the token (as described in paragraph 2.5), they could receive a new System Master Key that is completely valid and functional.

Proposed improvement: This problem can be addressed in two ways. A first method is to implement an upstream authentication process, as explained in paragraph 2.5. In case the issue of failed authentication persists, a possible solution would be to perform double encryption, encoding the messages sent with both the session key obtained during the Diffie-Hellman exchange and the system's System Master Key, which we assume only legitimate readers and tokens know. In this case, even in the event of a Man-In-The-Middle attack, it would not be possible to extract useful information from the received data, and it would not be possible to send valid messages.

2.7 Implementation Note - SPA Attack on DH, Page 6, Section 3.3

Description of the problem: In Section 3.3, it is stated that it is not necessary to implement protections against SPA attacks because DH operates on very large values, making such attacks practically impossible. This statement is incorrect, as SPA attacks are still feasible regardless of the size of the input. In fact, the input size is not a determining factor for security against SPA attacks.

Demonstration of the Vulnerability: An SPA (Simple Power Analysis) attack is a cryptographic attack technique that exploits information obtained by measuring the power consumed by a cryptographic device during the execution of an algorithm. SPA attacks rely on the principle that different cryptographic operations require varying amounts of power, and variations in power consumption can reveal information about the nature of the operation being performed. By carefully analyzing these power variations, an attacker could extract sensitive information.

Regarding the applicability of SPA to Diffie-Hellman (DH), it is important to note that DH involves modular multiplication operations on very large numbers. These operations can lead to distinct changes in power consumption, and an attacker might attempt to leverage these variations to deduce information about the DH private key. However, it is crucial to note that the specific implementation and adopted countermeasures can influence the feasibility of an SPA attack on a DH system.

For instance, the implementation note mentions the use of Montgomery, which is indeed a good idea but may not be sufficient. This technique involves pre and post-calculation parts that could still be susceptible to SPA attacks. Online resources also discuss SPA attacks on DH, as seen in this article.

Proposed improvement: Without a specific implementation of DH used by SuperSAM, discussing how to improve the algorithm is not feasible. However, some general ideas can be suggested:

- Introducing randomness during DH operations can increase the difficulty for an attacker to deduce sensitive information. Incorporating random values during the execution of DH operations is an example of this practice.
- Some algorithms and hardware implementations come with specific countermeasures for SPA. These measures are designed to mask power variations during cryptographic operations, complicating the extraction of information by an attacker.
- If possible, employing secure hardware, such as Hardware Security Modules (HSMs) designed to resist physical attacks, can be an effective defense.

2.8 Implementation Note - PRNG test metods: Section 3.4 on page 6

Description of the Problem: The implementation note states that the pseudo-random number generator (PRNG) has undergone serial tests and has passed them, asserting that the distribution of generated numbers is uniform, and due to its long cycle, it is cryptographically secure. However, the cryptographic security of a PRNG is influenced by various factors, and passing serial tests alone is not sufficient to ensure cryptographic security. There are pseudo-random number generators that can easily pass serial tests but still have significant issues and are not cryptographically secure.

Demonstration of the Vulnerability: To demonstrate the vulnerability of a cryptographic system, let's consider an example where a pseudo-random number generator (PRNG) passes serial tests but lacks cryptographic security. This example will illustrate how the security of a cryptographic system cannot be guaranteed solely by passing serial tests but requires a more in-depth assessment of cryptographic security.

Suppose we have a PRNG that generates sequences of seemingly random and uniform bits during serial tests. However, this PRNG is designed naively and predictably. For instance, it could be based on a simple mathematical operation like the sum of consecutive numbers modulo a fixed value:

$$X_n = (X_{n-1} + X_{n-2}) \mod M$$

In this case, even if the PRNG passes serial tests, it has significant vulnerabilities from a cryptographic perspective.
1. **Predictability:** Given a sufficient number of generated bits, an attacker might deduce the complete sequence, compromising the system's security.
2. **Lack of Entropy:** The PRNG lacks entropy and doesn't incorporate any real randomness, making it vulnerable to more sophisticated cryptographic attacks.
In this context, even if the PRNG passes serial tests, the cryptographic system as a whole becomes vulnerable due to the intrinsic weaknesses of the pseudo-random number generator.
This reasoning can also be applied to SuperSAM. A thorough analysis of the error requires studying the implementation, which will be addressed later. For now, the aim was to demonstrate that the claim in the implementation note is incorrect because, without considering other types of tests or adherence to standards, it is not possible to assert that a PRNG is cryptographically secure solely based on passing serial tests.

Proposed Improvement To demonstrate that a PRNG is cryptographically secure, it is necessary for the random number generation system to undergo comprehensive testing from all perspectives. A good idea would be to perform tests suggested by the National Institute of Standards and Technology (NIST), which has published several articles proposing statistical tests, such as `NIST SP 800-22 Rev. 1a` (link to document: Link to NIST article), providing a series of statistical tests to be conducted on random and pseudo-random number generators used in cryptographic applications.
In any case, statistical tests alone cannot certify a generator as appropriate and secure. Hence, another idea is to conduct a thorough cryptanalysis even after performing and passing statistical tests.

2.9 Code - Seed = 0, main.c/rand.c

Description of the Problem: In the `main.c` file, the `init_lfsrs(rngSeed1, rngSeed2);` function is invoked after declaring and initializing the variables `unsigned int rngSeed1 = 0;` and `unsigned int rngSeed2 = 0;`. The `init_lfsrs(rngSeed1, rngSeed2)` function is implemented in the `rand.c` file and is responsible for initializing the values of the `lfsr32` and `lfsr31` registers to zero.
Subsequently, these registers are used in the `uint8_t getRand(void)` function. Since both registers are initialized to zero, the random number generator will consistently return zero. This behavior makes the `getRand` function no longer a random number generator but rather a deterministic function that always produces the same output (zero). Having this function always output 0 poses significant security issues. In fact, this function is used in `masked_combined.c`, which will not function correctly and may lead to potential DPA attacks.

Demonstration of Vulnerability: When seeds are initialized to zero, registers `lfsr32` and `lfsr31` are also initialized to zero. In both cases where the feedback is 0, the function will execute the operation `retrand ^= (uint volatile)POLY_MASK_32;`. However, this operation only modifies the value of `retrand`, not the LFSR registers. Consequently, the value returned in the end will be zero, and this value will never change from its initial assignment.
Since the `getRand()` function consistently returns zero when the seeds are set to 0, the `MaskArray` function (contained in `masked_combined.c`) will consistently generate the same values for the shares. This compromises the main function of masking, which aims to introduce randomness into the data to prevent attacks where it would be possible to deduce the secret key.
Masking is primarily designed to protect against Differential Power Analysis (DPA) attacks. When adequate masking is lacking, it opens the possibility of successfully conducting attacks against AES algorithms, as the energy consumption of devices performing AES depends on the processed data. An attacker who can measure energy consumption can deduce information about the data and secret key used by AES.
Numerous online examples demonstrate that in the absence of effective masking implemented in an algorithm using AES, it is possible to deduce the secret key through the analysis of inputs and traces.
The lack of randomness in the shares introduced by the constant return of zero by `getRand()` makes the system significantly more vulnerable to such attacks, as correlations between power variations and secret operations become more evident and predictable.

Proposed Improvement In order for the code to function correctly and address the issues outlined in the paragraph above, it is necessary to initialize the `void init_lfsrs(uint seed1, uint seed2)` function with values other than 0. Therefore, it is crucial to find values that ensure a different sequence of random numbers with each execution.
This goal can be achieved in various ways; one approach could involve utilizing the system clock, as demonstrated below:

```
time_t t;
time(&t); # include <time.h>
init_lfsrs((uint)t, (uint)(t >> 16));
```

This initialization sets two distinct seeds based on the current time, providing a source of pseudo-random numbers and helping protect against Differential Power Analysis (DPA) attacks.
Another solution could involve using the `rand.h` library to obtain two values to be used as initial seeds.

2.10 Code - 9 Rounds in the Encrypt Function Instead of 10 in `masked_combined.c`

Problem Description: In the file `masked_combined.c`, it can be observed in the Encrypt function that AES-128 is performed with 9 rounds instead of the standard 10. This deviates from the AES standard specification, which is designed with a specific number of rounds to ensure adequate security. For AES-128, the specification mandates 10 rounds, a carefully chosen number to ensure robust cryptographic security.

Demonstration as a Vulnerability: Each round in the AES algorithm involves a series of complex mathematical operations, including substitutions, permutations, and linear transformations, adding an additional layer of complexity to the algorithm. The use of a specific number of rounds, such as the standard 10 rounds in AES-128, has been carefully selected to ensure resistance against various cryptographic attacks.
Altering the number of rounds from what is specified in the standard (available here: AES-128) can be dangerous for several reasons. First and foremost, the choice of 10 rounds was made considering a wide range of cryptanalysis techniques, ensuring robust security. Reducing this number opens the possibility of introducing vulnerabilities that could be exploited by specific types of attacks.
A crucial aspect to consider is that adhering to standards is essential to ensure a clear understanding of vulnerabilities and strengths of the algorithm. Conforming to standards means that the cryptographic community has examined and understood the security implications of the algorithm with that specific number of rounds. Non-compliance with the standard could lead to uncertainties about the security of the algorithm, as it is challenging to precisely predict new vulnerabilities that might arise from non-standard modifications.

Proposed Improvement: The decision to execute AES-128 with 10 rounds, in accordance with established standards, represents the optimal choice to ensure a balanced compromise between performance and security. This standard is the outcome of a meticulous design and cryptographic evaluation phase conducted by the cybersecurity expert community.
Furthermore, the standard implementation with 10 rounds provides an optimal balance between security and performance. The choice of this number of rounds has been carefully considered to deliver the right degree of security without excessively compromising the algorithm's performance.
Moreover, adhering to standards is a best practice in designing and implementing cryptographic algorithms. Following official specifications reduces the risk of unforeseen vulnerabilities and simplifies security management, contributing to ensuring consistency and effectiveness in implemented security measures.

2.11 Code - Key Selection in `main.c`

Problem Description: The key chosen in `main.c` is as follows:

```
uint8_t key[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

This does not represent an optimal key choice, as it is extremely simple and predictable, making the encryption susceptible to various attacks.

Demonstration of Vulnerabilities: The selection of a robust key is crucial for the security of encryption algorithms. In this case, the above key has several vulnerabilities compromising its security:

- The first five bytes of the key follow an incremental sequence, making it easy for an attacker to predict or deduce part of the key.
- The key lacks entropy, as a significant portion of the bytes are set to zero.
- The repetition of bytes (0x00) in the key reduces the diversity of values, increasing vulnerability to certain types of attacks.

The simplistic structure of the key makes it easily susceptible to brute-force and dictionary attacks, which are often highly effective when the chosen key is simple.

Proposed Improvement: To enhance the algorithm's security, it is crucial to choose a more complex key. In the C language, one can generate a random key using functions provided by the standard library or specialized cryptographic libraries. An example of generating a random key in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));

    uint8_t key[16];
    for (int i = 0; i < 16; ++i) {
        key[i] = rand() % 256;
    }
    return 0;
}
```

2.12 Code - Lack of Fault Attack Protection in AES Implementation in `masked_combined.c`

Problem Description: The current implementation of the AES algorithm lacks mechanisms to protect against fault attacks, which allow for memory corruption, putting the confidentiality of processed information at risk.

Demonstration of Vulnerabilities: It is plausible that an attacker could gain physical access to a device, especially in the case of tokens. This scenario makes it possible to execute fault attacks on AES, using various techniques to compromise the security of the cryptographic system.
A Bit Flipping attack represents a strategy in which the attacker tries to flip specific bits in the blocks of ciphertext generated by a particular encryption. This controlled manipulation can induce variations in the decrypted data, jeopardizing the correctness of the decryption process and compromising data integrity.
The attacker could also opt for an Initialization Vector (IV) Manipulation attack, attempting to alter the IV used in CBC mode. This type of attack would influence the propagation of fault effects along the block chain. Altering the IV can have significant consequences on decryption, leading to unexpected plaintext results.
Another strategy could be a Key Perturbation attack, where the attacker seeks to alter the key used during encryption or decryption operations. Injecting faults into the key can result in manipulations in the encrypted or decrypted data, undermining the system's confidentiality.

Proposed Improvement: To protect an AES implementation from fault attacks, specific measures ensuring the integrity and security of the algorithm in scenarios of possible memory manipulation are necessary. A recommended solution is the implementation of AES with authenticated encryption, such as GCM (Galois/Counter Mode) or CCM (Counter with CBC-MAC), as these modes offer both confidentiality and data authentication in a single operation.
Using AES with authenticated encryption adds an essential layer of security, ensuring that data is not only encrypted but also authenticated, thereby reducing vulnerability to fault attacks. Integrated authentication prevents the possibility of an attacker manipulating data in memory without being detected.