Securing Hardware Communication using Encryption and Attestation

Julian Pritzi
julian.pritzi@tum.de
Technical University of Munich
Munich, Germany

Abstract

This paper presents the SPDM-Broker, a novel hardware component that builds upon the existing opentitan platform as a Root of Trust (RoT), and acts as a gateway between a device and an insecure channel. The SPDM-Broker establishes a secure session over the channel to protect any exchanged messages. This paper first compares two protocols with respect to their security and functionality, to determine which one should be used for establishing the secure session. This comparison includes verifying the secrecy and authentication properties of one of the protocols using both the symbolic and computational model. Based on this evaluation we conclude that the SPDM protocol will be used by the SPDM-Broker, because we find it more flexible and future-proof. Given this target protocol, the paper presents a design for the SPDM-Broker, including a partial implementation and preliminary microbenchmarks to estimate the feasibility of the design. The benchmarking results show that symmetric encryption is a significant bottleneck, due to lacking accelerator support for the AES GCM mode in opentitan.

1 Introduction

In this paper, we propose a design to prevent attacks on an insecure channel between two low-level devices through the introduction of a transparent hardware module. This module acts as a gateway between a device and an insecure channel and establishes a secure session over the channel to protect any exchanged messages.

In § 2 we cover the necessary information to understand the remainder of the paper. Then in § 3 we evaluate what protocol we will use to establish a secure connection. For this, we consider a minimal custom protocol as well as a more complex industry standard. In § 4 we present our design for the hardware module, that makes use of the selected protocol. Further, § 5 and § 6 discuss the partial implementation and preliminary microbenchmarks of this design. Lastly, we summarize and present our conclusions in § 8.

2 Background

2.1 Opentitan

Opentitan [2] includes a combination of hardware and software components that compose a RoT platform. Opentitan provides a general set of tools to combine the different components into top-level platforms. The project currently has one integrated top-level platform that can already be used called Earl Grey. The Earl Grey platform is a predefined combination of IP cores connected through a TileLink Uncached Lightweight (TLUL) [3] bus, integrating the single-core RV32 Ibex processor [1] as its main CPU.The platform also includes accelerators for different tasks. Important for this paper are the cryptographic accelerators and the custom cryptographic coprocessor called Opentitan Big Number Accelerator (OTBN). The next subsections describe these components in more detail.

2.1.1 Cryptographic Accelerators

The Earl Grey platform contains a number of accelerators:

The **aes** core is an accelerator for the symmetric aes encryption and decryption. It supports the Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) modes and can operate on key lengths of 128, 192 and 256bits.

The **hmac** core is used for message authentication code (MAC) generation. It supports 256bit keys and uses SHA2 for computing the hash internally. The core can optionally be configured to only compute the SHA2 hash digest.

The **csrng** core is a cryptographically secure random number generator. It can generate random data from an initial seed or from an entropy source which allows for true random number generation. The core is designed to be used by hardware and software alike, through the entropy distribution network hardware has access, while software uses a special protocol over the csrng cores registers to communicate.

2.1.2 OTBN

The Opentitan Big Number accelerator (OTBN) is a cryptographic coprocessor specialized for big number computations that exceed normal-sized registers. For this, the core includes 32 x 256bit data registers and operates using regular 32bit registers for control flow. The wide registers are especially useful for cryptographic computations like asymmetric encryption where key sizes are large. OTBN is operated by a custom assembly code that is an adaptation of the RV32 ISA. OTBN implements a subset of the RV32 instructions and a set of additional instructions for controlling the wide data registers. The OTBN has its own data and code buffer which have to be programmed during run time, and is completely controlled by the register interface that is exposed to the main CPU.

2.2 Protocol Verification

As part of this work, different protocol verification strategies are employed to analyze and guarantee the security properties of a protocol. These properties are automatically proven using Tamarin § 2.2.1 and CryptoVerif § 2.2.2. We assume the attacker has full control over the exchanged messages and can read, modify or delete them. Because such an attacker is able to trivially perform a DoS attack by simply dropping all messages, DoS attacks are not further considered.

For these tools to reason about the protocol and its properties it is necessary to translate the protocol into a mathematical model. For protocol verification, there are two predominant mathematical models to target: The symbolic model and the computational model.

The symbolic model is more abstract, it assumes messages to consist of atomic terms and that cryptographic algorithms are unbreakable with no conceivable side effects. One particular implication of this is that encryption hides the length of the plaintext, while this is generally not true in the real world, in the symbolic model it is not possible to determine the size of individual message components. Using these strong assumptions it is possible to prove whether a protocol satisfies a specific property or not.

The computational model on the other hand works with bitstrings as its base component and uses functions on these bitstrings to model cryptographic operations. Length information is retained in this model, which allows for a more realistic analysis considering an attacker's computational feasibility to brute force certain inputs. The proofs of this model show properties of the protocol that are upheld with a reasonably high probability. The length and entropy information is used to compute the probability of a successful attack and the estimated time for an attacker to violate one of the protocol's security properties.

2.2.1 Tamarin

Tamarin [5] is an automatic prover in the symbolic model. Tamarin represents all the possible states of a protocol as a multiset of facts. State transitions are modeled as multiset rewriting rules. The initial proof state is the empty set from which Tamarin reaches different states using the multiset rewriting rules, this generates a trace of rewriting rules that are executed to reach any protocol state from the initial state. Proofs in Tamarin are constructed on these traces by arguing about facts and timestamps using first-order logic formulas.

2.2.2 CryptoVerif

CryptoVerif [8] expects the protocol modeled in the form of a game as an input and generates secrecy and authenticity proofs consisting of a sequence of games. The sequence shows a transformation of the input game to a known game, this is done by modifying the input game step by step, with only a negligible probability that the games differ. This produces a chain of modifications and can be used to derive attack probability and estimated time to attack the protocol. Because of the computational model CryptoVerif expects all values to be typed with a specific length, the protocol is then represented as a set of oracles that the attacker can use to cheaply perform otherwise expensive computations. Modifications are made according to the equations, that define the relation of functions and variables.

2.3 SPDM Protocol

The Security Protocol and Data Model (SPDM) [6] is a general protocol designed to enable authentication, attestation, and key exchange. It is developed by the Distributed Management Task Force (DMTF) standards organization The SPDM protocol is already part of Intel TDX 2.0 which supports trusted execution environments (TEE) for device I/O.

The SPDM protocol specifies a large set of message types that can be used in the SPDM protocol flow, including measurement requests, heartbeat messages, and key provisioning functionality but this paper focuses on the initial authentication and key exchange flow.

Figure 1 shows the initial messages exchanged in the SPDM protocol. The capabilities exchange is used to determine what set of features are supported by each communication partner. This allows for more restricted devices to only implement the minimum subset of features required to function properly. The negotiate-algorithms exchange serves a similar purpose in negotiating the cryptographic algorithms like hashing, encryption, and signing used for the remainder of the protocol exchange. Additionally, this also allows the possible deprecation of cryptographic primitives. As seen in the past with hashing algorithms like SHA1 [13] or encryption like DES [7,10,11] becoming insecure, it is necessary for a future-proof protocol to allow selecting the cryptographic algorithms to be used.

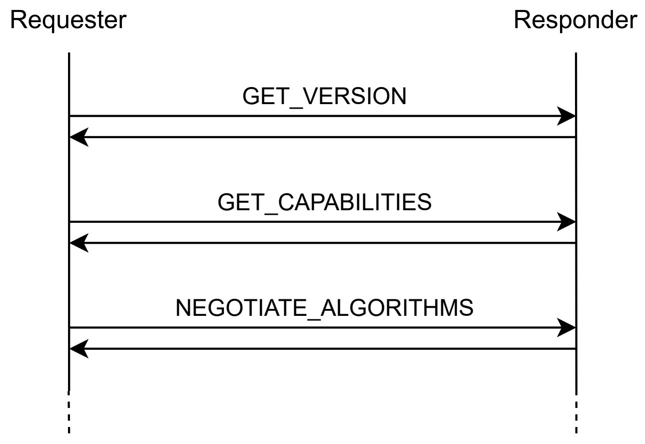

Figure 1: SPDM: Start of the protocol.

Figure 2 shows the responder authentication process that may follow after the initial messages. The requester first retrieves the digests of the certificates that the responder makes use of. If any of the digests are unknown to the requester, they can be resolved by requesting the full corresponding certificates from the responder. This design allows for less data to be exchanged if the requester and responder have previously already performed an authentication, as the requester may store retrieved certificates. Once all certificates are exchanged and verified the requester can challenge the responder to sign a random nonce. In the challenge reply the responder may include additional measurement information like the firmware version of the device or checksums if applicable.

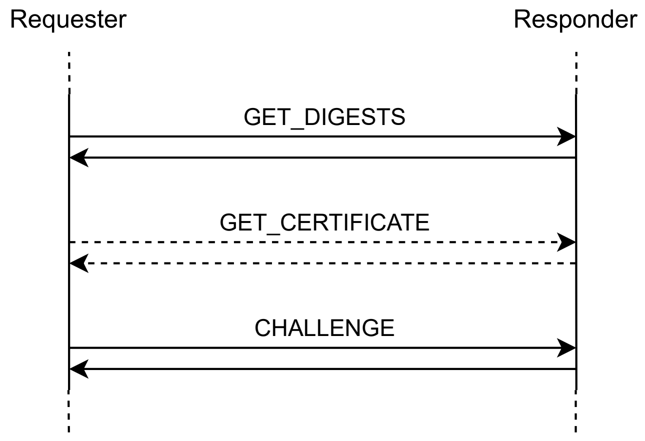

Figure 2: SPDM: Authentication of the Responder.

Figure 3 shows the symmetric key exchange and setup of a secure session between the requester and responder. The actual data sent during the key exchange is dependent on the key exchange algorithm negotiated during the messages in Figure 1. After the exchange, the communication is performed using authenticated encryption with associated data (AEAD). If supported the requester and responder can achieve mutual authentication, this is done similarly to the authentication of the responder to the requester but with the roles reversed. Finally, the finish message is used to complete the handshake between the requester and responder, after which application data may be exchanged inside the secure session.

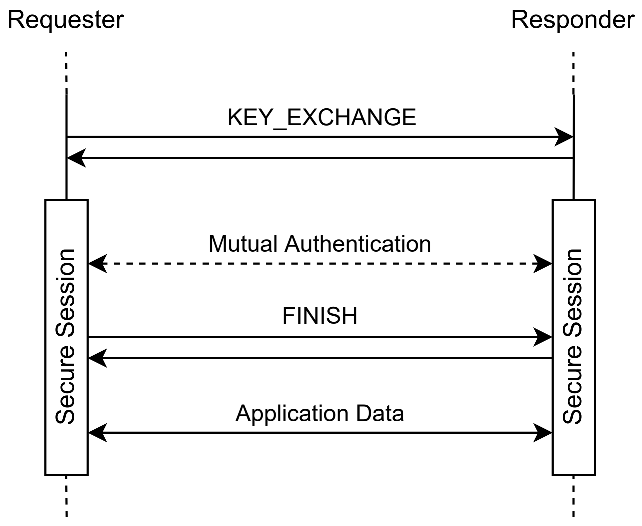

Figure 3: SPDM: Establishing a secure session.

3 Protocol Evaluation

3.1 Basic Protocol

Figure 4 shows a basic protocol that performs mutual authentication and a symmetric key exchange to establish a secure session. The following notation is used in the diagram: $\{x\}_{agent}$ denotes the transmission of message x that is secured by a signature of *agent*. Further n is used for random nonces and g^x is used for representing elements in a Galois field for elliptic curve Diffie Hellman key exchanges or a modulo group in the case of Diffie Hellman key exchanges. Every message received is verified by the receiver and the protocol flow is aborted in case of invalid data.

To evaluate the security properties of this protocol it is modeled and analyzed using both Tamarin and CryptoVerif. In both cases, the two main properties to prove are the secrecy of the exchanged key and the authenticity of the communication partner. More formally these two properties can be expressed as follows:

- **Secrecy** If the software and the device reach the last step of the protocol and successfully performed a symmetric key exchange then there exists no way for an attacker to know this symmetric key.
- **Authenticity** This includes two parts:
  - The software established a key with an authentic device.
  - The device established a key with an authentic software.

  Both of these conditions can be formalized by induction, ie. if the software responds to a message then it has to follow that the authentic device previously sent that message.

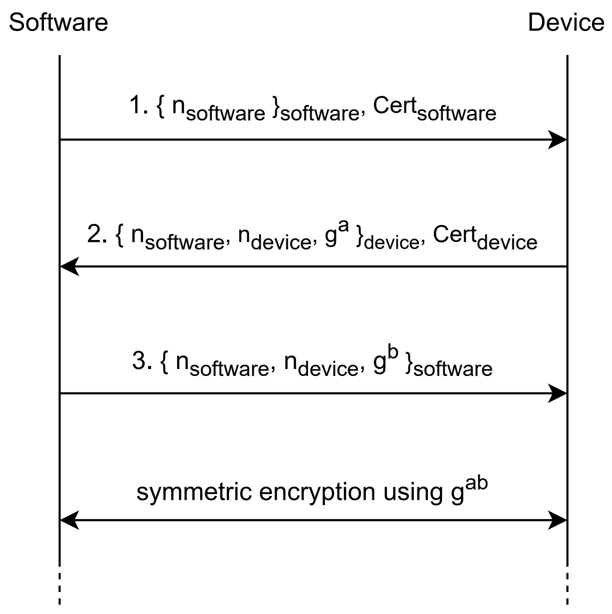
Software
Device
1. { n_software }_software, Cert_software
2. { n_software, n_device, g^a }_device, Cert_device
3. { n_software, n_device, g^b }_software
symmetric encryption using g^ab

Figure 4: Basic attestation and key exchange protocol.

3.1.1 Symbolic Analysis

The symbolic analysis of the protocol in Figure 4 is done using the Tamarin prover. For this, the protocol is modeled using multiset rewriting rules, each rule consisting of a set of input facts, output facts that are generated if the rule is executed, and a set of action facts that are created at the time point of the rules execution. The translation of the protocols messages in most cases is performed by listing the expected memory state of the agent as well as the expected message as inputs and the resulting modification to the memory as well as the message reply as output. Additionally, action facts are used to add metadata to rule executions that can later be used as part of the first-order logic formulas.

The formulas that are proven by the symbolic analysis are as follows:

$$\nexists\, key\ \#i\ \#j\,.\ (\mathrm{PrivKey}(key)@\#i \wedge \mathrm{K}(key)@\#j)$$

Logic formula 1. All the private keys, that are created and used in the protocol, in particular, the software's and device's secrets, cannot be obtained by an attacker.

$$\begin{aligned}&\forall\, software\ device\ key\ \#i\ \#j\,.\\ &\quad((\mathrm{SoftwareDone}(software, device, key)@\#i\\ &\quad\wedge \mathrm{DeviceDone}(device, software, key))@\#j\\ &\qquad\qquad\Longrightarrow (\nexists\, \#k\,.\ \mathrm{K}(key)@\#k))\end{aligned}$$

Logic formula 2. The symmetric key, established during the protocol is only known by the software and the device.

$$\begin{gathered}\forall\, software\ device\ keyh\ \#i\ \#j\,.\\ (\mathrm{SoftwareTrust}(software, device, keyh)@\#i\\ \Longrightarrow\\ (\exists\, \#k\,.\ (\#k < \#i \wedge \mathrm{DeviceReply}(device, software, keyh)@\#k)))\\ \wedge\\ \forall\, software\ device\ key\ \#i\ \#j\,.\\ ((\mathrm{DeviceDone}(device, software, key)@\#j\\ \Longrightarrow\\ (\exists\, \#k\,.\ (\#k < \#j \wedge \mathrm{SoftwareDone}(device, software, key)@\#k))))\end{gathered}$$

Logic formula 3. The symmetric key is established between two authenticated partners, this involves the following inductive property: If one agent reacts to a message it received instead of discarding it, then this message was sent by the authentic partner.

3.1.2 Improving the Symbolic Analysis using an Oracle

In addition to the protocol model and the logic formulas the Tamarin files also include a minimal Python script acting as an oracle. This script is invoked during the automated rewriting steps by Tamarin whenever a decision has to be made. Tamarin tries to generate the action facts contained in the logic formulas using a backward search by default. For a given action fact this involves identifying all the set of facts it depends on to be created. As there are many different ways to potentially create any given action fact this list of dependencies can grow large, so the oracle is used to provide the order in which these dependencies should be resolved. Depending on the order in which the dependencies are resolved the same proof can terminate or fail to terminate. Therefore finding the correct way to order fact dependencies is crucial to minimize the steps required for the proof.

Because Tamarin by default tries to find a contradiction to the provided logic formulas, it is beneficial to resolve dependencies that are likely impossible to obtain earlier. For example, if some action fact depends on $\mathrm{K}(ca_{priv})$, $\mathrm{SoftwareDone}(device, software, key)$ and $\mathrm{K}(ca_{pub})$, it is likely beneficial to try and resolve $\mathrm{K}(ca_{priv})$ as we predict that the attacker is not able to obtain the private key of the certificate authority. Resolving this dependency early should lead to a contradiction, allowing Tamarin to conclude that the action fact is not obtainable, eliminating the need to evaluate the other dependencies.

In conjunction with evaluating likely impossible dependencies earlier, it is also beneficial to evaluate simpler terms earlier than complex terms. This is because if an action fact depends on a complex combination of simpler facts, it is likely that if this combination is not obtainable then it comes from the fact that one or more of the facts used in the combination is not obtainable. For example, if some action fact depends on $\mathrm{K}(ca_{priv})$ and $\mathrm{K}(g\hat{}(ca_{priv} * inv(device_{priv}) * x))$ then it takes fewer steps to proof that $\mathrm{K}(ca_{priv})$ is not obtainable as it likely depends on fewer facts itself.

Both these approaches are implemented in the python oracle to minimize the steps required for generating a proof.

3.1.3 Computational Analysis

The computational analysis of the protocol in Figure 4 is done using the CryptoVerif prover. For this, the protocol is modeled as oracles that can be invoked by an attacker to cheaply perform computations. The input file for CryptoVerif is divided into different sections, the first section declares the parameters and types used in the model, these types have length annotations that are later used to determine the computational feasibility of breaking one of the security properties.

The next section defines the signature, hash, and Diffie Hellman functions and the assumptions that are made about them. The Diffie Hellman functions are defined with a negligible probability of both single and repeated exponentiation collisions, ie. we assume it is very unlikely that for arbitrary $x, y : g^x = g^y$ and it is very unlikely that for arbitrary $x, y, a, b : g^{x \cdot y} = g^{a \cdot b}$. Additionally, we make the computational Diffie–Hellman (CDH) assumption which states that given g, g^a, g^b it is computationally infeasible to compute $g^{a \cdot b}$. For the signature functions, we assume Universal Forgery under Chosen-Message Attack (UF-CMA) has a negligible probability, ie. it is very unlikely that an attacker is able to forge a signature or intentionally create an altered message for which the original signature is valid.

The last section describes the oracles that the attacker can make use of. The oracles in this case are either software or device responding to a message, so they act as functions to the attacker with a message as input and, if the input was valid, the response message is the output. Using this the attacker attempts to obtain information about the secret key or invalidate the inductive property used to represent authentication.

CryptoVerif is then able to automatically prove all these properties and compute associated probabilities.

3.2 Conclusion

As shown in the previous sections the protocol shown in Figure 4 upholds the basic security guarantees we analyzed in both the symbolic and the computational model. Yet when compared to SPDM the protocol is much more rigid, especially with respect to future-proofing. Primarily the ability to negotiate capabilities and the cryptographic algorithms used are benefits of the SPDM protocol. The advantages of the protocol in Figure 4 in comparison are its simplicity and minimality. This allows for easier analysis as performed in this paper. The SPDM protocol is also analyzed using the Tamarin tool in [9] and has a real-world use case in the form of the Intel TEE I/O. For these reasons, we will target the SPDM protocol for the rest of this paper.

4 Design

Following the evaluation we decide to target a design that uses the SPDM protocol. Therefore this section introduces a design for a hardware module that can be used transparently as part of a communication channel to establish an SPDM secure session to protect exchanged messages. The design builds upon the opentitan Earl Grey platform (§ 2.1) as we will explain in this section. In § 5 we will then introduce the first steps towards implementing this design as well as the implementation of some microbenchmarks, designed to help evaluate the feasibility of realizing this design using opentitan. The results of these preliminary tests will be discussed in § 6.

Figure 5 shows the initial design of the SPDM-Broker, the hardware module that is effectively an Earl Grey platform extended with two custom communication modules for interfacing with the local chip and the bus or network where the SPDM secure session will be established. The outward-facing interface of these modules is highly dependent on the connection of the local chip and bus and thus not fully designed yet. The current design focuses on establishing a secure connection with the bus or network on the public communication module interface. This session can then be used to either encapsulate the data sent to and from the chip or to configure the SPDM-Broker.

The design leverages existing components of the Earl Grey platform (marked green in Figure 5) to provide the functionality of the SPDM-Broker. When receiving a request to establish an SPDM-Session on the public interface the information gets forwarded to the Ibex core that uses the crypto accelerators and local memory to generate the replies. Once the secure session is set up, the SPDM-Broker forwards the requests to the local chip and encapsulates its response before forwarding it again on the public interface.

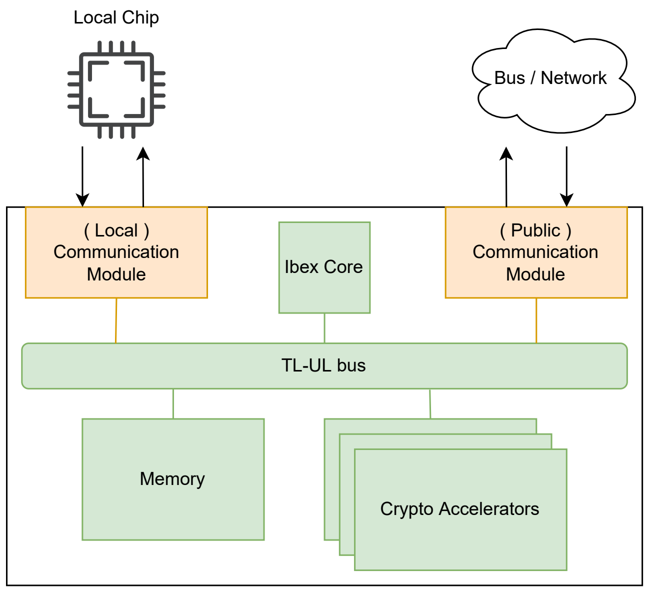

Figure 5: Initial SPDM-Broker design.

Using opentitan as a basis allows bootstrapping trust into the component as it is designed as a silicon RoT. Further, the Earl Grey platform contains a number of accelerators that can be used for accelerating the setup of the SPDM secure session. Because the SPDM protocol is so flexible with its cryptographic algorithm requirements, this paper takes into account the possible interoperability of the SPDM-Broker with intel TEE-I/O instead. The TDX Connect TEE-IO Device Guide [14] lists requirements for the SPDM protocol to be compatible with TEE-I/O. In particular, this paper considered the following requirements for cryptographic algorithms:

- One of the following hashing algorithms is required SHA256, SHA384. Opentitan's hmac accelerator is able to perform SHA256 computations.
- TEE-I/O requires one of the following asymmetric cryptography algorithms: RSA3072, ECDSA NIST P256 or ECDSA NIST P384 as well as one of these curves secp256r1 or secp384r1. Opentitan does not have a dedicated accelerator for either of those curves but it has existing programs running on the OTBN accelerator that implement the necessary functionality.
- The AES-256-GCM AEAD cipher is also required. Opentitan's aes accelerator does not support the AES GCM mode but it has support for the AES CTR mode with 256bit keys. Figure 8 shows the aes mode that is supported by opentitan's aes accelerator and Figure 7 shows the GCM mode as required by TEE-I/O. The CTR mode can be slightly modified and extended by the authentication tag computation to effectively perform as an AES GCM encryption. So the aes accelerator can be used for part of the computation while the remainder has to be done in software.

The remainder of the required functionality can be implemented as software running on the Ibex Core using a custom opentitan interface library written in Rust.

4.0.1 Opentitan-rs

The opentitan project already includes files, written in C, to interface with the hardware, but for this project we want to use the Rust language to increase security and usability. Rust's memory safety alone prevents a large number of bugs that can occur when implementing such a security-critical component. Additionally, the zero-cost abstractions allow for designing interfaces that make it impossible to represent illegal states. As an example: Using implicitly sized Rust slices instead of C pointers with separate length variables. We believe that this inability to represent illegal states further reduces the surface for errors and simultaneously increases usability by making use of more expressive higher-level abstractions.

5 Implementation

5.1 Opentitan-rs

The opentitan-rs template project is the basis for the Rust library running on opentitan, it is structured as a Rust workspace. Therefore the main workspace folder has subfolders for different parts of the project. The folders are:

- `opentitan-lib` The main contribution of the opentitan-rs template, the opentian-lib, is responsible for ensuring a proper runtime setup as well as providing interface functions to the user app. This folder contains all the setup and interface code as well as the linker script and memory layout descriptions. The library has a set of feature flags to enable or disable optional functionality like `atomic_emulation`, `verbose_logging` or `alloc` crate support.
- `opentitan` This folder is vendored in from the official opentitan repository. This serves as the target architecture of the compiled project binary. This folder is used as a reference for autogenerating code that depends on the hardware as well as for seamless simulation of the current project.
- `opentitan-macros` This folder is an auxiliary folder to the opentitan-lib and is required by Rust for proc macro support. The opentitan-rs project currently only exposes the `entry` proc macro to the app developer. This macro is used as an attribute to annotate the entry function of the app and implicitly ensures the function is set up correctly so that the opentitan-lib is able to call it.

- `app` This project is intended to be modified and programmed by the users of the opentitan-rs template. The main source file contains imports and crate-level attributes that are necessary for the opentitan-lib to seamlessly integrate. Ideally, Rust will allow for custom inner attributes to be attached at crate-level in the future, this would allow for omitting the app project and simply defining a custom proc macro that can be used in any project that imports the opentitan-lib.

5.1.1 Simulating the Project on Verilator

With the inclusion of the opentitan project as a subfolder it is possible to easily simulate the opentian-rs project using Verilator. For this to work all the tooling for the opentitan project has to be set up correctly and the project has to be built at least once. After that running `cargo run` inside the opentitan-rs workspace folder automatically retrieves the relevant files from the opentitan repository and converts the compiled elf file to a valid input for the Verilator simulation. During this process the folder `target/verilator` is populated with useful debugging information, including a disassembly of the binary, a trace of the simulated CPU instructions, and log files for the SPI, UART, and USB outputs of the simulated platform.

5.1.2 Automatic Register Generation

In order to provide device interface functions the opentitan-lib has to be aware of the memory-mapped I/O (MMIO) with the devices of the platform. For this, the opentitan project provides description files that are used by the opentitan project itself to generate the actual IP cores. The first description file is `hw/top_earlgrey/data/top_earlgrey.hjson`, which contains a description of all cores connected to the TL-UL bus and their mapping in the memory, this file can be used to determine the base address for the register-based MMIO. Second, every core has a description of its registers their layout, content, and documentation, for example for the aes core this information can be found in `hw/ip/aes/data/aes.hjson`. To stay as flexible and updated as possible, the opentitan-lib parses these files at compilation and translates the hardware description into a Rust structure that can then be used by other driver code of the opentitan-lib or the application running on top. The way this is implemented is using Rust proc macros, which receive the path to the file as an input and then generate valid Rust syntax using the quote crate. This approach already works and is heavily extensible for the future by improving the parsing and generation capabilities.

5.1.3 Heap Allocation

To allow more flexible development for the app, the opentitan-lib contains an optional implementation for heap allocation. This is done using the linked_list_allocator library [12] which has to be modified to work in a single-core environment. After implementing the global allocator the alloc library can be used in the app and everything just works out of the box.

5.1.4 Atomic Emulation

The Ibex core does not support atomic operations, yet many libraries require atomics for synchronization. This should not be required on a single-core machine as there can not be any concurrent access. But to facilitate the easy integration of libraries that depend on atomics the opentitan-lib supports emulation of atomics as a optional feature. This is implemented using a custom exception handler that catches the illegal instruction exception thrown by the CPU once a atomic instruction is encountered and then simulating the outcome. Because the platform is single core and interrupts are disabled during the simulation, the end effect is effectively the same as if the Ibex core supported atomic operations.

5.1.5 Testing functionality

The opentitan-lib also exposes a basic testing framework to the application. Functions can be annotated with the `#[test_case]` attribute to mark them as tests, they will then be simulated and tested when executing `cargo test` inside the main workspace folder. To achieve this the `entry` proc macro used conditional compilation to determine whether the opentitan-lib starts the annotated entry function or a custom, generated testing harness main function that tests all the annotated functions.

5.2 Microbenchmarks

The microbenchmarks that are evaluating the platform are implemented on top of the opentitan-rs template making use of the functionality provided by the opentitan-lib library. As briefly explained in § 4 custom aes functionality is implemented to estimate the cycles it takes for a GCM encryption that makes use of the aes accelerator when applicable.

6 Evaluation

The theoretical evaluation for using the Earl Grey platform as a basis for the SPDM-Broker is already discussed in § 4 where the possibilities to fulfill the requirements of TEE-I/0 are explained. This section looks at the data obtained from the microbenchmarks to provide a preliminary estimate of potential bottlenecks.

The first microbenchmark is for computing the digest of a minimal X.509 certificate that may be used as part of the SPDM protocol. For this, a simple test certificate is generated which had a resulting size of 560 bytes. These bytes were then hashed using the opentitan hmac accelerator to get a cycle estimate for the digest computation. The result is that

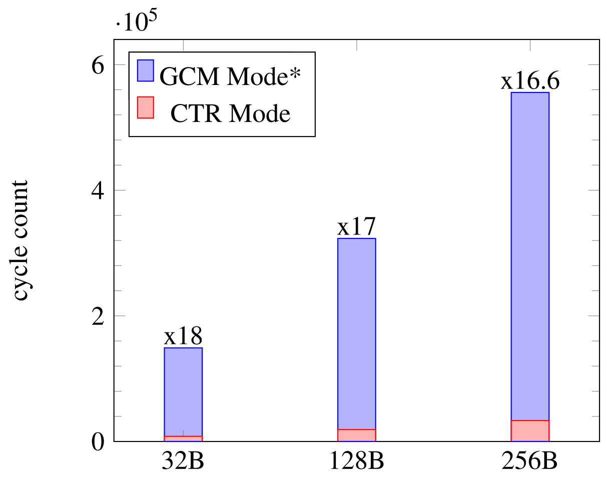

Figure 6: Encryption cycles measured for different number of bytes. The GCM implementation only partially uses the accelerator and mocks the tag computation. The slowdown of using GCM instead of CTR is denoted above the bars.

computing a certificate digest takes 37592 cycles or 376 μs on a 100MHz CPU.

In a second step signing using OTBN and the ECDSA P256 curve is benchmarked, which resulted in the following cycle measurements for signing and verifying a digest: Signing of a digest takes 896517 cycles or 8965 μs on a 100MHz CPU and verification of a digest takes 542334 cycles or 5423 μs on a 100MHz CPU.

Lastly, the mocked GCM implementation is benchmarked for an estimate of the required cycles for encryption and decryption using the accelerator when applicable, the result is presented in Figure 6. As expected the computation of the authentication tag in software significantly slows down the total performance of the encryption. The slowdown slightly decreases with larger data sizes as part of the authentication tag computation stays constant.

7 Related Work

Related work includes the paper on the formal analysis of the SPDM protocol in the symbolic model [9]. The paper presents formal models for the SPDM protocol version 1.2.1 and their formal analysis with regard to the protocol's main security properties. The paper proves the absence of a substantial set of possible attacks. Yet at the time of writing this, there is still no analysis performed in the computational model, which would cover a larger range of attacks.

This paper is also related to Intel TDX, in particular, TEE-I/O as described in [14]. This is because the SPDM protocol is part of TEE-I/O as the protocol used by the device security manager (DSM) to establish a connection with the TEE security manager on the main CPU.

There is also related work happening at Synopsys, as they recently announced their IDE Security IP Module for the TEE Device Interface Security Protocol (TDISP) [4] which is also used by TEE-I/O and potentially has some overlapping functionality with the SPDM-Broker, like AES-GCM encryption.

8 Summary and Conclusion

Formal analysis of the protocol in Figure 4 proves that it upholds the analyzed security protocols. When comparing the protocol with SPDM, SPDM promises to be more future-proof at the cost of increased complexity. For this paper's research, the SPDM protocol is thus further analyzed. This paper introduces a hardware component building on opentitan to realize the SPDM protocol in the form of a transparent broker. This design was evaluated using microbenchmarks, which show that the aes GCM encryption is a significant bottleneck due to the reliance on software for computing the authentication tag. Because encryption is a central part of the SPDM secure session, there likely needs to be a significant improvement to the GCM encryption's performance before implementing the SPDM-Broker design. Yet a more detailed evaluation is required to fully determine the viability of the SPDM-Broker design for real-world use cases, as this paper only provides preliminary results.

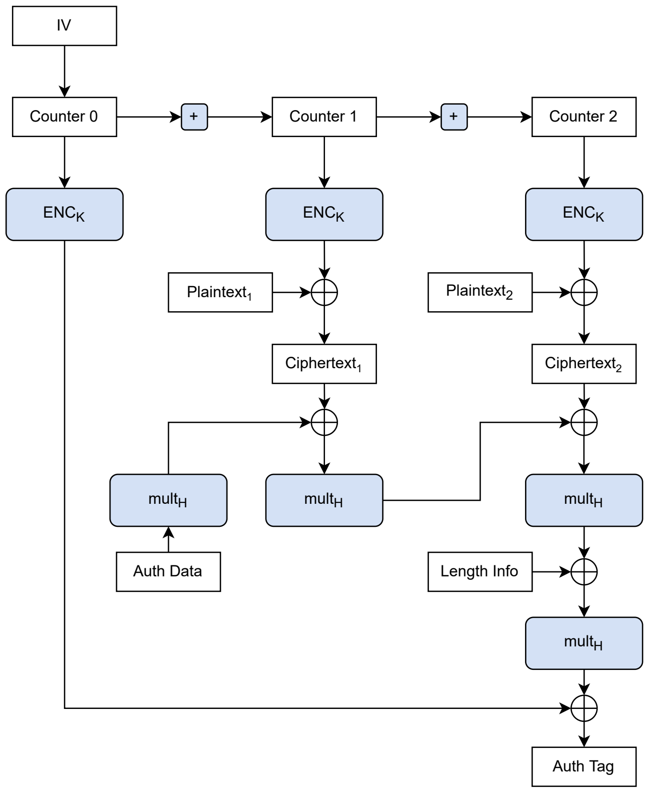

Figure 7: AES encryption in galois counter mode.

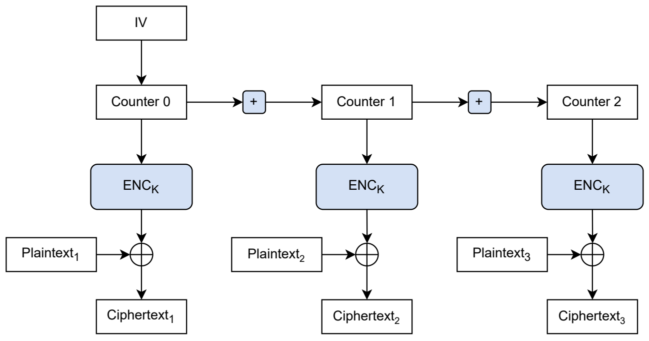

Figure 8: AES encryption in counter mode.

9 Future Work

This paper lays the foundation for future work. For one the opentitan-rs project is still in an early stage, while register description structures can automatically be generated there is still a lot of Rust idiomatic driver code missing. In particular, the library is missing features to properly use accelerators like OTBN, KMAC, or other opentitan cores like the key manager, SPI, and USB cores. The library also currently has no proper support for interrupts.

The majority of the future work is in conjunction with the SPDM-Broker. For one the exact design of the public interfaces, and possible ways to bind the local chip to the SPDM-Broker in such a way that it is possible to detect if it is changed, are all potential topics for future work.

Additionally, there is no actual prototype of the current SPDM-Broker design, so future work includes implementing the current design and testing it and its interoperability with other systems like TEE-I/O.

References

[1] Ibex risc-v core. `https://github.com/lowRISC/ibex`.

[2] Opentitan documentation. `https://opentitan.org/documentation/index.html`.

[3] Sifive tilelink specification. `https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf`.

[4] Synopsys ide security ip module for pci express 6.0. `https://www.synopsys.com/dw/ipdir.php?ds=security-pcie6-ide`.

[5] Tamarin prover. `https://tamarin-prover.github.io`.

[6] Spdm standard, May 2022. `https://www.dmtf.org/standards/spdm`.

[7] Eli Biham and Alex Biryukov. An improvement of davies' attack on des. *Journal of Cryptology*, 10:195–205, 1997.

[8] Bruno Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models*, Paris, France, June 2005.

[9] Cas Cremers, Alexander Dax, and Aurora Naska. Formal analysis of spdm: Security protocol and data model version 1.2. Cryptology ePrint Archive, Paper 2022/1724, 2022. `https://eprint.iacr.org/2022/1724`.

[10] Lars R Knudsen and John Erik Mathiassen. A chosen-plaintext linear attack on des. In *Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10–12, 2000 Proceedings 7*, pages 262–272. Springer, 2001.

[11] Hirokazu Kodera, Masao Yanagisawa, and Nozomu Togawa. Scan-based attack against des cryptosystems using scan signatures. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 599–602. IEEE, 2012.

[12] Rust-Osdev. Rust-osdev/linked-list-allocator. `https://github.com/rust-osdev/linked-list-allocator`.

[13] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on sha1, 2005.

[14] Jiewen Yao. Intel® trust domain extensions. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html`.