Transforming byzantine faults using a trusted execution environment

Mads Frederik Madsen, Mikkel Gaub, Malthe Ettrup Kirkbro, and Søren Debois
Department of Computer Science
IT University of Copenhagen
{mfrm,mikg,maek,debois}@itu.dk

***Abstract*—We present a general transformation of general omission resilient distributed algorithms into byzantine fault ones. The transformation uses the guarantees of integrity and confidentiality provided by a trusted execution environment to implement a byzantine failure detector. Correct processes in a transformed algorithm will operate as if byzantine faulty processes have crashed or their messages were dropped. The transformation adds no additional messages between processes, except for a pre-compute step, and the increase in states of the algorithm is linearly bounded: it is a 1-round, $n = f + 1$ translation, making no assumptions of determinism.**

***Index Terms*—Byzantine faults, translation, TEE, TPM, SGX**

I. Introduction

In the byzantine fault model, we know that $3f+1$ processes are necessary to solve the consensus problem [1]–[3], while $2f + 1$ processes are sufficient to solve the problem in the crash-fault model [3]. By *solve*, we mean with some relaxation of either liveness, asynchrony or determinism, to circumvent FLP impossibility [4]. Using hybrid fault models, where *some* subsystems are assumed to fail only by crashing, new solutions have achieved $2f + 1$ fault tolerance [5]–[10]. The trusted subsystem prevents a process from *equivocating*, the action of sending contradicting messages, which increases the fault tolerance from $3f + 1$ to $2f + 1$ [5,11].

These small trusted subsystems take different forms. Most notable are the attested append-only memory (A2M) identified by Chun et. al. in [5], and the authenticated monotonic counter (TrInc) identified by Levin et. al. in [11]. The solutions using these trusted subsystems make the common assumption that the component, in fact, does not experience byzantine faults, but fails only by crashing.

We show how any algorithm tolerant to general omission faults and unreliable channels can be transformed into a byzantine fault-tolerant algorithm. Note that while the transformation works regardless of synchrony, it does require the underlying algorithm to be resilient to omission faults *also* in the synchronous setting.

The transformation operates by moving the entirety of the algorithm into a Trusted Execution Environment (TEE). We do not assume that the TEE is free from byzantine faults, but rather use the integrity and confidentiality guarantees provided by the TEE. The intuition behind the transformation is that the integrity guarantee simulates a perfect local byzantine fault detector oracle, which can detect only byzantine faults, and can be used to translate any such fault to a crash or a message omission. Meanwhile, the confidentiality guarantee ensures that, given a shared secret between the integrity protected processes, no process can falsely pass itself off as being integrity protected.

The transformation works on both synchronous (but omission resilient) and asynchronous algorithms; transforms both deterministic and randomised algorithms; introduces no overhead on the number of messages past pre-computation; imposes only a small constant overhead on message size; and preserves the algorithm's fault tolerance. To our knowledge, no transformation with these properties currently exists.

The overhead of transformations are traditionally measured in rounds and fault tolerance (e.g. [12]–[15]). E.g. a 4-round, $n = 3f + 1$ transformation converts one round of message send/receives in the algorithm into four and requires that at most $\lfloor \frac{n-1}{3} \rfloor$ processes exhibits faults. (A translated algorithm will have the lower fault tolerance of the transformation and the original.) We say that a transformation is a crash to byzantine transformation if it can transform a crash fault-tolerant algorithm to be byzantine fault-tolerant and that the transformation translates byzantine faults to crash faults if a byzantine fault on one process presents as a crash fault to the transformed algorithms on other, correct, processes. Using these terms, this paper presents a 1-round, $n = f + 1$ general omission to byzantine transformation.

This paper comprises 5 sections: in Section II we present related work; in Section III we give our system model; in Section IV we present and prove correct our core transformation; and in Section V we give an example application.

II. Related work

Automatically transforming or translating algorithms to have fault tolerance in stronger fault models is a well-known idea. Transformations generally focus on the translation of faults and are therefore referred to as translations. Translations are distinguished by the synchrony of the system in which they are applied, since they, usually, cannot translate faults in both synchronous and asynchronous systems.

Synchronous translations. These include the composable translations of [12], where the composition of auxiliary translations results in two crash-to-byzantine translations: a 4-round, $n = 4f + 1$ translation and a 6-round, $n = 3f + 1$ translation. The translations differ in their broadcast primitives (reliable, validated, etc.). These results were improved upon

in [13], which presents 2, 3, and 4-round translations, with fault tolerance related to the round-increase: $n > \max(6f-3, 3f)$, $n > \max(4f-2, 3f)$ and $n = 3f+1$, respectively. Recently [16] proposed a translation assuming each process has a unique cryptographic signature and that at most one in any replicated pair of a process fail simultaneously, achieving under these assumptions a crash-to-authenticated-byzantine translation.

None of these translations caters to randomized algorithms: they all rely on comparing deterministic behaviour to check whether a process is exhibiting byzantine faults.

Asynchronous translations. In this setting, [17] proposes a crash-to-byzantine translation, by building a reliable broadcast primitive which filters messages to present byzantine faults as crashes. The protocol requires three broadcast rounds and has a fault tolerance of $n = 3f + 1$. Coan [14] builds upon this result and proposes a 2-round, $n = 4f+1$ translation, and a 3-round, $n = 3f+1$ translation; both making asynchronous crash fault-tolerant algorithms into byzantine fault-tolerant ones. Ho et. al. [18] proposes an asynchronous crash to byzantine translation using an ordered[1], authenticated and reliable broadcast primitive, yielding a fault tolerance of $n = 2f + 1$ if the primitive is built using cryptography, or $n = 3f + 1$ if it is built without cryptography. The broadcast primitive requires 3 rounds for each original message.

Lastly, our translation is related to that of Clement et. al. in [15], where it is shown that the non-equivocation given by a trusted subsystem is not enough to ensure the $2f + 1$ lower bound in the hybrid fault model, but that transferable authentication—e.g., digital signatures—is needed as well. In this setting, they create a 1-round, $n = f + 1$ crash to byzantine translation using a trusted subsystem that ensures non-equivocation and transferable authentication. Our translation distinguishes itself on the following points: their translation can translate asynchronous algorithms, while ours can translate both synchronous and asynchronous algorithms; they assume that the trusted subsystem can only fail by crashing, while we assume only the integrity and confidentiality of a TEE; their translation requires the algorithm to be deterministic, as their primitive requires simulation of the sending process' underlying state machine by the receiving process, while ours require no such simulation, and thus can translate both randomized and deterministic algorithms; each sent message from a process in their translation includes all previously received and sent messages to enable complete simulation of the sending process by the receiving process, making the message size grow linearly with the number of rounds. In our transformation, the messages have the constant overhead of a Message Authentication Code tag (MAC).

III. System Model

We will now present our system model, which is related to that of [18], where *hosts* (processes) can have several *linked* (connected) *agents* (state machines), each with separate state and progression. We will not, however, model links (channels) as separate state machines, but instead, use the abstraction of channels as passive components for delivering messages.

A. Processes and channels

A system comprises *processes*, *state machines* and *channels*: A distributed algorithm consists of n processes $\{p_0, \ldots, p_{n-1}\}$, each containing a set of state machines. State machines on the same process may be connected with reliable FIFO-channels; whereas state machines on separate processes can be connected only with unreliable channels, which may drop, reorder, duplicate, corrupt, or redirect messages. We make no synchrony assumptions about channels and processes, as we solely use the assumptions of the underlying system.

"State machines" here are any kind of automaton that consumes inputs and produces outputs. A simple example is Mealy Machines; however, we emphasise that our transformation also applies to more powerful machines such as pushdown automata or Turing machines. Note that some of the machines introduced by our translation—notably the wrapper machines and the machines appending MAC codes to messages—*cannot* be adequately represented as finite state machines, but require more powerful computational models.

Formally, a *state machine* in this paper consists of a set of states, transitions between the states, and an initial state. (When state, transitions, and alphabet of actions are all finite, the machine is a Mealy Machine.) Transitions are on the form $(s_0, t_{in}) \to (s_1, t_{out})$, where s_0 and s_1 are, respectively, the beginning and end states of the transition. t_{in} and t_{out} are tuples of the form (B, c), where B is an arbitrary length bit string, and c is a named channel. t_{in} is an input message on channel c required for the transition to initiate, and t_{out} is the output message produced and sent via channel c as a consequence of the transition. We use $\emptyset$ to indicate that a transition does not require an input or an output.

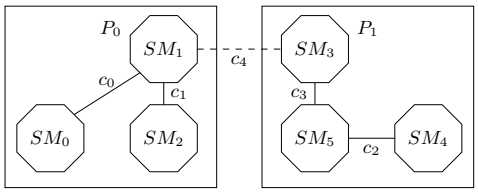

Fig. 1. Simple example of processes and channels in our system model.

Figure 1 shows an example of processes and channels. This simple setup has 2 processes, P_0 with the state machines $SM_0, \ldots SM_2$ and P_1 with state machines $SM_3, \ldots SM_5$. The state machines of P_0 are connected with the reliable channels c_0 and c_1, while the state machines of P_1 are connected with the reliable channels c_2 and c_3. The processes are connected through the unreliable channel c_4 between SM_1 and SM_3.

This model is inherently non-deterministic: state machines of a process can be in different states across otherwise identical runs. However, we do not rely on the determinism of an algorithm in our transformation, and our transformation does not introduce non-determinism.

[1] *FIFO*-ordered per sending process, and not ordered between distinct processes, since this would make the construction of such a primitive reduce to a consensus problem.

B. Faults

As mentioned, channels between processes are unreliable, and messages may be arbitrarily dropped from these channels. We use channel omissions as a proxy for all omission faults in the system, including send- and receive omissions. Note that this convention makes it impossible to detect where the message omission has taken place, a key problem in general omission resilient systems [19]. Formally, we model a crash failure by an ε-transition to a special state s_e, with no outgoing transitions. Once a state machine enters this state, it can neither transition away, receive inputs, nor create outputs, altogether behaving as if crashed.

To model byzantine faults, we must allow arbitrary behaviour. To this end, we model a byzantine fault of a process as the removal, addition or substitution of any number of that process' state machines with arbitrary replacement machines. We have conventional assumptions regarding cryptography: a byzantine fault cannot produce faults requiring the simulation of secrets – notably, we assume that no byzantine fault will be able to produce previously unobserved messages with correct Message Authentication Codes (MACs).

C. TEE guarantees

A TEE is a subsystem[2] providing Authenticity, Integrity, Confidentiality and Remote Attestation [20]–[22]. **Authenticity** is the property that a program saved to persistent storage and later loaded into a TEE, loads successfully into the TEE only if it is the unaltered original program. In other words, the program cannot be changed after it has been compiled into a TEE compliant binary[3]. **Integrity** is the property that only a program running in a TEE can change the data in the memory of a TEE, and the program can only change the data in the TEE that has been allocated to it. This integrity property still holds while the data resides outside the TEE, e.g. in persistent storage. We presently work with a weaker integrity property: unauthorised changes to data inside the TEE cause the immediate loss of all cryptographic secrets in that TEE. We choose this weaker model because nothing prevents a byzantine process from trying to impersonate a newly crashed process. In the worst case, the imposter process would replicate the crashed process' state, making it indistinguishable from the crashed process from the point of view of other correct processes, except for any confidential secrets the crashed process might have had. **Confidentiality** is the property that data created in the TEE can only be read by a program running inside the TEE, both during program execution and when residing in persistent storage. Furthermore, a program inside the TEE can only read its own confidential data. **Remote Attestation** is the property that a program running inside a TEE can *prove* its Authenticity to remote hosts. We will assume that the Remote Attesation property enables a confidential and integrity-protected exchange of symmetric cryptographic keys between TEE programs, as seen in e.g. [23].

For ease of modelling, we let Integrity encompass Authenticity, i.e. we view programs as data generated by an application running inside a TEE, which means that we will be given Authenticity as a by-product of Integrity.

We note that for practical TEE implementations, the Integrity property is up to common assumptions about cryptography, e.g., the TEE implementation *Intel® Software Guard Extensions* (SGX) detects integrity violations except with negligible probability under the assumption that AES128 is a random permutation [24]. Our notion of Integrity conforms to the one of SGX, where the processor will halt entirely on an integrity fault. Nothing prevents a byzantine process from trying to impersonate a crashed process, but the integrity property will ensure that the impostor process will not be able to replicate the confidentiality-protected secrets.

To model these properties, we need three more concepts:

1) An integrity protected area of each process.
2) A state machine (SM_c) for each such area in which all cryptographic secrets resides.
3) An attestation process (P_{RA}) and an attestation state machine (SM_A). An SM_A resides on each process, and together with P_{RA} enables Remote Attestation.

The state machine SM_c enjoys the Confidentiality property: so only SM_c may access cryptographic secrets residing on the process. We assume that no other process, including ones arising from byzantine failures, can "guess" these secrets. The integrity protected area enjoys the Integrity property in the sense that if it encounters a byzantine failure—if one of its state machines is substituted—the SM_c machine disappears.

Details of Remote Attestation varies with the implementation, e.g., the GlobalPlatform standard has no standardised Remote Attestation mechanism, but supports different implementations [25,26]. We assume that SM_A and P_{RA} are able to perfectly attest to state machines in the integrity protected area, i.e. uniquely identify the state machines and their states and verify that they are located in an integrity protected area. Moreover, we assume that, as part of that attestation, the remote attestation process is able to securely provision SM_c with a shared symmetric secret. These assumptions are supported by, e.g., Remote Attestation in SGX [27,28], which relies on trusting an *Intel® Attestation Server* (IAS).

As some implementations of TEEs have limited access to hardware peripherals, we do not allow channels from the integrity protected area of a process to another process. Instead, we introduce state machines outside the integrity protected area that has a reliable channel into the integrity protected area, and an unreliable channel to another process. We name these state machines *wrapper state machines* or simply *wrappers*.

D. Weakening of channels

To allow failing processes arbitrary behaviour, we allow correct state machines to receive messages from all state machines on faulty processes. Messages from faulty processes can be received on any and all channels by state machines on

[2]Note, we are not referring to the implementation of a TEE, which can be achieved in several ways, but the properties implementations have in common.

[3]Notice that this definition allows for a malicious compiler to change the code during compilation. This problem can be circumvented by using a trusted compiler running inside a TEE.

correct processes. The only exception we allow is that faulty state machines outside the integrity protected area cannot send messages on channels with both endpoints inside an integrity protected area. This is a realistic assumption: communication between modules in the same TEE is usually implemented by shared memory, which also resides inside the TEE, and thus is protected from tampering by modules outside the TEE.

E. Integrity violations vs. byzantine faults on SGX

We explain in more detail, using Intel SGX as an example, exactly in what sense we can construct a byzantine failure detector using SGX primitives. The primary protection against integrity violations in SGX is an integrity tree with the root stored in SRAM on-die [24]. All integrity-protected memory is MAC'ed in blocks collected in a Merkle-tree, with the root of this stored in on the CPU itself. When reading data from memory, MACs of the loaded blocks are verified against the root of the Merkel tree: If they do not verify, the CPU halts, requiring a physical reset. However, Gueron notes in [24] that, both the CPU *and its caches* are trusted components, i.e., reside within the "Trust boundary perimeter", which results in some confidentiality attacks [29]–[31], but no known integrity attacks exist at the time of writing. It follows that SGX might not protect against faults taking place in the CPU cache, nor any affecting the CPU itself.

Regardless, this mechanism protects against physical and software attacks on memory: No process, including other SGX processes, can (except with negligible probability) modify data in the integrity protected memory without the CPU halting [27]. This is an argument that *SGX can detect and protect against byzantine faults that change code and main memory after process initiation*. Note that a "byzantine fault" here may not "guess" secrets encapsulated in SGX.

IV. Transformation

We now present the transformation of general omission tolerant algorithms to byzantine fault-tolerant algorithms, under the assumption that the algorithm handles unreliable channels.

The core idea of our transformation is to move the entire set of state machines into the integrity protected area, thus guaranteeing the integrity of the algorithm. However, we will need to ensure the integrity of all messages between the protected areas, both on the unprotected areas of the processes and while in transit on the channels. To this end, the transformation involves modifying both processes and individual state machines. On the process-level, we move constituent state machines into the integrity protected area; on the individual state machine level, we sign messages from the state machines to ensure integrity during transmission.

We present the transform in stages: Steps 1–4 adds integrity protection and remote attestation; Steps 5–6 implements byzantine failure detectors by validating such attestations.

A. Stage 1: Integrity protection & remote attestation

These are steps 1–4:

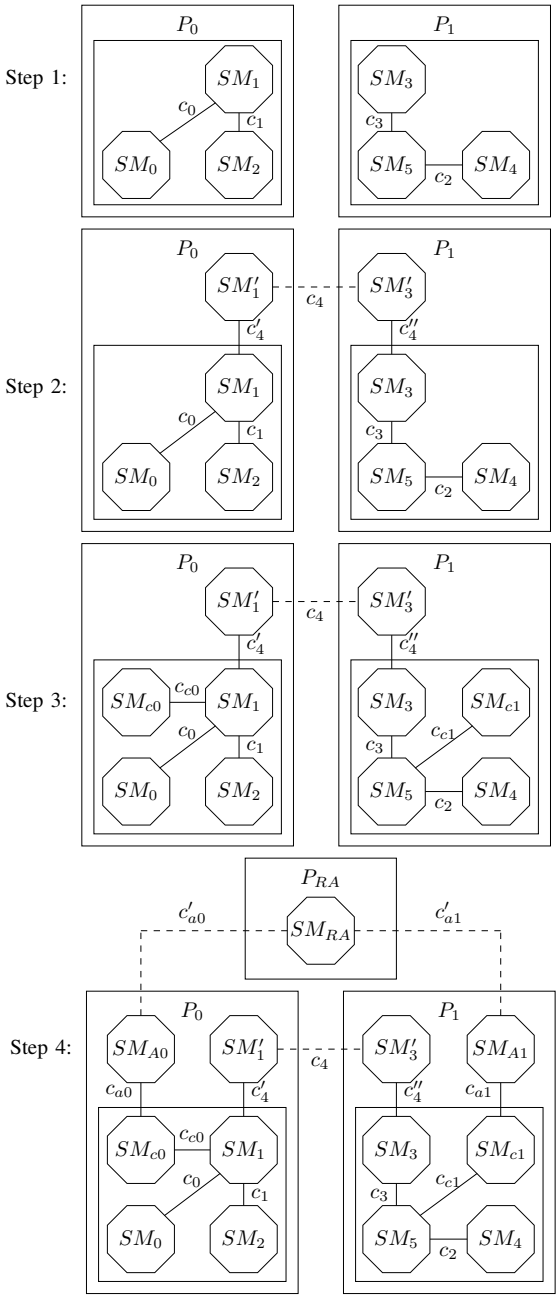

Fig. 2. Visualisation of the process-level transformation of Figure 1.

1) Encapsulate state machines in their process' integrity protected area.
2) Add wrapper machines and connect them to the integrity protected state machines, and to each other.
3) Add SM_c, a machine hosting secrets, to each process.
4) Add SM_A to all processes, and connect them to P_{RA}.

Note that the machines added in steps 2–4 are not finite. We visualise these steps for a very simple protocol in Figure 2.

Step 1 moves the state machines of each process into the integrity protected area of that process, visualised in Figure 2 as a box inside the processes. We temporarily remove channels to external processes.

Step 2 adds back the missing channels via a "wrapper"

state machine residing *outside* the integrity protected area of a process. For each (unreliable) inter-process channel, we add (1) a wrapper at each endpoint, (2) an (unreliable) channel with endpoints in the wrappers and (3) a (reliable) channel from each end-point wrapper to the original state machine end-point.

Step 3 adds a cryptographic state machine SM_c to all integrity protected areas. SM_c will store any secrets, will MAC messages to be transmitted, and will be cleared of secrets if a byzantine fault is detected in the integrity protected area.

Step 4 adds remote attestation: A machine SM_A on all processes and a P_{RA} process.

B. Stage 2: Detecting byzantine failures

All processes must first complete a pre-compute step to participate in the transformed algorithm. In the pre-compute step, the processes are (a) remotely attested and (b) provisioned with a shared cryptographic secret. All processes will share the same secret after completing step (b). These steps do not presuppose synchrony or eventual delivery: Any message of the remote attestation being infinitely delayed is equivalent to the first message of the underlying algorithm being infinitely delayed. With this shared key, a Message Authentication Code (MAC) is appended to messages, which verifies to other processes that the message was constructed by a non-failed integrity protected state machine. Because we have assumed that a byzantine failure manifests as the loss of cryptographic secrets, a correct MAC guarantees that the message in question originated from a correct process.

This mechanism requires MAC'ing all outbound messages and checking all inbound messages, steps 5–6:

5) MAC each outbound inter-process message sent from within the integrity-protected area. We transform each transition $(s_0, t_{in}) \to (s_1, (B, c))$ into the two transitions:

$$(s_0, t_{in}) \to (s', (B, c_c))$$
$$(s', (B || \mathsf{MAC}(B), c_c)) \to (s_1, (B || \mathsf{MAC}(B), c))$$

Here c_c is a reliable channel to SM_c, s' is a new state, where the state machines wait for a reply from SM_c, and $B || \mathsf{MAC}(B)$ is B appended with a valid MAC tag.

6) Verify the MAC on each inbound message. Each transition of the form $(s_0, (B, c)) \to (s_1, t_{out})$ becomes:

$$(s_0, (B||MAC(B), c)) \to (s', (B||MAC(B), c_c))$$
$$(s', (\text{"1"}, c_c)) \to (s_1, t_{out})$$
$$(s', (\text{"0"}, c_c)) \to (s_0, \emptyset)$$

Here s' is an intermediate state in which we await a reply from SM_c. If SM_c replies that the MAC is valid ("1") we proceed to s_1; otherwise, the MAC is invalid, and we return to the state s_0. In the latter case, the machine behaves *as if* no message was ever received.

Note that the machines added in steps 5 and 6 are not finite. Step 5 and 6 compose; if a transition has inter-process messages as both input and output, the transition is transformed by step 5, and the resulting transition with the inter-process output is then transformed by step 6. Note that if SM_c handles requests in order, neither step 5 nor step 6 introduce new non-determinism, because both steps use a waiting state, such that the requesting state machine must receive a response from SM_c before continuing. The entirety of the transformation introduces no new non-determinism since each external send and receive have simply been broken up into smaller atomic steps, during which the state machine is waiting.

C. Overhead

The number of new states and transitions are both linearly bounded by the number of messages to/from state machines on other processes. Each inter-process message send adds one state and one transition; each inter-process message receival adds one state and two transitions.

Counting messages, each inter-process message in the original system incurs additional intra-process messages in the transformed system for MAC'ing (2), for the sending and receiving wrappers (2), and for MAC verification (2); altogether 6 messages. Intra-process messages incur no overhead.

An algorithm with m_e inter-process (external) messages and m_i intra-process messages, the transformed system will use m_e inter-process messages and $6m_e + m_i$ intra-process messages, plus some number of messages for the initial remote attestation and provisioning pre-compute step.

We emphasise that the overhead is all intra-process: barring the initial remote attestation, the number of inter-process messages, messages *between* processes, does not increase. Moreover, in practice, each inter-process message has only the small constant-size overhead of a MAC.

D. Correctness

We show how a byzantine fault in any part of the process will be translated to either perpetual or sporadic message omission. A failed process with perpetual message omission seems crashed to all other processes since it cannot in any way affect the other processes in the system. We call this failure mode *constant omission*. Note that a single byzantine fault might affect more than one state machine and thus several different types of state machines on the same process.

Integrity protected state machines. Recall that a byzantine fault consists of the removal, addition, or replacement of machines; and that such a fault causes the removal of the SM_c state machine containing secrets of the integrity protected area. With the lost secrets assumed unguessable by new state machines, no machine in the failed process will be able to produce MACs (Step 5). State machines of correct processes will, therefore, behave as if they received no messages from the failed process (Step 6): all correct non-wrapper state machines experience constant omission from the faulty process. Note that this case includes the cryptographic machine SM_c.

Wrapper state machines. A *replacement* for a wrapper machine cannot fake correct MACs and so behaves equivalently to an unreliable channel: the failed machine can drop, reorder, corrupt and redirect the messages. Since the original algorithm is assumed to handle unreliable channels, the algorithm can handle all these faults except malicious corruptions which are

handled by the addition of the MAC. *Removing* or *adding* machines are special cases of replacements. Altogether a byzantine fault in wrapper machines translates to ordinary channel failures or constant omission.

Remote attestation state machines. We assume that remote attestation is an atomic operation, and thus do not consider faults at that site *during* that attestation process. We proceed to consider the two cases of the remote attestation machine SM_A experiencing a byzantine fault (1) before and (2) after the attestation completes.

(1) If the remote attestation machine suffers a byzantine fault *before* being remotely attested, it can drop, reorder, corrupt or redirect messages. However, it cannot create new valid messages in this process, nor can it read the shared secret that is provisioned to the cryptographic state machine as a result of the attestation. As such, a fault in the remote attestation machine can at worst have the consequence that either the cryptographic state machine is provisioned with a wrong secret, or not provisioned at all. Both cases translate to constant omission: any message sent from the integrity protected area will be dropped by any correct receiving process.

(2) If the remote attestation state machine experiences a byzantine fault *after* the remote attestation process has completed, that failure is equivalent to a byzantine fault in a wrapper state machine, since the remote attestation state machine resides outside the integrity protected area, and does not have special privileges.

Clearly, an unhandled byzantine fault in P_{RA} violates all of the former guarantees, as byzantine processes can now be provisioned with the shared secret. Therefore, the remote attestation process must be protected in a way that guarantees that only correct processes are provisioned with the shared secret. Such mechanisms exist [32]–[34].

Except for byzantine faults in the remote attestation process (P_{RA} and SM_{RA}), we have seen that on all ordinary processes, the transformation translates byzantine faults to omissions, constant omissions, unreliable channels and crashes.

V. Transformation example

We exemplify the transformation with the classic central-server mutual exclusion algorithm (CSME), see, e.g., [35]. In mutual exclusion, a collection of processes share access to one or more resources, referred to as the *critical section* (CS). To prevent data-races and race-conditions, a mutual exclusion algorithm ensures that at most a single correct process has access to the critical section at any given time, that is, at most one correct process may execute in the CS at a time (safety) and any requests to enter and exit the critical section eventually succeeds (liveness). As an aside, this example demonstrates that the translation has relevance outside consensus problems.

This algorithm does not provide liveness under process crashes. If the server crashes, no process can gain access tokens, and thus no requests for access will succeed. Similarly, if a client crashes with the token, the token is lost. However, safety is still guaranteed: a process cannot access the critical section without a token. Under byzantine faults, neither safety nor liveness can be guaranteed, e.g., for instance, the server could serve the tokens to all requests without waiting for acknowledgements from the clients.

We now show how applying the transformation from Section IV to CSME yields an algorithm which provides the same guarantees (safety) under byzantine failures as the untransformed does under crash failures.

First, we model the algorithm as Mealy Machines, channels and processes. We model two different state machines: a client state machine, and a server state machine. The modelling is more straight-forward with pushdown automata, but we continue with Mealy Machines for clarity.

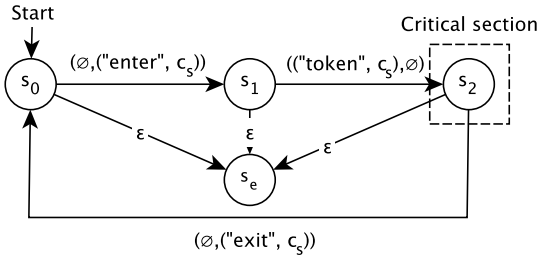

Fig. 3. Client state machine (SM_1 and SM_2) from the central server algorithm for mutual exclusion. Note that the channel c_c represents the unreliable channel to the server, and is different across client instances.

The client state machine is modelled in Figure 3. It is in one of four states: not having requested access to the CS (s_0), waiting for the token from the server (s_1), being in the CS (s_2), or have crashed (s_e).

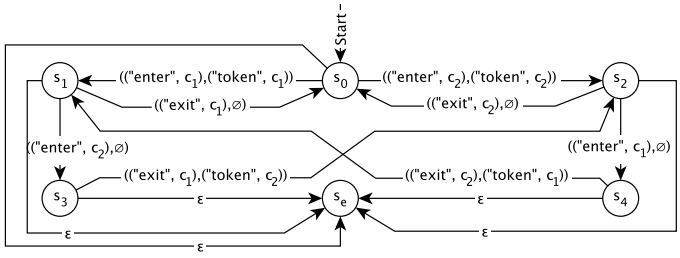

Fig. 4. Server state machine (SM_0) from the central server algorithm for mutual exclusion. Note that this server state machine can only handle two client processes.

Figure 4 shows a model of the server state machine in an algorithm with two client processes. It encompasses 6 states:

s_0 no client has the token, no requests have been received
s_1 client 1 has requested the token
s_2 client 2 has requested the token
s_3 client 1 has the token, client 2 has requested the token
s_4 client 2 has the token, client 1 has requested the token
s_e the state machine has crashed.

We will assume a CSME system with two client processes P_1 and P_2 each running an instance of the state machine from Figure 3, with an unreliable channel to the server process P_0, which runs an instance of the state machine from Figure 4.

We first show how the state machines are transformed to have their messages MAC'ed by SM_c and to have SM_c verify

the messages they receive. This is done by applying the state machine transformation described in Section IV.

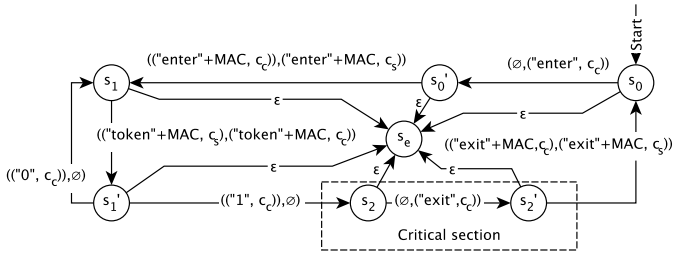

Fig. 5. Client state machine in CSME, after it has been transformed to handle byzantine faults.

Figure 5 shows the client state machine after the transformation. To enter the critical section, a request is sent to SM_c over the inter-process channel c_c for MAC'ing and transitions to s_0'. When the MAC'ed message is returned, this message is sent to the server process. The state machine is now at s_1, which is equivalent to s_1 in the untransformed state machine. c_s is now a channel to/from the wrapper state machine. When a token is received from c_s, the message's MAC is sent to SM_c for verification. If the MAC is correct, the state machine will transition to s_2, which is in the critical section and is equivalent to s_2 in the untransformed state machine. A similar process is followed when exiting the critical section: the exit message is sent to SM_c for MAC'ing, and the returned message is sent to the wrapper state machine for redirection to the server process. Figure 9 shows the transformed server state machine, omitting s_e for readability. Figures 6 and 7 show the processes before and after the transformation, respectively.

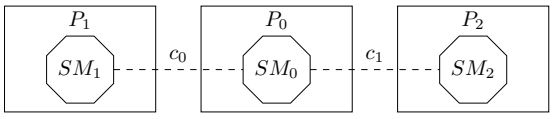

Fig. 6. Processes and channels in the central server algorithm for mutual exclusion, with one server and two clients.

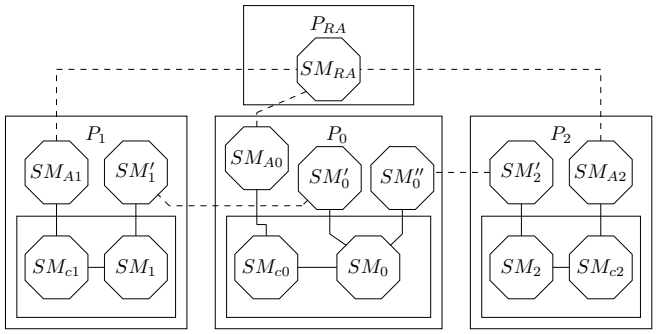

Fig. 7. Processes and channels in the central server algorithm for mutual exclusion, after they have been transformed to handle byzantine faults.

In the transformed CSME algorithm, correct processes exhibit the *same behaviour* under byzantine faults, as correct processes under crash faults in the original algorithm. The example here illustrates exactly that the translation directly inherits fault resilience properties of the original algorithm: CSME has safety but not liveness under crash and omission failures, so the transformed CSME algorithm also has safety but not liveness, but under the stronger fault model of byzantine failures and omission failures.

For instance, consider the fault where the server state machines try to serve tokens on any request, without waiting for the token to be first released by the client who currently holds it, violating safety. Model this byzantine fault as SM_0 has been exchanged with SM_{byz} (see Figure 8), which serves a token to any request by the clients. Under this byzantine fault happens, none of the tokens can be MAC'ed. Thereby, any correct client process (Figure 5) will get stuck in a loop between s_1 and s_1', when SM_c rejects the token as it has not been MAC'ed. This is behaviour is equivalent to the server process crashing: a token will never be accepted, and the client will be unable to enter the CS.

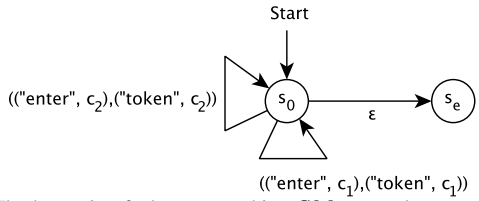

Fig. 8. The byzantine fault state machine SM_{byz} on the server process.

VI. Conclusion

We have given a 1-round, $n = f + 1$ transformation which translates byzantine faults to crashes and omission. The transformation relies on the integrity and confidentiality guarantees of a Trusted Execution Environment and requires remote attestation. Using this model, we have shown how, by the properties of a TEE, the transformation translates the byzantine faults into either crashes, unreliable channels or omission faults. This ultimately makes an argument for the use of TEEs as trusted subsystems, and for the validity of using TEEs as the subsystem running other small trusted subsystems, by use of the presented transformation.

As future work, it would be interesting to investigate (1) performance ramifications on this technique when applied to more complex distributed algorithms, and (2) possibly implement automated variants of this translation for protocol specification, e.g., Session types, SDL or UML-variants.

References

[1] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[2] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[3] G. Bracha and S. Toueg, “Asynchronous consensus and broadcast protocols,” *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, 1985.

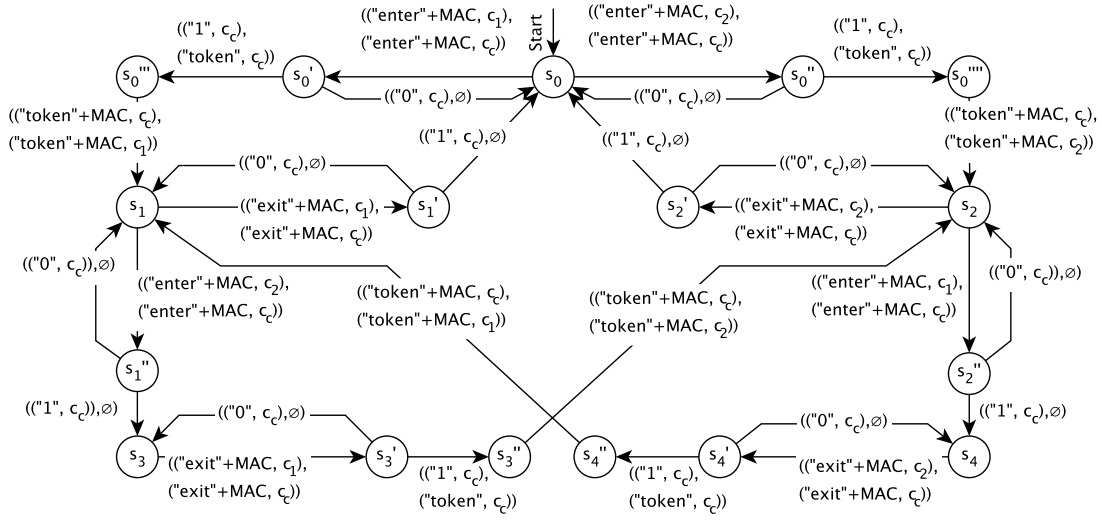

Fig. 9. Server state machine in the central server algorithm for mutual exclusion, after it has been transformed to handle byzantine faults. Note that s_e has been omitted for improved readability.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, “Attested Append-only Memory: Making Adversaries Stick to Their Word,” in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07. New York, NY, USA: ACM, 2007, pp. 189–204.

[6] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “EBAWA: Efficient Byzantine agreement for wide-area networks,” in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium On*. IEEE, 2010, pp. 10–19.

[7] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, “CheapBFT: Resource-efficient byzantine fault tolerance,” in *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 295–308.

[8] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, “Efficient byzantine fault-tolerance,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[9] J. Liu, W. Li, G. O. Karame, and N. Asokan, “Scalable Byzantine Consensus via Hardware-assisted Secret Sharing,” *arXiv preprint arXiv:1612.04997*, 2016.

[10] J. Behl, T. Distler, and R. Kapitza, “Hybrids on Steroids: SGX-Based High Performance BFT,” in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys ’17. New York, NY, USA: ACM, 2017, pp. 222–237.

[11] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, “TrInc: Small Trusted Hardware for Large Distributed Systems.” in *NSDI*, vol. 9, 2009, pp. 1–14.

[12] G. Neiger and S. Toueg, “Automatically increasing the fault-tolerance of distributed algorithms,” *Journal of Algorithms*, vol. 11, no. 3, pp. 374–419, Sep. 1990.

[13] R. A. Bazzi and G. Neiger, “Simplifying Fault-Tolerance: Providing the Abstraction of Crash Failures,” Georgia Institute of Technology, Technical Report, 1993.

[14] B. A. Coan, “A compiler that increases the fault tolerance of asynchronous protocols,” *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1541–1553, 1988.

[15] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues, “On the (limited) power of non-equivocation,” in *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*. ACM Press, 2012, p. 301.

[16] D. Mpoeleng, P. Ezhilchelvan, and N. Speirs, “From crash tolerance to authenticated byzantine tolerance: A structured approach, the cost and benefits,” in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, Jun. 2003, pp. 227–236.

[17] G. Bracha, “Asynchronous Byzantine agreement protocols,” *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[18] C. Ho, D. Dolev, and R. van Renesse, “Making Distributed Applications Robust,” *International Conference On Principles Of Distributed Systems*, vol. 4878, pp. 232–246, 2007.

[19] K. J. Perry and S. Toueg, “Distributed agreement in the presence of processor and communication faults,” *IEEE Transactions on Software Engineering*, no. 3, pp. 477–482, 1986.

[20] “Introduction to Trusted Execution Environments,” May 2018.

[21] Intel, “Intel® Software Guard Extensions Programming Reference,” Octobber 2014.

[22] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted Execution Environment: What It is, and What It is Not,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64.

[23] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, “Innovative Technology for CPU Based Attestation and Sealing,” in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, vol. 13, 2013, p. 7.

[24] S. Gueron, “A Memory Encryption Engine Suitable for General Purpose Processors,” Intel Development Center, Israel, Tech. Rep. 204, 2016.

[25] “GlobalPlatform Technology TEE System Architecture Version 1.2,” Nov. 2018.

[26] C. Shepherd, R. N. Akram, and K. Markantonakis, “Establishing mutually trusted channels for remote sensing devices with trusted execution environments,” in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, p. 7.

[27] V. Costan and S. Devadas, “Intel SGX Explained.” *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[28] “Intel® Software Guard Extensions Developer Reference — Intel® Software.”

[29] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache Attacks on Intel SGX,” in *Proceedings of the 10th European Workshop on Systems Security - EuroSec’17*. Belgrade, Serbia: ACM Press, 2017, pp. 1–6.

[30] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software Grand Exposure: SGX Cache Attacks Are Practical,” in *Proceedings of the 11th USENIX Conference on Offensive Technologies*. USENIX Association, 2017, p. 12.

[31] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution,” *arXiv preprint arXiv:1802.09085*, 2018.

[32] F. Stumpf, O. Tafreschi, P. Roder, and C. Eckert, “A Robust Integrity Reporting Protocol for Remote Attestation,” in *Future Wireless Networks and Information Systems*. Springer Berlin Heidelberg, 2006.

[33] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, “Remote attestation on program execution,” in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing - STC ’08*. Alexandria, Virginia, USA: ACM Press, 2008.

[34] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O’Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, “Principles of remote attestation,” *International Journal of Information Security*, vol. 10, no. 2, Jun. 2011.

[35] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Mar. 2004.