Distributed Dynamic Condition Response Graphs in a Trusted Execution Environment

Preliminaries

Mikkel Gaub, Malthe Ettrup Kirkbro & Mads Frederik Madsen

December 19, 2017

Contents

1 Introduction

In an increasingly digital business world, collaboration between companies is an everyday occurrence. An intuitive way to model such a collaboration, is by using workflows, which will be the basis of this report. Performing collaborative work on a workflow between multiple, possibly untrusted, parties is mostly the same challenge as is posed in any distributed network. Most solutions to this are, however, cumbersome in that they require a very large amount of data to be transmitted. The new innovation applied in this project is Intel SGX and as such it will be investigated what sort of advantages SGX grants in the implementation of a distributed network.

1.1 Problem

During this project we will investigate whether or not it is possible to create a secure implementation of a distributed Dynamic Condition Response [1] (DCR) engine using Intel Software Guard Extensions [2] (SGX). We define this distributed SGX-DCR engine informally as a system where all correct processes agree on each state a given workflow has been in. As the majority of the challenge of the previous statement is making the proposed system distributed, the strong guarantees provided by Intel SGX will be investigated for application in order to achieve consensus among the participating parties.

1.2 Plan

The project will be split into two phases:

1. The first phase, which is described in its entirety in this report, is an investigation into the feasibility and usability of Intel SGX in conjunction with distributed DCR graphs.

2. The second phase, will be the design and implementation of the DCR-SGX system along with a more in-depth foray into Intel SGX and its capabilities. Should this prove more easily attainable than it is currently believed to be, the implementation will be expanded to include generic applications, creating a distributed network of smart contracts [3].

2 Related works

An existing platform which partially provides a solution to the goal of this project is Ethereum [4]. Ethereum is a distributed computing platform based on blockchain technologies, which does, however, also feature a currency, making computations disproportionately expensive, both monetarily and computationally due to the large amounts of computations required by proof-of-work based blockchains. Even though SGX is a new technology, it has already been proposed to solve several issues in the blockchain concept. SGX has been used as an intermediate for faster consensus about transactions in [5], however it still relies on the underlying blockchain to prevent double-spending.

In [6] SGX has been utilized to implement a new Byzantine fault tolerance (BFT) consensus algorithm, called *FastBFT*, which solves some of the scalability issues of blockchains. This is done by using the *strawman design* where a request is sent to a node, the *primary*, who prepares a vote by distributing parts of a secret to all nodes. The secret can be reconstructed given enough parts and then compared to the hash of the secret which is common knowledge. This means that consensus can be achieved in only $O(n)$ messages, rather than the $O(n^2)$ messages achieved by other algorithms. The reason that Intel SGX is needed for this algorithm is that the primary, who distributes the secrets, can fake being any node as he has all parts of the secret. Trusted Execution Environments (TEE) as introduced by Intel SGX, means that these secrets can be computed and distributed without the primary every having access to them. A further issue with the strawman design, is that the primary can change the orders of requests, thereby equivocate which request is being voted on. This is once again solved by Intel SGX, by numbering requests with a trusted counter that can only be incremented.

In particular, the Hyperledger Sawtooth [7] project is interesting as it is closely related to the goals of our project. Hyperledger Sawtooth is an ongoing blockchain project, which replaces the need for mining by using a consensus algorithm called Proof of Elapsed Time (PoET). In PoET a number of nodes who choose to participate as validators each create a timer using Intel SGX. This timer is different per validator and contains a timestamp some time in the future, with a degree of randomness. A validator can also check that a timer is valid, using Intel SGX, and also whether or not that timer has expired. The first validator to distribute a valid and expired timer to the rest of the validators, is elected leader for the current round of decision-making. This is a strong indicator that Intel SGX can be used for efficiently attaining consensus.

3 Background

The three primary components of the proposed solution are the following: the workflow, which will be described with DCR, Intel SGX and method for achieving distributed consensus. We assume the reader has some familiarity with DCR graphs and therefore it will not be described in this paper.

3.1 SGX

Intel SGX is a TEE technology which allows users to define protected areas of memory, so called *enclaves*. Intel guarantees [8] that any code run and data loaded in the TEE is protected from access by any process running outside of the enclave. More specifically, the guarantees encompass *confidentiality*, i.e. no other process can read the data in the enclave, and *integrity*, i.e. no other process can modify the data in the enclave. If the enclave contains secrets which the user wishes to keep, but still does not want to trust external processes with, the enclave can be sealed on disk, essentially encrypting it for later use. This means that data can be stored and processed securely, with its security guaranteed by Intel, if done properly. An additional feature of Intel SGX is that the result of any code run using Intel SGX can be verified by other users, given the code and the output.

This is all made possible by unique keys generated during manufacturing and permanently stored inside the fuse array of the processor. Some of these keys are known by Intel for the system to be recognized when contacting Intel servers.

In short, the major innovation in Intel SGX is the option of running hardware secured software, which enables tamper proof messages, where the sender can be verified as being a correct process. This eases some of the inherent problems in consensus, as seen in [6] and [7].

3.1.1 Enclave

What allows SGX enabled CPUs to provide these strong guarantees builds on the following two hardware details:

Processor Reserved Memory (PRM) is a sequential block of memory reserved for SGX, inaccessible from untrusted software and even hardware.

Enclave mode is a mode under which a logical processor gains access to the PRM.

Using these hardware facilities SGX defines the concept of an enclave. An enclave is, as its name suggests, a software module isolated completely from the rest of the system. Its memory is located solely in PRM, preventing access by other processes and enclaves running on the system. The PRM is protected from non-enclave processes by enclave mode. When a process is not in enclave mode and tries to access the PRM, the memory access is denied [2]. The PRM of an enclave process is protected from accesses by other processes in enclave mode by the SGX Enclave Control Structure (SECS). The SECS holds meta data about the enclave processes, and among this is a virtual-to-physical memory mapping called the Enclave Page Cache (EPC). Through the EPC an enclave's access to physical PRM is restricted to what has been allocated for the enclave-process [2].

While SGX provides powerful guarantees trough its enclave concept, it does not guarantee software correctness and will not protect against flawed software. Instead SGX encourages developers to isolate a minimal piece of their software, the Trusted Computing Base (TCB), in a trusted enclave environment, and keep the remainder as traditional system processes [9]. By minimizing the size of the TCB, and thus the amount of code one must trust, common security principles indicate that the chance of security flaws decreases [9].

In order for an isolated enclave to be useful, communication between trusted and untrusted software is enabled through *enclave calls* (ECALL) and *out calls* (OCALL). This interface must be defined at compile-time, specifying an API of ECALLs for the enclave as well as any untrusted services needed as OCALLs, in a *EDL*-file [9].

When built, an enclave module is a plain binary on the untrusted file system. As an enclave under such circumstances would be vulnerable to tampering before initialization, SGX enforces a strict signature policy. An enclave must include an *Enclave Signature* containing **(a)** a hash of the code and initialization data of the enclave, **(b)** the author's public key and **(c)** an enclave version/product number. During enclave initialization a hardware check is performed, ensuring the Enclave Signature matches the enclave binary loaded from the file system.

Another powerful tool of an SGX enclave is the ability to read from and write data to an untrusted storage medium while ensuring confidentiality of its contents. Such capabilities are needed as the PRM is volatile and will not persist after shutdown. In SGX this process is known as *sealing*. Sealing allows encryption and decryption using a *Seal Key*, which is confidential to the Enclave Signature.

3.1.2 Attestation

Attestation, within the Intel SGX platform, is the action of verifying the existence of specific enclaves. The applications of this are two-fold, in the case where there are multiple enclaves running locally on the same CPU and in the case where enclaves need to communicate to enclaves running on external CPUs. These two situations are handled by two different processes, aptly named local and remote attestation.

Local attestation builds on a secret fused into the CPU. An enclave can use an SGX instruction to generate a report uniquely identifying the enclave. This report is MACed with a key derived from the fused secret and enclave identifier. It is not possible to fake such a MACed report, as the MACing happens in hardware components that ensure that the report corresponds to the caller enclave. The fused secret is not directly accessible by software components, but can only be used by certain hardware instructions that protect against malicious use. An attestation challenger will ask for a MACed report from the client enclave, and after receiving it derive the same key to verify the MAC. Because a report can only be created by the enclave it describes, the challenger can be certain of which enclave the client is running if the MAC is valid. To prevent replays the MACed report is allowed to contain a block of arbitrary data, in which the challenger can require a nonce.

Remote attestations build on the same concept of a report, but as challenger and client are now on different CPUs, they no longer hold the same fused secret. Instead remote attestation relies on the group signature scheme Enhanced Privacy ID (EPID) and a third party issuer. Each SGX enabled CPU is granted an EPID Member Private Key by the third party issuer some time after manufacturing. Like the fused secret, this private key is not directly accessible to an enclave. Under the EPID scheme, all issuer generated private keys share the same public key (EPID Group Public Key). When remote attestation takes place, the client enclave generates a report and attests locally with the Quoting Enclave (QE). The QE, now convinced of the client enclave's identify, strips the MAC off the report and instead signs it with the EPID Member Private Key, which the QE has special privileges to access[1]. The remote challenger can verify the signed report with the EPID Group Public Key and be sure that **(a)** the report is signed by an EPID Member Private Key, **(b)** EPID Member Private Keys are granted secret to QEs, and **(c)** QEs will not sign false reports.

[1]Enclaves signed by Intel have special privileges throughout SGX. This allows SGX to implement complicated SGX instructions in software.

3.1.3 Monotonic counters

One of the functions provided by Intel SGX, which is especially relevant for this project, is the access to *trusted monotonic counters*. As indicated by the name, monotonic counters are integer counters, that can only be incremented. They are implemented as a block of non-volatile memory accessible only through SGX instructions, protecting against replay attacks.

3.2 Consensus

One of the goals is to establish agreement in a history of a workflow (see section 1.1). Agreement on this history implies a solution to *consensus*. See [10] for formal definition. Solving consensus will likely be the part where Intel SGX is the most applicable, suggested by the consensus solutions in fastBFT [6] and PoET [7]

In a famous result from 1985, known as the FLP impossibility result [11], it was shown that consensus was impossible to solve in an asynchronous setting with as little as a single crashing process. The FLP impossibility result can, very simplistically, be expressed as the problem of distinguishing a failed process from a delayed message in an asynchronous system.

4 Implementation

In order to demonstrate the application of Intel SGX we show the implementation of a monotonic counter in C++.

4.1 Trusted monotonic counter example

The implemented example is separated into two parts:

TMCtest is the enclave code. It is able to seal and unseal data, so that it is protected from tampering outside the enclave. It can create, increment and read monotonic counters. Due to the nature of enclaves, it is not able to make any I/O calls.

Untrusted is the untrusted part of the code. It essentially acts as a wrapper for the enclave, in that it is responsible for all I/O calls. It is able to take and parse commands from the command-line and call the appropriate enclave functions. It is also able to print and read sealed monotonic counters.

The interface between TMCtest and Untrusted is specified by the EDL file `TMCtest.edl` (snippet 1). The EDL specifies which functions the untrusted code can call in the enclave (ECALLs), in the `trusted` block. It is also able to specify which functions the enclave can call in the untrusted code (OCALLs), in the `untrusted` block, but there are none of these present in the example:

```
enclave {
    from "sgx_tae_service.edl" import *;
    trusted {
        /* define ECALLs here. */
        public uint32_t create_sealed_monotonic_counter([out, size=sealed_mc_size]
            uint8_t* sealed_mc_result, uint32_t sealed_mc_size);
        public uint32_t increment_monotonic_counter([in, out, size=sealed_mc_size]
            uint8_t* sealed_mc_result, uint32_t sealed_mc_size);
        public uint32_t read_sealed_monotonic_counter([in, out, size=sealed_mc_size]
            uint8_t* sealed_mc_result, uint32_t sealed_mc_size, [in, out] uint32_t* mc_value);
        public uint32_t get_size();
    };
    untrusted {
        /* define OCALLs here. */
    };
};
```

Snippet 1: `TMCtest.edl`, the interface between the enclave and the untrusted code.

The struct created for the monotonic counter data is called `monotonic_counter` (snippet 2), and is only accessible by the enclave. It consists of the current monotonic counter value, as wells as the unique id of the monotonic counter, which SGX uses for accessing the appropriate non-volatile memory.

```
typedef struct _monotonic_counter
{
  sgx_mc_uuid_t mc;
  uint32_t mc_value;
} monotonic_counter;
```

Snippet 2: `monotonic_counter` struct in `TCMtest.cpp` .

When creating and accessing monotonic counters, SGX needs to establish a session with the Platform Services Enclave (PSE), which is a protected enclave only accessible by other local enclaves. The

PSE provides access to protected functionality, such as monotonic counters, sealing and attestation. After establishing this session, the creation of a monotonic counter is simply a call to `sgx_create_monotonic_counter`. When the monotonic counter has been created, the struct is sealed and passed to the untrusted code. Here it is important to note that the memory containing the unsealed data must be cleared before returning the sealed data to the untrusted code, in order to prevent leaking data.

Incrementing a monotonic counter is very similar to creating it. Due to the fact that the untrusted code only has access to sealed structs, however, the passed data must first be unsealed. This is accomplished with a call to `sgx_unseal_data`. The data is also verified (i.e. that the value of the monotonic counter is as expected), to ensure that the data has not been tampered with. The verification does not guarantee that another instance of the enclave has not unsealed the data and incremented the counter. However, it has to be another instance of the exact enclave, as the sealing process is encryption with a Seal Key, which is unique to the enclave and platform [9].

5 Challenges

The primary challenges in this project are concerned with SGX and consensus.

5.1 SGX security

All aforementioned hardware guarantees are given by Intel, but since very little of the proprietary technology is documented, all security properties of SGX rely on the correctness of Intel's hardware implementation. Historically Intel has had vulnerabilities in similar hardware components, like the CVE-2017-5689 security incident exposed in [12], allowed unprivileged access to the Intel Active Management Technology.

SGX claims to provide confidentiality and integrity for running enclaves. This is ensured by preventing untrusted software access to the PRM and enclaves access to other regions of the PRM besides their exclusive region. However, while it is not possible to breach the integrity of PRM, [2] shows several issues regarding confidentiality. Because enclaves use the system page table even for data residing on PRM, it is possible for untrusted software to learn the page access order by manipulating the OS controlled page table. Another possible attack vector is to perform cache timing by cleverly choosing the physical memory position of enclave memory pages on set associative caches. By mapping snooping software to the same cache set as the enclave, the snooping software can evict the enclave's memory from the cache

and use timing to figure out if it is accessed again. Lastly, processors with hyper-threading support are vulnerable to instruction snooping, as a snooping process sharing physical core with an enclave can use performance counters to determine which instructions the out-of-order scheduler is able to run in parallel with the enclave. The presence of these attacks means that enclaves using data-dependent memory access will likely not ensure confidentiality. However, to our knowledge, no publicly known vulnerabilities exist on the integrity guarantee. And since our system has very few confidentiality requirements, the known vulnerabilities do not seem problematic for the purposes of this project.

As mentioned SGX does not protect against flawed software, so it is up to the developer to prevent side-channel attacks through OCALLs that might expose secret data. A noteworthy mention to illustrate the vigor needed is the common pitfall mentioned in [9] when using ECALLs and OCALLs. Because enclaves are compiled using a standard C++-compiler, structure padding is likely to happen. The SGX environment does not protect against leaking secret information through uninitialized structure padding when passing structures in ECALLs and OCALLs – this, and other common security pitfalls, are not part of any SGX security guarantees. Intel recommends to always clear secrets from the enclave memory after use in [9], regardless of the guarantees given by the PRM, underlining the risk of unintended vulnerabilities.

5.2 Consensus

Given that our problem deals with agreeing on a history of a workflow, we must somehow circumvent FLP impossibility. A few solutions for circumventing FLP-impossibility have been suggested, for instance using a probabilistic protocol, which achieves consensus except for some negligible probability. Most notable of these is perhaps the blockchain technology, (formal analyses of the complexity and probability of success can be found in [13]). However, the basis of a blockchain succeeding is an incentive for peers to use computing power to secure the system. This incentive takes the form of a currency [14]. While a clever solution, we are not interested in basing our system on a currency, thus subjecting the viability of the system on the rise-and-fall of the rate and popularity of the currency. Instead we have looked at alternatives for side-stepping FLP-impossibility. We have found the survey by Correia et al. [15] to be very helpful for this. The following three options present themselves:

One solution is to assume that some subsystem or step of the protocol behaves synchronously. This is known as *partial synchrony*. If the system does not live up to these assumptions, the protocol will naturally fail. The challenge is this solution to identify a subsystem that can reliably be assumed to

behave synchronously.

A variation on *partial synchrony* is to assume some subsystem has some trusted properties, which are otherwise not guaranteed in the environment. These subsystems are known as *wormholes*. According to the survey [15], the Trusted Platform Module (TPM) is an example of a wormhole, and since SGX can implement some of the same concepts as those available in TPM, it might be possible to use SGX as a wormhole. The survey mentions several protocols which might be transformed to solve the consensus problem, including one that uses a Unique Sequential Identifier Generator, which is essentially a monotonic counter. The challenge in this approach is to identify which of the properties in the wormhole allow side-stepping FLP impossibility, and then implementing them using SGX.

The most simple way of side-stepping FLP impossibility, is to modify the consensus problem, i.e. weaken the requirements so that some failures are allowed. For instance, the Paxos algorithm [16] satisfy variations on the agreement and integrity requirements, but does not satisfy the appertaining termination requirement [17]. The challenge in this approach is to identify which requirements can be weakened with the least repercussions, and how.

6 Future Work

The research demonstrated in this paper, shows the proposed project is reasonable. An example of the implementation of a monotonic counter has been shown to function, and the considerations and challenges regarding Intel SGX have been explored. Exactly how the advantages given by Intel SGX can be leveraged to create distributed consensus, more efficiently or to a greater degree than a typical distributed systems, is still unclear. As Intel SGX is still rather new technology, attaining a greater understanding of it is challenging, given the few in-depth descriptions in existence.

For the second part of this project, the research described in this paper will be applied in the design of an algorithm to perform decentralized code executions using Intel SGX. This algorithm will then be expanded to the development of a decentralized DCR engine. If this does not expend the available resources, the implementation of the DCR engine will be expanded to a platform for running arbitrary code decentralized.

References

[1] Thomas T. Hildebrandt & Raghava Rao Mukkamala. *Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs*. arXiv preprint arXiv:1110.4161 (2011).

[2] Victor Costan & Srinivas Devadas. *Intel SGX Explained*. IACR Cryptology ePrint Archive (2016)

[3] Nick Szabo. *Formalizing and Securing Relationships on Public Networks*. First Monday, Volume 2, Number 9 - 1 September (1997) Accessed 2017-12-18.

[4] *Ethereum White Paper*. `https://github.com/ethereum/wiki/wiki/White-Paper`. Accessed 2017-12-11.

[5] Rakesh Gopinath Nirmala. *Improving the Security and Efficiency of Blockchain-based Cryptocurrencies*. Master thesis. Alto University. 2017-08-28

[6] Jian Liu & Wenting Li & Ghassan O. Karame & N. Asokan *Scalable Byzantine Consensus via Hardware-assisted Secret Sharing*. arXiv preprint arXiv:1612.04997 (2016)

[7] *PoET: Proof of Elapsed Time*. On-line resource. `https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html#proof-of-elapsed-time-poet` Accessed 2017-12-18.

[8] *Intel® Software Guard Extensions*. `https://software.intel.com/en-us/sgx` Accessed 2017-12-18.

[9] *Intel® Software Guard Extensions Developer Guide*. `https://software.intel.com/en-us/documentation/sgx-developer-guide` Accessed 2017-12-18.

[10] George Coulouris and Jean Dollimore and Tim Kindberg and Gordon Blair *Distributed Systems, Concepts and Design, 5th ed.* Pearson education (2005)

[11] Michael J. Fischer & Nancy A. Lynch & Michael S. Paterson. *Impossibility of distributed consensus with one faulty process*. Journal of the ACM (JACM) 32.2 (1985)

[12] *Silent Bob is Silent*. `https://embedi.com/wp-content/uploads/dlm_uploads/2017/11/silent-bob-is-silent.pdf` Accessed 2017-12-18.

[13] Andrew Miller & Joseph J. LaViola, Jr. *Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin*. On-line paper (2014)

[14] *Bitcoin White Paper*. `https://bitcoin.org/bitcoin.pdf`. Accessed 2017-12-11.

[15] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves & Paulo Verissimo. *Byzantine consensus in asynchronous message-passing systems: a survey*. Critical Computer-Based Systems 2.2 (2011)

[16] Leslie Lamport *The Part-Time Parliament*. ACM Transactions on Computer Systems (TOCS) 16.2 (1998)

[17] Jean-Philippe Martin & Lorenzo Alvisi. *Fast Byzantine Consensus*. IEEE Transactions on Dependable and Secure Computing 3.3 (2006)