IEEE Copyright Notice

This work has been published in “2018 IEEE International Conference on Software Quality, Reliability and Security (QRS) “, 16 - 20 July 2018, Lisbon, Portugal.
DOI: 10.1109/QRS.2018.00050

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8424988

Hardening Application Security using Intel SGX

Max Plauth, Fredrik Teschke, Daniel Richter and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Digital Engineering
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—The release of Intel's *Software Guard Extensions* (SGX) refueled the interest in trusted computing approaches across industry and academia. The corresponding hardware is available, but practical usage patterns and applications are still lacking notable prevalence rates. This paper addresses this gap by approaching trusted computing from the point of view of a software engineer. To help developers in overcoming the initial hurdles of integrating SGX with existing code bases, a small helper library is presented. Furthermore, hardening strategies are identified and applied in a case study based on the simple KISSDB database, demonstrating how SGX can be used in practice.**

***Keywords*-Secure Processing, Enclaves, Trusted Computing**

I. Introduction

Cloud computing has proven itself as a viable and popular business model, making data security a very hot topic. Encryption as a way of securely *transporting* data is a time-proven concept. By comparison, techniques for secure data *processing* are still in their infancy. For some time now there have been trusted computing approaches addressing the challenges of secure processing [1], however they have not gained the traction and prevalence they may have deserved, yet. Among such solutions are *Trusted Platform Modules* and ARM's *TrustZone* security extensions. Recently, Intel has started shipping its *Software Guard Extensions* (SGX) in many of its newer CPU models released after late 2015. As a result thereof, the widespread availability of trusted computing hardware is foreseeable, catering for the growing demand for trustworthy applications in the field of cloud computing.

SGX is drawing a lot of attention from the research community, bringing forth many innovative, yet academic use cases [2]–[4]. In practice however, SGX still has to pick up traction, even though the technology is available and ready for being used. Recent revelations about SGX enclaves being susceptible to the *Spectre* vulnerability [5], [6] have not helped in improving the reputation of hardware-based trusted computing approaches. However, even in the light of *Spectre*, it is the high complexity [7] of SGX that appears to be a major inhibitor for wide utilization of the technology, requiring profound knowledge in the fields of cryptography, operating systems, and hardware design.

We address this issue by taking a practical perspective, approaching the challenges of trusted computing in distributed scenarios from a *software engineer's point of view*. The goal of this paper is to show how developers can harden their applications today, based on the following contributions:

- We provide a brief overview of the core aspects of SGX.
- We present a helper library[1] assisting developers in overcoming the hurdles of integrating the official SGX *Software Development Kit* (SDK) with their code base.
- In a case study, we demonstrate the steps necessary for porting existing applications to run inside SGX enclaves, using the KISSDB database as an exemplary application.[2]

Hereinafter, this paper is structured as follows: Section II provides background about the Intel *Software Guard Extensions*, after which Section III covers the basics of enclave development. Subsequently, Section IV introduces a helper library that alleviates the process of getting started with SGX. Afterwards, a case study is conducted in Section V, demonstrating the steps necessary to port the KISSDB database to run inside SGX enclaves. Lastly, Section VI takes care of bringing this paper into line with the landscape of related works before a conclusion is reached in Section VII.

II. Background

The *Software Guard Extensions* (SGX) are implemented entirely in the CPU hardware, with the functionality being exposed through an instruction set extension. SGX protects individual software modules in so-called "enclaves", which are encrypted, process-like memory regions containing the code, as well as stack and heap memory of the trusted module. Hence, compared to other trusted computing approaches, the *trusted computing base* (TCB) is very small, comprising only the protected module and the SGX implementation on the CPU.

SGX changes the memory access semantics by introducing a protection scheme inverse to the existing privilege levels [7]. Enclaves are protected from being accessed from privileged code, even from code running in *system management mode* (SMM), as well as from *direct memory access* (DMA) [8]. When enclave memory is loaded from DRAM into CPU cache, the memory contents are decrypted and the CPU enforces isolation by checking whether it is currently executing code of the correct enclave. When a memory page is evicted from the CPU caches, it is encrypted and integrity-protected [9]. Figure 1 illustrates how enclaves relate to existing privilege levels.

[1] https://github.com/ftes/sgx-lib
[2] https://github.com/ftes/kissdb-sgx

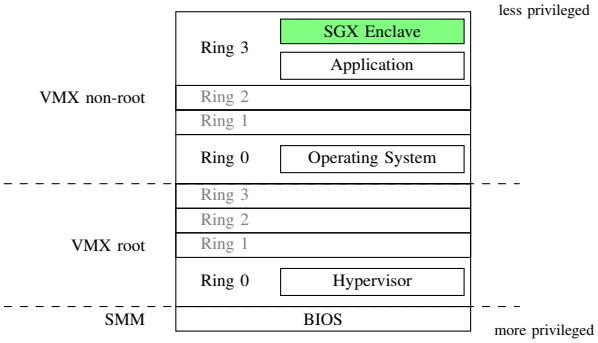

Fig. 1. Intel SGX enclave within the privilege level hierarchy. Illustration adapted from [7].

SGX relies on the untrusted operating system to perform its regular management tasks such as scheduling and memory allocation, including the steps for setting up an enclave. Enclave attestation is used to expose any attempts by a malicious operating system to load a tainted enclave. Each SGX-capable CPU has an embedded cryptographic private key. Using a special group signature scheme, the CPU uses this key to attest the state of an enclave [10]. Attestation can occur locally to set up secure communication channels between different enclaves on the same CPU [11]. Attestation can also be performed remotely, in which case the attestation is not performed entirely in hardware, but relies on so-called "architectural enclaves" [7]. These enclaves increase the software portion of the TCB, yielding a major point of criticism towards SGX.

Code running in an enclave may not execute certain calls, hence illegal calls can only be handled by an untrusted wrapper [12]. Among the set of illegal calls are instructions which may cause a *VMEXIT*, input/output instructions and instructions which require a change in privilege levels (e.g. system calls) [13]. An enclave can use a key derived from its identity (initial measurement) to encrypt any data it wishes to expose to the untrusted world.

Multiple threads can be active at the same time in an enclave [8]. The number of threads must be defined statically before the enclave is initialized. Also, the maximum enclave size in terms of memory must be specified before the enclave is initialized. Even though SGX 2 allows a dynamic number of threads and dynamic memory size, no hardware capable of SGX 2 is available at the time of writing. CPUs capable of SGX 1 have been available since late 2015.[3]

III. Enclave Development

Intel offers a *Software Development Kit* (SDK) for authoring enclaves and integrating them into an application. The SDK is available for both Windows and Linux, providing the following features [13], [14]:

- **Language support:** Currently, C and C++ are the only programming languages supported by the SDK.
- **Interface definition:** An enclave's interface is defined in the *Enclave Definition Language* (EDL).
- **Debugging:** Enclaves can be operated in debug mode, in which case all protection mechanisms are disabled.
- **Simulation mode:** In the absence of SGX hardware, the hardware can be simulated for development purposes.
- **Trusted library:** Helper functions for enclave development, including a subset of the standard C library (e.g. without file input/output), random number generation, cryptographic primitives, key exchange and data sealing.
- **Complete authoring chain:** Enclaves can be compiled and signed so that they can be used in production.

An example EDL interface definition is shown in Listing 1. It is divided into a trusted (E-call) and untrusted (O-call) section. Based on this interface, the SDK generates proxy functions. For all *trusted* functions (E-calls), proxies are generated for the untrusted wrapper. For all *untrusted* functions (O-calls), proxies are generated for the enclave.

The proxy code is necessary for parameter marshalling. The function signature includes additional annotations for the parameters. The annotations show the direction of data flow (`in`, `out`, `user_check`). If *in* (and/or *out*) are specified, the proxy code will copy the parameter by value before calling the function (and/or afterwards). A pass-by-reference can be achieved with *user_check*. Pass-by-value is recommended for security reasons since enclaves cannot rely on untrusted memory to be stable.

```
enclave {
 trusted {
  public void add_secret(int secret);
  public void print_secrets();
  public void test_encryption();
  public void set_key([in, size=128] uint8_t *key);
 };
 from "sgx_lib.edl" import *;

 untrusted {
        };
};
```

Listing 1. The Enclave Definition Language (EDL) file demonstrates the differentiation between functions in the trusted and untrusted world.

The proxy needs to know how much data to copy for pointer arguments. This is handled by the annotations *size, sizefunc* and *count*. The first two define the size of an individual element statically or dynamically. The number of elements can be defined with *count* either statically as a number or dynamically by referencing a different scalar parameter. For a full reference of EDL, see [14].

In addition to the architectural enclaves (remote attestation etc.), Intel also provides some helper enclaves as part of the *Platform Software* (PSW). These enclaves expose functionality such as monotonic counters and trusted time [14]. They can be accessed via trusted library functions included in the SDK.

IV. SGX Helper Library

To enable easier and faster prototyping, a helper library wrapping the SDK has been developed over the course of the case study presented in Section V. The library contains scripts and wrapper functions that make working with the SDK easier by addressing the four central aspects of enclave development discussed hereinafter.

[3]https://github.com/ayeks/SGX-hardware

A. Generation of O-call Proxies

O-call proxies are a necessity whenever required functions of the C library live outside of the enclave. In such cases, a shim is needed inside the enclave to proxy calls to the outside world, as illustrated in Figure 2. Since defining these proxies involves touching several files and repeatedly inserting a similar method signature, we automated this process by introducing the script `add_ocall.sh`. As an example, consider adding a proxy for the `_ftelli64` Windows C library function. The helper script has to be called as exemplified in Listing 2.

```
sgx-lib/add_ocall.sh "int64_t␣_ftelli64([user_check]
    ↪ ␣FILE*␣file);"
```

Listing 2. Example invocation of O-call generation script. The script generates EDL code, trusted header code and trusted and untrusted proxy implementations.

From a developers point of view, it would be desirable for the enclave code to directly link against the C library implementation. Theoretically, the SDK supports this feature by adding a corresponding declaration to a function signature in the EDL file [14]. However, the generated stub in the enclave has a different signature in case the function has a return value. The generated signature of the trusted `fopen` O-call is shown in Listing 3.

```
void fopen(FILE* retVal, const char* filename, const
    ↪  char* mode);
```

Listing 3. For generated O-calls, the SDK modifies the signature and passes return values via pointer parameters.

To use unmodified legacy code in an enclave, the library functions must have the exact same signature. To provide trusted functions with the original signature, the functions had to be overloaded in the enclave, which is not possible in C. The O-call proxies generated by our custom `add_ocall.sh` script bypass this issue by appending the `_ocall` suffix. Additionally, an untrusted proxy implementation is generated, delegating the call to the C library implementation.

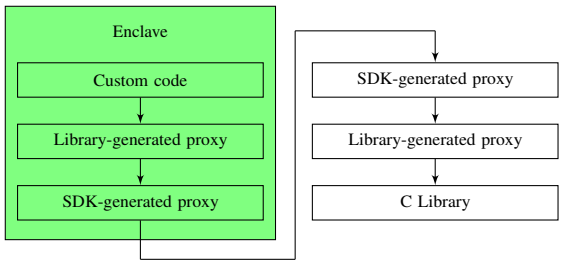

Fig. 2. Interaction of proxies generated by Intel's SDK and the helper library. The SDK proxies deal with parameter handling, whereas the untrusted library proxy delegates to the C library.

B. Error Code Handling

A variety of error codes is defined for SGX, which are frequently used by many SDK functions and the generated proxies. Since looking up error codes manually is tedious and time-consuming, our helper library contains a trusted and untrusted utility function to check return values. The descriptions are scraped from the Intel SDK's `sgx_error.h`.

C. Easy-to-Use Encryption

Even though the SDK includes a cryptography library, some of the SDK's cryptography functions are cumbersome to use. Due to the use of block ciphers and nonces the encrypted/sealed data size is not trivial to determine. In addition to a thin wrapper for data sealing, the helper library provides an extensive wrapper for encryption. Regular encryption must be used whenever the developer needs to be in control of the encryption key. This is often the case when encrypted data from the unsafe world is provided as an input which has not been encrypted by the enclave itself. Also, encryption adds far less overhead than the data sealing operations performed by the SDK as presented in Subsection V-A. Listing 4 demonstrates the corresponding functions exposed by the library.

```
uint32_t get_sealed_data_size(uint32_t
    ↪ plaintext_data_size);
int seal(const void* plaintext_buffer, uint32_t
    ↪ plaintext_data_size, sgx_sealed_data_t*
    ↪ sealed_buffer, size_t sealed_data_size);
int unseal(void* plaintext_buffer, uint32_t
    ↪ plaintext_data_size, sgx_sealed_data_t*
    ↪ sealed_buffer);

uint32_t get_encrypted_data_size(uint32_t
    ↪ plaintext_data_size);
int encrypt(const void* plaintext_buffer, uint32_t
    ↪ plaintext_data_size, sgx_lib_encrypted_data_t
    ↪ * encrypted_buffer, sgx_aes_ctr_128bit_key_t*
    ↪  key);
int decrypt(void* plaintext_buffer, uint32_t
    ↪ plaintext_data_size, sgx_lib_encrypted_data_t
    ↪ * encrypted_buffer, sgx_aes_ctr_128bit_key_t*
    ↪  key);
```

Listing 4. An extract of `sgx_lib_t_crypto.h` demonstrates the simplicity of the encryption functionality provided by the helper library.

Encryption/decryption is done using AES block cipher in counter mode (`sgx_aes_ctr_encrypt` library function). According to NIST, counter mode encryption is efficient because output blocks can be derived in parallel, even before the complete payload is available [15]. NIST also mandates that the counter must be unique over all messages encrypted under the same key. If the counter space is large enough compared to the payload sizes, the encryption key can be re-used if the initial counter – also known as *initialization vector* (IV) or nonce – is chosen at random.

The library-provided `encrypt` function chooses a random IV using SGX's trusted source of randomness by calling `sgx_read_rand`. The IV is added to the encrypted output. The `decrypt` function does the opposite: It reads the nonce from the beginning of the input data and uses it to decrypt the data.

D. Transparent Encryption of Input/Output

The concept of transparent de- and encryption of input/output data has been frequently proposed in related work (see Section VI). Transparent encryption protects data operated on by legacy code without requiring any code modifications. The library implements this concept by intercepting calls to the C library for file input/output. Developers can choose the desired security level at compile time using macros:

- **No security:** Useful during development, file input/output occurs in plain text.
- **Encryption with custom key:** Useful for debugging. A symmetric encryption key is required, which can be set using `set_secure_io_key()`.
- **Data sealing:** This is the default option and seals all input/output to the enclaves identity.

V. Case Study: KISSDB

In this section we present a case study, covering the process of hardening an existing *Database Management System* (DBMS) using SGX. We chose a DBMS because it is an interesting target for trusted computing techniques, as stored data may be sensitive, requiring protection from the infrastructure provider or other tenants. In order to avoid excessively complex code bases, we are using KISSDB[4] as a test subject, a very simple key/value store implemented in plain C using only standard string and file input/output functions. KISSDB stores key/value pairs of fixed size and does not provide any processing, but only offers a put/get interface as well as iterators. Figure 3 illustrates the simplicity of KISSDB's database file layout.

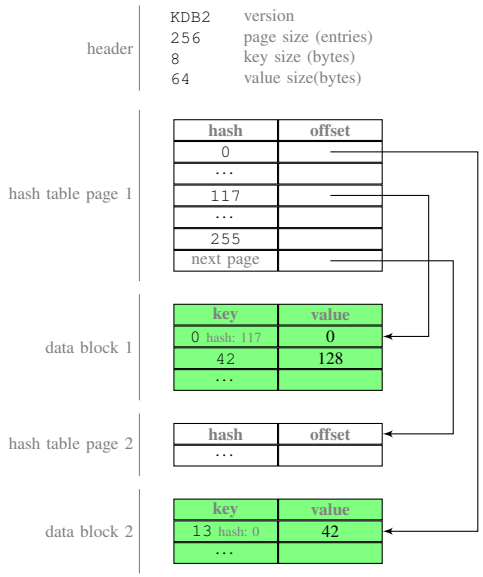

Fig. 3. Visualization of the KISSDB file layout. Meta data (header and hash tables) is not encrypted.

In this case study, Intel SGX is used to protect the data KISSDB operates on. The focus is laid on which part of KISSDB can be extracted, and how it can be secured transparently using the helper libraries features presented in Section IV.

A. Design Decisions

During the process of hardening KISSDB, several design decisions were made concerning enclave design, scope of enclaves, decomposition, handling of meta data, iterators, and cryptographic strategies. In this section, we report these design decisions as well as the reasons for them. The resulting architecture is shown in Figure 4.

[4]https://github.com/adamierymenko/kissdb

1) Enclave Design: With the goal of keeping the enclave as small as possible, we decided to only move application code into the enclave, with a shim C library being employed to utilize the external host C library. Also, this option is the easiest to implement, even though it results in a large enclave interface.

2) Scope of Enclaves: Since KISSDB does not support concurrency, only one enclave needs to be set up per database file. As a result thereof, one enclave is set up per invocation of the `open()` function.

3) Decomposition: KISSDB is not sub-divided into trusted and untrusted functionality. A single enclave is used for all trusted functionality. KISSDB has such a small code base, it would add unnecessary complexity to identify parts that should be pulled out. KISSDB does not support any data processing, which otherwise would have been a likely candidate.

4) Plain Text Meta Data: Meta data (header, hash tables) is stored in plain text. This keeps the required changes to the legacy code base to a minimum as discussed later on in Subsection V-B. This has the following security implications:

- The meta data is not protected, disclosing the number of entries as well as the key and value size.
- The key hashes are not encrypted. If the hash scheme is not cryptographically secure, an attacker may learn information about the hash values.
- Also, if the key space is small or non-uniformly distributed, an attacker may learn information about the keys by pre-computing all (or all likely) key hashes.

5) Iterators: A KISSDB iterator is a cursor which allows iterating through all values. The cursors position is identified by the hash table page number and item offset within that page. Several iterators can exist in parallel for a single database. The iterator is something that inherently belongs to the consumer using the iterator, hence the iterator data (page number and page offset) is stored outside of the enclave. This way the enclave remains stateless. Since the meta data is stored in plain text, this is property does not pose an additional security threat.

6) Encryption versus Data Sealing: Data sealing encrypts the data with a key derived from the enclave's identity. This identity is based only on the initial state (code) of the enclave. Instead, we decided to empower users to specify the encryption key upon creation of a KISSDB instance (`open()` in Figure 4).

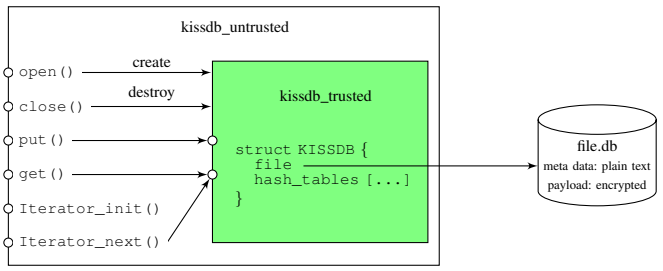

Fig. 4. Visualization of hardened KISSDB architecture.

B. Implementation

This section highlights some implementation details.

1) Proxies in untrusted wrapper: The untrusted wrapper acts as a proxy to the enclave. The `open` and `close` functions must also set up and destroy the enclave. For this purpose, they use the library's helper functions. Listing 5 shows parts of the enclave's interface definition.

```
public void KISSDB_close_ecall();
public int KISSDB_get_ecall([in, size=key_size]
    ↪ const void *key, [out, size=value_size] void
    ↪ *vbuf, unsigned long key_size, unsigned long
    ↪ value_size);
```

Listing 5. Extract of the enclave interface (EDL).

Listing 6 shows the corresponding implementation of the untrusted wrapper which delegates to the SDK-generated proxies.

```
void KISSDB_close(KISSDB *db) {
  KISSDB_close_ecall(db->eid);
  destroy_enclave(db->eid);
  memset(db,0,sizeof(KISSDB));
}

int KISSDB_get(KISSDB *db, const void *key, void *
    ↪ vbuf) {
  int retval;
  KISSDB_get_ecall(db->eid, &retval, key, vbuf, db->
      ↪ key_size, db->value_size);
  return retval;
}
```

Listing 6. Extract of the SDK-generated E-call proxies (C).

2) Plain text meta data: For the case study, the library was configured to transparently encrypt all file input/output (see Section IV). The meta data should be stored in plain text, so a distinction has to be made between meta data and payload. This option requires the least changes to KISSDB's code.

The key and value size (which are also written to the file header) are adapted to include the cryptographic nonce and rounded to the next cipher block size. By keeping the header and hash tables in plain text, KISSDB's file navigation logic does not have to be altered. The offset calculation is preserved.

3) Different trusted/untrusted data structures: The database structure is used both inside and outside of the enclave. Different fields are required inside and outside of the enclave. The hash tables are held only in enclave memory to facilitate encrypting them in future. The untrusted wrapper on the other hand must hold the enclave ID. This is needed for accessing E-calls and destroying the enclave.

Passing the encryption key in the plain via the untrusted wrapper breaks security. This functionality was explicitly excluded from the scope of this case study for the sake of simplicity.

C. Unresolved Issues

To keep the extents of this case study manageable, several aspects had to be excluded from the scope of our report. Without addressing these issues, several critical security issues remain, prohibiting production use. The list of unresolved aspects include attestation and key provisioning, file integrity and freshness, cryptographic hash functions, and file layouts.

1) Attestation and Key-Provisioning: In a production setting, the consumer should attest the enclaves identity and at the same time perform a key exchange with the enclave (see Section III). With the exchanged key, the database encryption key could be provisioned securely.

2) File Integrity and Freshness: Cryptographic mechanisms should be utilized to ensure file integrity. Suitable candidates for ensuring freshness of the file include monotonic counters, such as the ones provided by the Intel SDK. However, this approach also requires additional considerations, e.g. in order to enable migrating of database files between machines.

3) Cryptographic Hash Function: KISSDB uses the *djb2* hash function to compute key hashes. This is not a cryptographic hash function. The hash tables (which are not encrypted) thus may leak information about the keys, even if the key space is large and uniformly distributed. It should be replaced with a cryptographic hash function.

4) File Layout: The file layout is deterministic. If values are added in the same order, the file layout will always be the same. Knowing the consumer's behavior, this weakness opens the door for known plain text attacks. This could occur if a consumer writes a fixed value upon opening the database (e.g. version information).

VI. Related Work

This section presents related work on hardening applications using Intel SGX as a trusted computing solution. First, application-specific approaches are listed. Afterwards, general approaches are described.

A. Application-Specific Approaches

1) Verifiable Confidential Cloud Computing (VC3) [16]: In VC3, secure map-reduce jobs are executed in enclaves using an unmodified version of Hadoop. With Hadoop taking programs (jobs) as input, it is sufficient to protect these jobs. The Hadoop engine runs outside of the enclave. VC3 manages to protect against a malicious Hadoop engine by protecting the integrity of the overall result using only the map-reduce jobs.

2) SecureKeeper [4]: The approach extends SecureKeeper, keeping ZooKeeper data protected within enclaves. To this end, parts of the ZooKeeper functionality are refactored. Furthermore, the authors analyze memory access speeds in SGX and give recommendations on memory management in enclaves.

B. General Approaches

1) Haven [3]: The approach by Microsoft uses SGX to isolate an entire legacy application within an enclave, with the *Drawbridge* library operating system providing all required library and runtime functionalities within the enclave. The Drawbridge LibOS is a trimmed down version of Windows 8 refactored to run as a set of libraries within the picoprocess. While this approach results in comparatively large enclaves, the interface between the enclave and the untrusted world is kept minimal.

2) SCONE [2]: The approach by Arnautov et al. runs Docker containers inside of SGX enclaves, representing the middle way between running an entire operating system inside an enclave (e.g. Haven), and moving only core application logic and shim libraries into the trusted world. A comprehensive classification of these different enclave design approaches is present, as reproduced in Figure 5.

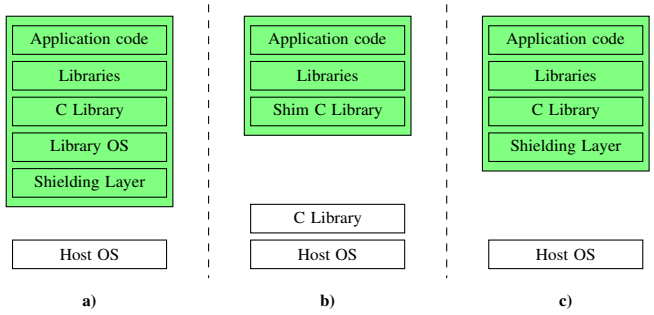

Fig. 5. Enclave design alternatives. The options are: a) Library operating system inside the enclave. b) Minimal enclave size with an external C library. c) Untrusted system calls with an internal C library. The helper library presented in this paper applies approach b). Illustration adapted from [2].

3) Software Partitioning Case Study: The work presented by Atamli et al. [17] evaluates different approaches for decomposing OpenSSL into enclaves, with the most important ones being decomposition by functionality and decomposition by data sets (e.g. tenants).

VII. Conclusion

A strength of the Intel *Software Guard Extensions* (SGX) technology is that it enables a very compact *trusted computing base* (TCB). However, developing enclaves involves a lot of complexity, requiring profound knowledge in the fields of cryptography, operating systems and hardware design.

To unburden software engineers, this paper presented a helper library for alleviated enclave development using Intel *Software Guard Extensions* (SGX), as well as a case study documenting the process of migrating the KISSDB database to run inside SGX enclaves. KISSDB was used a test subject to exemplify some fundamental aspects faced during the process of enabling existing code bases to leverage the secure processing capabilities introduced by SGX. Both the helper library as well as the SGX-enabled port of KISSDB are available for public use (see Section I). Even though some aspects such as attestation have not been addressed in the case study, it offers a solid starting point, effectively helping software engineers to overcome the initial hurdles faced during the development of SGX-enabled applications.

Acknowledgement

The results presented in this paper are based on the Master's thesis by Fredrik Teschke [18]. All implementations are available for public use.

References

[1] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "Trusted execution environments on mobile devices," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1497–1498.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 689–703.

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 8:1–8:26, Aug. 2015.

[4] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper Using Intel SGX," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 14:1–14:13.

[5] LSDS group at Imperial College London, "Sample code demonstrating a Spectre-like attack against an Intel SGX enclave." *GitHub (Website)*. [Online]. Available: https://github.com/lsds/spectre-attack-sgx

[6] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution," 2018, arXiv:1802.09085.

[7] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, p. 86, 2016.

[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[9] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," *IACR Cryptology ePrint Archive*, p. 204, 2016.

[10] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," Intel Corporation, Tech. Rep., 2016.

[11] I. Anati, S. Gueron, S. Johnson, and V. 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*, Aug. 2013.

[12] Intel Corporation, *Intel Software Guard Extensions Programming Reference*, Oct. 2014.

[13] ——, *Intel Software Guard Extensions Developer Guide*.

[14] ——, *Intel SGX Evaluation SDK User's Guide for Windows OS*, Sep. 2015.

[15] L. Chen, J. Franklin, and A. Regenscheid, "Guidelines on hardware-rooted security in mobile devices (draft)," Tech. Rep. 800-38A, 2012.

[16] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 38–54.

[17] A. Atamli-Reineh and A. Martin, *Securing Application with Software Partitioning: A Case Study Using SGX*. Cham: Springer International Publishing, 2015, pp. 605–621.

[18] F. Teschke, "Hardening Applications with Intel SGX," Master's Thesis, Hasso Plattner Institute for Digital Engineering, University of Potsdam, Jul. 2017.