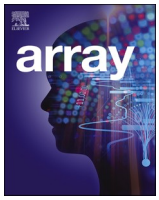
Array 13 (2022) 100118

Contents lists available at ScienceDirect

Array

journal homepage: www.sciencedirect.com/journal/array



TFHE-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments

Lars Brenna [a,*], Isak Sunde Singh [b], Håvard Dagenborg Johansen [c], Dag Johansen [c]

[a] UiT Arctic University of Norway: UiT Norges arktiske universitet Tromso, Troms, Norway
[b] UiT - Arctic University of Norway| UiT Arctic University of Norway: UiT Norges arktiske universitet, Norway
[c] UiT Arctic University of Norway: UiT Norges arktiske universitet, Norway

A B S T R A C T

Fully Homomorphic Encryption (FHE) and Trusted Execution Environ-ments (TEEs) are complementing approaches that can both secure computa-tions running remotely on a public cloud. Existing FHE schemes are, however, malleable by design and lack integrity protection, making them susceptible to integrity breaches where an adversary could modify the data and corrupt the output.

This paper describes how both confidentiality and integrity of remote compu-tations can be assured by combining FHE with hardware based secure enclave technologies. We provide a software library for performing FHE within the Intel SGX TEE, written in the memory-safe programming language Rust to strengthen the internal safety of software and reduce its attack surface.

We evaluate a sample application written with our library. We demonstrate that we can feasibly combine these concepts and provide stronger security guar-antees with a minimal development effort.

1. Introduction

Outsourcing data and computation services to public cloud providers de-mands security mechanisms that can enforce strict data confidenti-ality and in-tegrity regulations. This is particularly important for ap-plications and orga-nizations that prosess sensitive data. Two orthogonal approaches for securing data processing activities are actively being touted as potential game changers: Homomorphic Encryption (HE) and hardware based TEEs.

HE promises computation on encrypted values without revealing their con-tent. Research in the area increased after 2009, when Craig Gentry [1], in his doctoral thesis, described the first technique for achieving FHE nearly 30 years after the idea was conceived [2]. FHE enables outsourcing of many types of computations that previously had to be kept in-house due to confidentiality constraints, including health-data processing, financial processing, and genome research.

Though FHE schemes can provide confidentiality, they cannot pro-vide in-tegrity as all HE schemes are malleable by design. A maliciously altered result is theoretically indistinguishable from the correct one. If the remote service processing data is not trusted for confidentiality it should not be trusted for integrity either. The actual computations performed on data encrypted using FHE will also be visible, which might be unacceptable in some situations as the operations themselves might be secret. As such, FHE only partially solves the problem of outsourcing computation with integrity constraints to public cloud services. While the problems with data integrity are unsolved, FHE has limited practical use.

Trusted Execution Environments (TEEs) have similar ambitions as HE in that they protect the integrity and confidentiality of programs and data hosted on remote and untrusted machines. Trusted Execution En-vironments (TEEs) do this by isolating running processes from the operating system and other con-currently running processes through various hardware facilities. However, it has been shown that existing TEEs, such as the Intel Software Guard Exten-sions (SGX), are suscep-tible to several types of side-channel attacks where an adversary can gain information of the code and data within a secure environ-ment [3–6]. Although most attention in the literature has been given to SGX, some attacks target all processors supporting Simultaneous Multithread-ing (SMT) [7]. Hardware technology that reveals secrets internally thus cannot be relied on to provide highly assured confiden-tiality in public cloud settings. There are some ways to counter this, such as using oblivious primitives like Oblivious RAM (ORAM) [8], which obscures access patterns to prevent infor-mation leakage through side-channels. Oblivious methods do, however, incur significant per-formance overhead to computation.

In this paper, we investigate the intersection between these concepts

* Corresponding author.
E-mail addresses: lars.brenna@uit.no, larslars.brenna@gmail.com (L. Brenna).

https://doi.org/10.1016/j.array.2021.100118
Received 25 February 2021; Accepted 2 December 2021
Available online 28 December 2021

within the stated security context, and propose a hybrid approach that combines the confidentiality strengths of FHE with the integrity strengths of TEEs. We do so using the memory-safe programming language Rust [9]. Using Rust miti-gates large classes of dangerous and common security-related bugs, including memory corruption errors, buffer overflows, uninitialized memory, data races, dereferenced pointers to unallocated memory (e.g., null-pointer dereferencing), and dereferenced pointers causing access violations [10–12]. We evaluate the performance of our hybrid approach by implementing a program that uses FHE both outside and within SGX. By comparing the relative performance difference, we demonstrate that a hybrid approach is feasible in terms of per-formance while retaining more robust security and safety guarantees than using either FHE or SGX separately. To our knowledge, our approach is the first work that combines a TEE with FHE to cover integrity weaknesses of FHE.

2. Background

All HE systems are *malleable* by design since an attacker can transform a ciphertext into a different ciphertext and then have it decrypted to a related plaintext. For instance, consider the following homomorphic encryption scheme:

$$E_k(x) \otimes E_k(y) = E_k(x, y) \tag{1}$$

$E_k(x)$ is the encryption of the plaintext x with the key k, $\times$ is some binary operation between plaintexts, and $\otimes$ is a *lifted* version of $\times$, operating in the ciphertext space. Note that the lifted operator $\otimes$ does not necessarily involve the same operations as the $\times$ operator, which implies it may have a higher complexity. Assume an attacker knows x and y in addition to their encryptions $E_k(x)$ and $E_k(y)$, and there exists some pair (x, y) such that $x \times y \in /\{x, y\}$. The attacker can then compute $E_k(x) \otimes E_k(y)$ to obtain a ciphertext C, that corresponds to the encryption of $x \times y$, which beforehand was assumed to be

different than x and y. Because of this, the attacker has obtained a ciphertext that corresponds to a plaintext, $x \times y$ that they know, but whose ciphertext they have not observed previously.

Although malleable encryption schemes are secure under standard Indistin-guishability under Chosen-Plaintext Attack (IND-CPA), they are not secure un-der Indistinguishability under Adaptive Chosen-Ciphertext Attack (IND-CCA2) [13], as opposed to non-malleable cryptosystems [14]. Furthermore, it has been shown that some encryption schemes that are IND-CPA become insecure when they encrypt their own decryption key [15], often referred to as *circular security*. As.

HE schemes encrypt their decryption key as part of the bootstrapping process, they have circular security properties.

A TEE is an isolated computing environment guaranteeing to protect both code and data loaded within it. Although various definitions of TEEs have been proposed [16–19], Sabt et al. [20] compare these definitions and formalize a description for TEEs by building on the notion of a *separation kernel*, first described by Rushby [21], and define four main security policies.

A TEE should guarantee the authenticity of the executed code, including the integrity of the runtime state, such as CPU registers. It should guarantee the confidentiality of code, data, and runtime state persisted to secondary memory, for instance through encryption. A TEE should have the possibility of provid-ing remote attestation, proving trustworthiness for third-parties. Updates of content within a TEE should be done securely. A TEE should resist all attacks that are performed against main memory. Attacks performed through backdoor security flaws should not be possible. Consequently, a TEE should be secure in a way that even an OS is separated and cannot access nor modify it. These conditions warrant that tasks can be sent to third-parties and executed within a TEE, without requiring trust in that party. This allows for data-sensitive tasks to be outsourced, given they provide a TEE.

Several known methods exist for an adversary to physically attack hard-ware components to extract information. This includes power-monitoring (or power-tweaking) attacks such as Plundervolt [22], acoustic cryptanalysis attacks [23], electromagnetic attacks, and optical attacks. Software-based side-channel attacks range from page-fault based attacks [3], cache-based attacks [4], and interface-based attacks [5], all targeting confidentiality.

The TEE manufacturer must also be trusted to provide sound software and development tools. In the case of Intel SGX, various software systems and a Software Development Kit (SDK) are provided, in addition to the on-chip hardware mechanisms. As of February 2021, the Intel SGX Linux SDK consists of around 360 000 Source Lines of Code (SLoC).

3. The TFHE-rs library

In this paper, we propose a hybrid approach that combines the confidentiality strengths of FHE with the integrity strengths of TEEs, and have developed a Rust library as a proof-of-concept.

The TFHE-rs library combines HE with code executing inside a TEE to provide both confidentiality and integrity. By processing ciphertexts within a TEE, an adversary cannot modify nor even read the ciphertext, eliminating the issue of malleability and thus providing stronger security. For our TEE we use Intel SGX and for homomorphic operations we use the Fast Fully Homomorphic Encryption over the Torus (TFHE) scheme, first described by Chillotti et al. [24].

TFHE is a symmetric lattice-based FHE scheme that works by represent-ing polynomials with coefficients over T, the set of real numbers modulo 1, or R/Z. Chillotti et al. [24] also provide an accompanying library implemen-tation [25], which we will refer to as TFHE-c in this paper. A key benefit of the TFHE-c library is that it is designed to compute on bits. In contrast, other schemes like Homomorphic Encryption Arithmetic of Approximate Num-bers (HEAAN) (also called Cheon-Kim-Kim-Song (CKKS)) [26] and Brakerski- Gentry-Vaikun tanathan (BGV) [27], work with approximate numbers as the plaintext space is within the complex numbers.

The BGV scheme is more appropriate than the others for use with integer arithmetic. This scheme is applicable for building circuits, but is more complex

in use and requires the developer to have considerable knowledge of its inner workings to establish an efficient HE program. An implementation of BGV can also be found in HElib.[1] All of these schemes build on the Learning With Errors (LWE) problem or its ring-variant, Ring Learning With Errors (RLWE). Our TFHE-rs library implementation is heavily inspired by the existing TFHE library [25] [2] and with key parts running within the TEEs of Intel SGX for integrity. It is implemented in Rust rather than C++ to help ensure memory safety.

Moreover, TFHE-rs is written entirely in the safe subset of Rust, and will not compile if the unsafe keyword is used in our codebase. This is enforced by a crate diagnostics attribute, forbid (unsafe_code), which also prevents over-riding the attribute in our crate. However, some of our external dependencies require the use of the unsafe part of Rust to interact with low-level operations, such as providing randomness through assembly instructions.

3.1. Datastructures

In the TFHE-c library, many structures have fields that are strictly pointers to another struct type. In C and C++, this is indistinguishable from an array pointer, unless one looks at the initialization site. Dynamically allocated arrays such as these are equivalent in

[1] https://github.com/homenc/HElib.

[2] We build entirely on the code at this commit https://github.com/tfhe/tfhe/commit/76db530cf736a25115ea0b0ccdb9267b401bb9a7.

```
1  use serde::{Deserialize, Serialize};
2
3  #[derive(Deserialize, Serialize)]
4  struct Ciphertext {
5    data: Vec<i32>,
6  }
```

functionality to the Rust std:vec:Vec type, and are unambiguous in contrast to the original library's implementation.

Structures with pointer-fields in C++ do not specify whether they *own* the data they reference or whether the pointers reference memory given to it dur-ing initialization. For instance in the case of struct Data { val: Vec<i32>} versus struct Data { val: &mut [i32]} (lifetime annotations elided for brevity). This distinction is necessary for Rust, as it tracks ownership. In TFHE-rs, we chose the former as it is more manageable than the latter, and it seems that the TFHE library chose this solution as well, based on their usage. Integer and floating-point data types have direct equivalents in Rust, and are thus translated directly.

The TFHE source code has some structures where a field is a pointer to values within a dynamically-allocated array that a different field in the same structure also references, i.e. self-referential structures. When one moves a value in memory, the referenced value in the self-referential structure is invalidated. This makes them inherently dangerous and thus disallowed by the type system in Rust. As a solution, we chose to remove these fields and access the values directly, at the loss of some readability.

The TFHE library also has some occurrences of void pointers meant to be specialized by a Fast Fourier Transform (FFT) implementation. The use of these pointers is somewhat equivalent to Rust's trait system which allows multiple implementations while providing a stable interface. Since we do not aim to allow multiple implementations of the FFT, we could avoid this abstraction.

3.2. Parameter sets

TFHE-rs supports creating keys of different security levels. Choosing pa-rameters for encryption schemes based on LWE is complicated, as choosing a parameter set with incompatible values might lead to an insecure or slow sys-tem. Our implementation currently supports the two parameter-sets defined in TFHE-c, which have estimated security levels of 80-bit and 128-bit, known as bit security [28]. However, the key size is not directly proportional to the secu-rity level, as in AES, where a security level of 128-bit equates to a 128 bits key size. In TFHE, a security level of 128-bit equates to a $\sim$ 24 MB bootstrapping key [29]. The default parameter set in our library is the 128-bit security version as cryptographers do recommend 128-bit security to be safe until theoretically the year 2090 [28].

3.3. Serialization

All data structures that might need to be transmitted are serializable and deserializable, using the Rust package Serde.[3] Serde designs serialization and deserialization so that any data structure that implements one of two traits can be serialized or deserialized to one of the tens of different serialization for-mats supported. This is unlike the TFHE library, where serialization of data can only be done through specific functions for reading and writing files and streams. These functions are somewhat limited and do not allow the developer to specify the serialization format. In TFHE-rs, a macro allows deriving the implementation automatically, such as (line 3 highlights derive macro):

Implementing these traits allows the user of the library implementation to choose the serialization format that fits the use-case best. As ciphertexts are quite large and contain many integers, a binary format might be best suitable.

3.4. SGX integration

We chose to use Fortanix's Rust EDP[4] rather than one of the several avail-able SDKs. SDKs typically allow low-level control of SGX and the SDK, while the Fortanix Rust EDP aims for an easy way to write programs for SGX by being a platform compilation target.

Fortanix's project is recognized by Rust as a supported target platform and currently has an official tier 2 status.[5] Tier 2 support means code is guaranteed to build on the platform and is part of the language's continuous build testing system. As such, regular Rust programs that do not use multiple processes or rely on OS functionality should work out of the box. These guarantees allow us to easily integrate our FHE library into a program that runs within an SGX enclave and is the main reason why we chose to use the Fortanix Rust EDP for working with SGX.

Our example program using our TFHE-rs implementation and the Fortanix Rust EDP requires no special handling other than specifying the stack and heap size required for the program. The lack of special handling implies that users of our ported library can easily use the hybrid solution of FHE and SGX in the cloud.

4. Evaluation

We evaluate the performance of TFHE-rs using micro benchmarks and by implementing the classic Yao's Millionaires' Problem [30]. Because our key objective is to mitigate the integrity weaknesses in FHE schemes while retaining performance, our experiments focus on the computational overhead incurred by our hybrid approach. As baseline we use the TFHE-lib implementation by Chillotti et al. [25].

TFHE-lib provides several different FFT processors, including FFTW, which claims to be the fastest free FFT implementation available.[6] TFHE-rs uses the RustFFT crate, which does not currently use any Single Instruction, Multiple Data (SIMD) instructions, only pure Rust, and therefore cannot use FFTW. To factor out potential unrelated performance benefits that stem from the usage of FFTW, TFHE-lib is linked with the Nayuki project's portable C implemen-tation.[7] Furthermore, Chillotti et al. [29] provide two benchmarks: one uses the Lagrange half-complex representation internally, and the other does not. We use the latter benchmark as our implementation does not use the Lagrange representation.

4.1. Micro benchmarks

Each micro benchmark was repeated 50 times to obtain averages and

[3] https://crates.io/crates/serde or their homepage https://serde.rs/

[4] https://github.com/fortanix/rust-sgx or their homepage https://edp.fortanix.com/

[5] https://forge.rust-lang.org/release/platform-support.html.

[6] http://www.fftw.org/.

[7] https://www.nayuki.io/page/fast-fourier-transform-in-x86-assembly.

Table 1
Summary of micro benchmarks.

	time (µs)	throughput (KiB s^{-1})
Encryption	1.654 50	73.916
Decryption	0.797 62	148.79
Key Generation	527.67	

stan-dard deviations. These measurements are done without involving SGX. Our measurements are summarized in Table 1.

4.1.1. Encryption and decryption speed

The encryption procedure is slower due to random number generation and allocation, whereas the decryption procedure consists of only simple arithmetic. This implies that the throughput of decryption is also twice as high as for encryption.

4.1.2. Key generation

The key generation procedure generates the secret symmetric key used for encrypting and decrypting data in the TFHE scheme, and the bootstrapping key and the key-switching keys which are required during the bootstrapping process. We collectively name these the bootstrapping keys for brevity, as it is the only process using them. The key generation uses an average of 527.67 µs ± 24.269 µs to generate the keys. As this process depends heavily on random number generation, it is affected by fluctuations in time used to generate num-bers.

4.2. Bootstrapping

The bootstrapping procedure takes an LWE sample as input, along with an output message encoded in the message space and the bootstrapping keys. As shown in Fig. 1, the average execution time of a single bootstrapping procedure is 1.1937 s, significantly higher than the implementation of the original paper taking around 53 ms on similar hardware [29] and improved work leading to around 13 ms [25].

However, the TFHE affords some optimizations we have not implemented in TFHE-rs. Firstly, it uses the Lagrange half-complex representation, which reduces the number of multiplications required in the bootstrapping procedure by nearly a third. It also reduces the number of external products required, the expensive operation performed in the bootstrapping procedure. Secondly, the original implementation uses FFT processors based on SIMD instruction sets such as AVX, providing large speedups. The outliers observed in the figure are, similarly to the outliers in the decryption and encryption procedures, likely related to interactions with other processes using the CPU. As most of the samples fall in a near-identical spot, it is reasonable to assume most results will lie in this range. Additionally, this procedure is deterministic and was benchmarked using the same inputs, so we assume that the outliers can

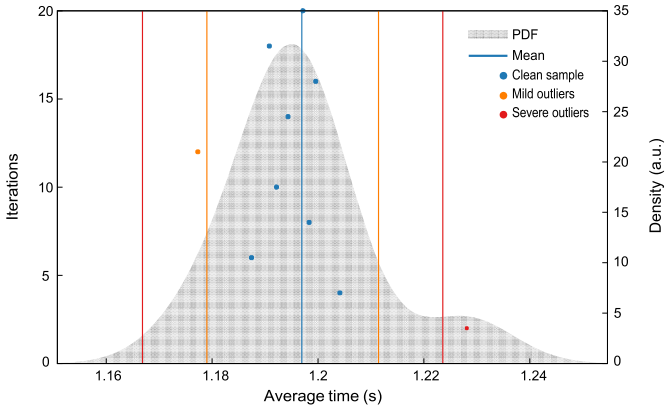


Fig. 1. Detailed view of the estimated PDF of bootstrapping. The mean estimate is 1.1937 *s* and the median is 1.1969 *s* with a std. deviation of 13.234 ms.

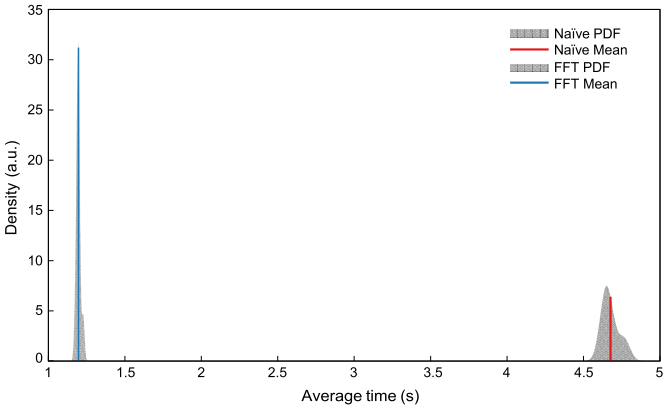


Fig. 2. The execution time of the bootstrapping operation in our implementation with and without the polynomial multiplication based on FFT. The *Naïve* red region marks the näıve implementation of the polynomial multiplication, while the *FFT* blue region marks the optimized version.

be disregarded and that the mean can be used as an estimate.

4.2.1. Comparison between optimized and non-optimized implementation

Fig. 2 shows a comparison of the performance differences between our näıve polynomial multiplication procedure, with time complexity of O n^2, to the FFT-based implementation. We observe that the FFT-based implementa-tion provided a decrease in execution time of 74.408%.

One thing to note is that changing from the 128 to the 80 bit parameter set makes the bootstrapping operation ∼ 2 times faster in the FFT-optimized implementation. This result shows that the parameter set used in encryption has a substantial impact on performance.

Executing the TFHE-c library's benchmark using the FFT implementation written in C without SIMD instructions gives us an average of 614.47 ms for a single bootstrapping operation. Compared to our implementation, this is only ∼ 2 × faster, which is not too bad considering that our objective was not to implement a fast implementation, but rather an implementation that was memory safe, easy to use, and would easily integrate with SGX.

Finally, we also performed the benchmark of the TFHE library with all their optimizations included. We use their spqlios FFT processor with the FMA instruction set extensions and achieved an execution time of 14.771 ms. This number is similar to their findings, but should not be compared directly to ours as it implements several more optimizations.

4.3. Yao's millionaires' problem

Andrew Yao introduced a Secure Multi-Party Computation (SMPC) (com-putations performed by multiple parties with private inputs) problem in 1982 known as Yao's Millionaires' problem [30]. The problem is simple and considers two millionaires, Alice and Bob, wishing to figure out which of them is wealth-ier, while at the same time keeping their actual wealth private. Essentially, the problem aims to calculate the following: $a \leq b$, where a represents the wealth of Alice in some monetary unit, and b represents Bob's wealth in the same unit, while remaining private for the computing party. Yao's Millionaires' Problem is a problem that may seem simple in practice, but it operates under conditions that make it challenging to solve. Thus, it is good to use as proof that a par-ticular system can solve problems in the domain of SMPC. The problem has several solutions, with techniques ranging from oblivious transfer methods [31], private set intersections with HE [32] to FHE.

4.3.1. Socialist millionaire problem

In this modification of Yao's Millionaires' problem, two millionaires wish to compare their wealth and figure out if it is equal or not, while not

disclosing information about their actual wealth to each other [33]. Essentially it aims to calculate the following: $a = b$, where a and b represent the wealth of the two parties, respectively, and are private.

4.3.2. Fused millionaire problem

To increase the computational load in our experiments, we consider a fused problem of Yao's Millionaires' problem and the socialist millionaire problem. In this problem, we aim to figure out the total ordering of two parties' wealth while keeping their actual wealth's private. That is to say when a represents party A's wealth, and b party B's, and both are private, we want to figure out which one of the three cases is true:

- A is wealthier than B
- B is wealthier than A
- A and B have equal wealth

We solve this problem by encrypting the values using the TFHE scheme. Technically, this requires two parties to compute on encrypted data jointly using a multi-key setup. Multi-key HE is possible, as shown in Ref. [34], which conveniently turns the TFHE scheme we use into a multi-key TFHE scheme.

Using the multi-key TFHE scheme, the two parties would encode and encrypt their respective amount of wealth, transmit them to a computing node, where their partial evaluation keys are combined, and then the comparisons computed.

4.3.3. Implementation of the fused millionaire problem

We start by producing the binary decomposition of the two values. We use two 32-bit signed integers for this purpose. For each of the values, we decompose them into bytes in big-endian order, then decompose those into the individual bits. We use big-endian as we implemented the circuits we use to work on big-endian values. Then each bit is individually encrypted with our TFHE implementation. This results in two pairs of 32 ciphertexts representing the encryption of the two values. In a multi-key setup, the two parties perform these actions separately after completing a key-exchange protocol. Note that our implementation of the TFHE scheme does not support multi-key setups as we based it on an implementation that also did not support it. However, supporting it would only necessitate adding a key combination step that scales linearly with the number of parties.

After this, the setup phase is complete. We then perform the comparison circuit equivalent to computing $a \leq b$ and the equality circuit equivalent to $a = b$, both computing on a list of encrypted bits (two pairs of 32-bits) producing encrypted results. These two circuits are independent and are thus evaluable in parallel, although our implementation performs them sequentially.

4.4. TFHE-rs with and without SGX

Next, we evaluate and compare the performance of TFHE-rs with and with-out the use of SGX. We repeat each experiment 25 times, timing only the relevant sections. Running with 80-bit security, TFHE-rs with SGX finished with an arithmetic mean of 90.504 s and a standard deviation of 0.602 86 s while the FHE-only version finished in 116.08 s and a standard deviation of 2.3548 s. These results indicate that TFHE-rs is approximately 28% faster with SGX.

There is known overhead associated with SGX memory encryption and pag-ing. However, we explain the performance improvement between the two ver-sions of TFHE-rs by how an SGX enclave handles memory. For this, we profiled our non SGX program using the 128-bit security parameter set, which is the one that uses most memory, with the memory profiler for Linux.[8] The observed memory usage over time is shown in Fig. 3, while the rate of allocations of deallocations are shown in Fig. 3b and c. From the figures we can see that the program constantly consumes around 100 MiB of memory, and has a rate of allocations and deallocations of about 100 000 per second.

As can be seen from the memory profiling, the program does a significant amount of allocations and deallocations. Regular Rust code on Linux relies on the standard libc malloc[9] allocator. However, the Fortanix Rust EDP platform, which our SGX implementation builds on, uses the dlmalloc[10] allocator. This allocator emphasizes minimizing memory usage and fragmentation, something that would occur when a program regularly allocates and deallocates memory, like our program does. Although each allocation in TFHE-rs is relatively small, around a few kibibytes each, relating to the vectors used for numeric processing, and never above 1 MiB, we wanted to see if a different allocator could account for the 28% speedup in execution time. We therefore modified TFHE-rs to use the dlmalloc allocator and repeated the tests. With the modified TFHE-rs we obtained an average of 93.556 s with a standard deviation of 1.4932 s for the program with 80-bit security and an average of 160.61 s and 3.9251 s. This result is only approximately 3% slower than TFHE-rs with SGX. The combined results are shown in Fig. 4.

As can be observed from the figure, using 128-bit security significantly im-pacts performance. The hybrid program executes roughly 72.5% slower, and the FHE only using dlmalloc for allocation is 71.7% slower than the ones with 80-bit security. As mentioned, this is because ciphertexts in the TFHE scheme grow substantially in size with increased security and thus increases the required computation. As for the performance difference with and without SGX, the ex-ecution time difference of only 3% is low. Thus, a user would benefit from using the SGX version, which covers the integrity weaknesses of the FHE.

Why the SGX version is faster than the program using only FHE is unclear. However, the standard deviations measured for the FHE-only version are higher, as seen in Fig. 4, which implies that the performances could be more similar than they appear. Another reason could be because of automatic frequency scaling of the CPU. However, turning this off yielded the same re-sults. We witnessed no other behavior, including syscalls, that could explain the performance differences. The syscalls mostly consisted of memory alloca-tion calls, which are handled by the enclave memory manager. Our program is single-threaded; it does not use any SIMD instructions, nor include randomness during the measured execution times. If we are to speculate, the performance gain might be a result of the SGX SDK optimizing some situations that are normally not possible to optimize, due to the complexity of the OS and process interactions.

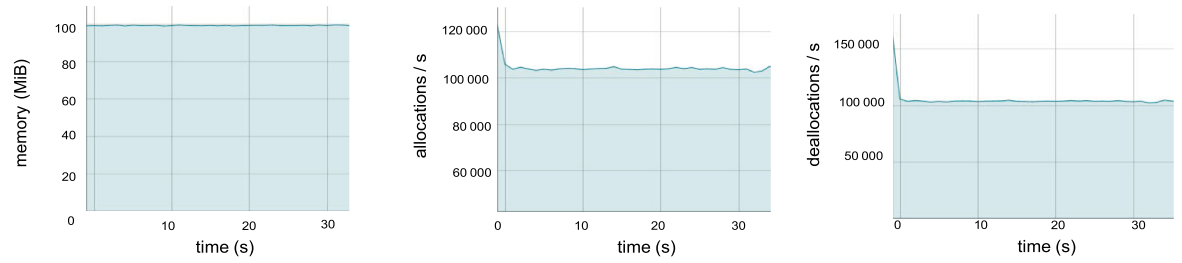
5. Related work

Drucker and Gueron [35] state that most secure cloud database solutions tend to provide confidentiality and integrity of data by using either a TEE or HE. They show that combining a TEE and using HE is feasible and does not need to rely on the TEE for confidentiality purposes. They compare their work to CryptDB [36] and MrCrypt [37], which both use Partially Homomorphic En-cryption (PHE), but lack integrity security for both code and data. Drucker and Gueron combine the PHE scheme Paillier [38] and SGX, where SGX pro-vides integrity of code and data (in addition to some confidentiality guarantees, side-channel attacks aside). The Paillier cryptosystem ensures data is private and provides confidentiality, even within the enclave. The combination allows the system to place less trust in Intel, as the Paillier cryptosystem guarantees confidentiality for the encrypted data while allowing some computations. In their experiments, they only experience

[8] https://github.com/koute/memory-profiler.

[9] https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html.

[10] http://gee.cs.oswego.edu/dl/html/malloc.html.



(a) Memory usage during the first part of the program, at a constant usage of around 100 *MiB*.

(b) Memory allocations during the first part of the program, at a constant level around 100 *k allocs/s*.

(c) Memory deallocations during the first part of the program, at a constant level around 100 *k deallocs/s*.

Fig. 3. Memory usage characteristics for FHE with the default system allocator.

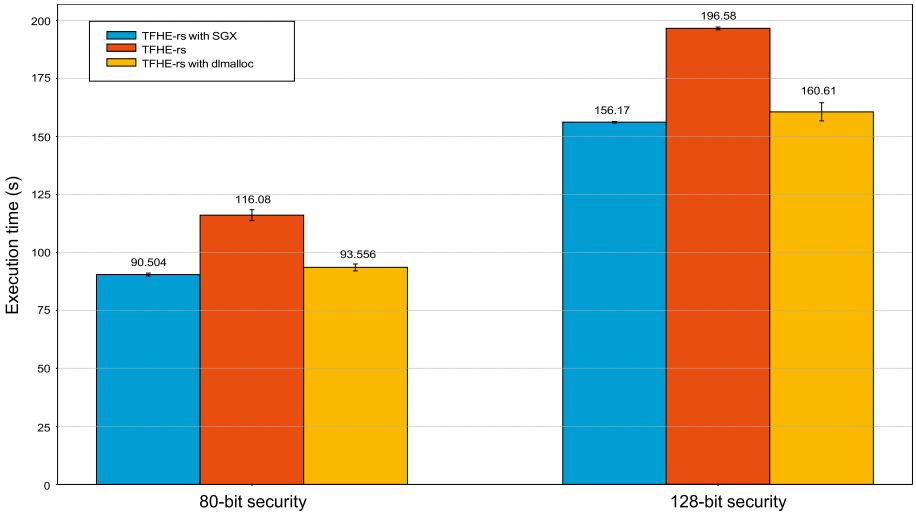


Fig. 4. Execution times of our fused millionaire problem. Experiments were performed 25 times and respective standard deviations are represented by the vertical error bars.

around 1.7 × performance slowdown compared to not running in SGX with PHE. Execution time grows linearly with the number of summed entries, as expected.

SAFETY [39] combines PHE and SGX to securely process genome data to identify genetic risk factors for diseases. This data is quite sensitive and often comes with strict regulations on how to process and store it. By combining Paillier encryption with SGX they created a system which achieved a 4.8 × speedup compared to existing secure computing techniques.

TEEFHE [40] is an example of combining FHE with SGX by performing the bootstrapping step within SGX. They use the BGV [27] scheme implemented in Simple Encrypted Arithmetic Library (SEAL) and modify the library to run within SGX. They distribute the work across several nodes, where some nodes process the ciphertexts using homomorphic operations in untrusted environ-ments. When nodes require the bootstrapping procedure, they transmit them to a node with the SEAL library running within SGX. SGX enclaves perform encryption and decryption, preserving data and code integrity and confidential-ity, as they do not consider side-channel attacks. Decrypting and encrypting a ciphertext removes the encoded noise and *refreshes* the ciphertext, effectively doing the same as a bootstrapping operation, but at a lower cost. As the un-trusted compute servers perform computations on the encrypted data, they do not preserve data integrity in the case of an attack.

A large corpus of work exists that address the confidentiality problems re-lated to side-channel attacks and cloud hosted computations. Chen et al. [41] propose a software framework that detects side-channel attacks by a privileged attacker, such as a malicious or virus-infected OS. Some types of side-channel attacks that exploit access-pattern information leakage can be protected against using techniques such as ORAM [8]. ORAM can be seen as a compiler that transforms memory accesses of a program into a program where the distribu-tion of memory accesses differs (is independent) from the original program while preserving the semantics of the program. Path ORAM [42] improves upon reg-ular ORAM and has a low space overhead and in some cases, asymptotically improved performance compared to earlier work. Circuit ORAM [43] further improves the techniques and gives an implementation with a complexity near the theoretical lower-bound.

ZeroTrace [44] is an example of oblivious primitives in action. Security is strengthened against access-pattern side-channel attacks in SGX using a block-level memory controller to hide access patterns. Both Path ORAM and Circuit ORAM are implemented and gave in some situations only a logarithmic over-head in bandwidth costs between enclave code and ORAM servers. ZeroTrace mitigates considerable weaknesses in SGX as it protects against shared resource and page-fault related attacks by converting programs into oblivious represen-tations. Another example of oblivious memory primitives in SGX is Oblix [45], an oblivious search index. The authors introduce something they call *doubly-oblivious* techniques, as it ensures that accesses to external servers as well as the

ORAM clients internal memory are oblivious. An ORAM client is a pro-gram which accesses an external resource (an ORAM server) through oblivious techniques. These doubly-oblivious techniques ensure that even if an adversary were to observe accesses to a client's internal memory, it could learn no informa-tion on the data. Oblix additionally designs oblivious algorithms that are more efficient than earlier work and implements a contact number discovery service akin to Signal's service implemented in SGX as a demonstration [46]. They use different techniques than Signal, but achieve speedups ranging from $\sim 9\times$ to $\sim 140\times$ faster while strengthening security at the same time, by utilizing the doubly-oblivious techniques.

The CacheOut [6] attack exploits the fact that hardware-cache that is flushed and overwritten can still be recovered. CacheOut can even selectively choose parts of data to leak with relatively high efficiency, unlike previous attacks where the attacker could only observe the leaked data the CPU enclave was currently accessing. This attack requires hardware fixes and proves once again that SGX enclaves do not fully protect the confidentiality of data and code in enclaves, and that other protective measures are required. SGAxe [47] exploits the CacheOut attack to compromise both the confidentiality but also the integrity of an en-clave's memory. The attack extracts the secret attestation key used by enclaves to prove that they are genuine, meaning a malicious attacker such as a malicious cloud vendor could pass a fake enclave for a real one, tricking the client. This attack compromises many security guarantees needed in our hybrid TEE and FHE solution, but most importantly, it compromises the integrity guarantees required for our system to work.

The Load Value Injection (LVI) attack [48] builds on the Meltdown [49] attack to inject the attacker's data into the victim's data stream. This vulnera-bility breaches the data integrity guarantees that SGX should provide as it opens the possibility for the victim's code to execute on the attacker's data, breaking all the correctness guarantees of the user's code. Additionally, it might lead to software crashes by injecting data structured in a format the victim's code did not expect. Patching LVI necessitates extensive software patches, estimated to impact per-formance of SGX enclaves between 2–19 $\times$.

6. Concluding remarks

This paper presented and evaluated the TFHE-rs library for per-forming FHE, specifically the TFHE [29] scheme, written in pure memory-safe Rust. It embeds in SGX as a single dependency by using the Fortanix Rust EDP. Our TFHE-rs implementation was based on an existing library written in a mix of C and C++ [25]. There is no user-required configuration apart from the minimum required for creating an SGX enclave. TFHE-rs provides pre-made circuits to make it easy for users to create common circuits and built-in serialization and deserialization support for easy transfer to and from enclaves.

We evaluated the performance characteristics of TFHE-rs with and without an SGX enclave and found that the performance overhead is negligible. The evaluation showed that using TFHE-rs with SGX is 3% faster than a version of TFHE-rs without SGX. This result is not in line with what we conjectured, which was that TFHE-rs with SGX should be slower. Based on our experience, we conjecture that specific memory management implementations particularly affects performance. The default system allocator on Linux (libc's malloc) was 28% slower than the dlmalloc allocator used by the Fortanix Rust EDP in the SGX setup. As such, a system with a similar setup to ours should emphasize low memory usage and experiment with different allocators to ensure that they stay within the memory limits imposed by SGX. However, the measured stan-dard deviation does account for most of the performance difference, and the benchmarks themselves take long enough for this discrepancy to be due to envi-ronmental factors in our experimental setup (i.e., due to system load). Overall, this is a positive result, as our hybrid solution is both more secure and faster.

Thus, we conclude that using FHE operations within SGX, written in the memory-safe language Rust, is both feasible and provides several additional se-curity guarantees, given that the developer ensures a reasonable memory usage.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is funded, in part, by the Norwegian Research Council grants number 263248 and 275516.

References

[1] Gentry C. A fully homomorphic encryption scheme, Ph.D. thesis. Stanford University; 2009. https://crypto.stanford.edu/craig/.
[2] Rivest RL, Adleman L, Dertouzos ML. On data banks and privacy homomorphisms, foundations of secure computation. Academia Press; 1978. p. 169–79.
[3] Xu Y, Cui W, Peinado M. Controlled-channel attacks: deterministic side channels for untrusted operating systems. In: 2015 IEEE sympo- sium on security and privacy; 2015. p. 640–56. https://doi.org/10.1109/SP.2015.45.
[4] Brasser F, Müller U, Dmitrienko A, Kostiainen K, Capkun S, Sadeghi A-R. Software grand exposure: SGX cache attacks are practi- cal. In: 11th USENIX workshop on offensive technologies (WOOT 17), USENIX association, vancouver, BC; 2017. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.
[5] Wang J, Cheng Y, Li Q, Jiang Y. Interface-based side channel attack against Intel SGX. CoRR abs/1811.05378 2018. http://arxiv.org/abs/1811.05378. arXiv:1811.05378.
[6] van Schaik S, Minkin M, Kwong A, Genkin D, Yarom Y. Cache- out: leaking data on Intel CPUs via cache evictions. 2020. https://cacheoutattack.com/.
[7] Percival C. Cache missing for fun and profit. 2009.
[8] Goldreich O. Towards a theory of software protection and simulation by oblivious RAMs, in: proceedings of the nineteenth annual ACM sympo- sium on theory of computing. STOC '87, ACM, New York, NY, USA 1987:182–94. https://doi.org/10.1145/28395.28416. http://doi.acm.org/10.1145/28395.28416.
[9] Matsakis ND, Klock FS. The rust language. ACM SIGAda - Ada Lett 2014;34:103–4.
[10] van der V, Veen N, dutt Sharma L, Cavallaro H Bos. Memory errors: the past, the present, and the future. In: Balzarotti D, Stolfo SJ, Cova M, editors. Research in attacks, intrusions, and defenses. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 86–106.
[11] Miller M. Trends, challenges, and strategic shifts in the soft- ware vulnerability mitigation landscape. URL. 2019. PjbGojjnBZQ?t=848, https://youtu.be/.
[12] Shen Y, Chen Y, Chen K, Tian H, Yan S. To isolate, or to share? That is a question for Intel SGX, in: proceedings of the 9th asia-pacific workshop on systems. URL. In: APSys '18, association for computing machinery, New York, NY, USA; 2018. https://doi.org/10.1145/3265723.3265727. https://doi.org/10.1145/3265723.3265727.
[13] Das A, Dutta S, Adhikari A. Indistinguishability against chosen cipher- text verification attack revisited: the complete picture. In: Susilo W, Reyhanitabar R, editors. Provable security. Berlin, Heidelberg: Springer Berlin Heidelberg; 2013. p. 104–20.
[14] Dolev D, Dwork C, Naor M. Non-malleable cryptography. SIAM J Comput 2001;30. https://doi.org/10.1145/103418.103474.
[15] Acar T, Belenkiy M, Bellare M, Cash D. Cryp- tographic agility and its relation to circular encryp- tion. URL. In: EUROCRYPT 2010, springer verlag; 2010 [cryptographic-agility-and-its-relation-to-circular-encryption/], https://www.microsoft.com/en-us/research/publication/.
[16] Advanced trusted environment: omtp tr1 v1.1, OMTP limited. 2009.
[17] Garfinkel T, Pfaff B, Chow J, Rosenblum M, Boneh D. Terra: a virtual machine-based platform for trusted computing, in: proceedings of the nineteenth ACM symposium on operating systems principles. In: SOSP '03, ACM, New York, NY, USA; 2003. p. 193–206. https://doi.org/10.1145/945445.945464. http://doi.acm.org/10.1145/945445.945464.
[18] specificationsdevice.asp TEE system architecture. 2011. Online; accessed 19-May-2020, http://www.globalplatform.org/.
[19] Vasudevan A, McCune JM, Newsome J. Trustworthy execution on mo- bile devices, Springer Publishing Company. 2013.
[20] Sabt M, Achemlal M, Bouabdallah A. Trusted execution environment: what it is, and what it is not. In: 2015 IEEE trustcom/BigDataSE/ISPA, vol. 1; 2015. p. 57–64. https://doi.org/10.1109/Trustcom.2015.357.
[21] Rushby JM. Design and verification of secure systems, in: proceedings of the eighth ACM symposium on operating systems principles. In: SOSP '81, ACM, New York, NY, USA; 1981. p. 12–21. https://doi.org/10.1145/800216.806586. http://doi.acm.org/10.1145/800216.806586.
[22] Murdock K, Oswald D, Garcia FD, Van Bulck J, Gruss D, Piessens F. Plundervolt: software-based fault injection attacks against Intel SGX. In: 41st IEEE symposium on security and privacy (S&P'20); 2020.

[23] Genkin D, Shamir A, Tromer E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Garay JA, Gennaro R, editors. Advances in cryptology – CRYPTO 2014, springer berlin heidelberg. Berlin: Heidel- berg; 2014. p. 444–61.
[24] Chillotti I, Gama N, Georgieva M, Izabachène M. TFHE: fast fully homomorphic encryption over the Torus. J Cryptol 2019;33. https://doi.org/10.1007/s00145-019-09319-x.
[25] Chillotti I, Gama N, Georgieva M, Izabachène M. TFHE: fast fully homomorphic encryption library. Technical Report August 2016 [Https://tfhe.github.io/tfhe/].
[26] Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In: Takagi T, Peyrin T, editors. Advances in cryptology – ASIACRYPT 2017, springer international pub- lishing, cham; 2017. p. 409–37.
[27] Brakerski Z, Gentry C, Vaikuntanathan V. Fully homomorphic encryp- tion without bootstrapping. Cryptology ePrint Archive, Report 2011/277 2011. https://eprint.iacr.org/2011/277.
[28] Lenstra AK. Key lengths. Contribution to the handbook of information security. 2010.
[29] Chillotti I, Gama N, Georgieva M, Izabachène M. Faster fully homo- morphic encryption: bootstrapping in less than 0.1 seconds. In: Cheon JH, Takagi T, editors. Advances in cryptology – asiacrypt 2016. Berlin, Heidelberg: Springer Berlin Heidelberg; 2016. p. 3–33.
[30] Yao AC. Protocols for secure computations, in: proceedings of the 23rd annual symposium on foundations of computer science. In: SFCS '82, IEEE computer society, Washington, DC, USA; 1982. p. 160–4. https://doi.org/10.1109/SFCS.1982.88. https://doi.org/10.1109/SFCS.1982.88.
[31] Ioannidis I, Grama A. An efficient protocol for Yao's millionaires' problem, in: 36th annual Hawaii international conference on system sciences. Proceedings of the 2003;2003:6.
[32] Lin H-y, Tzeng W-g. An efficient solution to the millionaires' problem based on homomorphic encryption. In: In ACNS 2005, vol. 3531; 2005. p. 456–66. of Lecture.
[33] Jakobsson M, Yung M. Proving without knowing: on oblivious, agnos- tic and blindfolded provers. In: Koblitz N, editor. Advances in cryptology — crypto '96. Berlin, Heidelberg: Springer Berlin Heidelberg; 1996. p. 186–200.
[34] Chen H, Chillotti I, Song Y. Multi-key homomophic encryption from TFHE. Cryptology ePrint Archive, Report 2019/116 2019. https://eprint.iacr.org/2019/116.
[35] Drucker N, Gueron S. Achieving trustworthy homomorphic encryption by combining it with a trusted execution environment. JoWUA 2018;9:86–99.
[36] Popa RA, Redfield CMS, Zeldovich N, Balakrishnan H. CryptDB: protecting confidentiality with encrypted query processing, in: pro- ceedings of the twenty-third ACM symposium on operating sys- tems principles. In: SOSP '11, ACM, New York, NY, USA; 2011. p. 85–100. https://doi.org/10.1145/2043556.2043566. http://doi.acm.org/10.1145/2043556.2043566.
[37] Tetali SD, Lesani M, Majumdar R, Millstein T. MrCrypt: static anal- ysis for secure cloud computations, in: proceedings of the 2013 ACM SIGPLAN international conference on object oriented programming sys- tems languages & applications. In: OOPSLA '13, ACM, New York, NY, USA; 2013. p. 271–86. https://doi.org/10.1145/2509136.2509554. http://doi.acm.org/10.1145/2509136.2509554.
[38] Paillier P. Public-key cryptosystems based on composite degree residu- osity classes. In: Stern J, editor. Advances in cryptology — eurocrypt '99. Berlin, Heidelberg: Springer Berlin Heidelberg; 1999. p. 223–38.
[39] Sadat MN, Aziz MMA, Mohammed N, Chen F, Wang S, Jiang X. SAFETY: secure gwAs in federated environment through a hYbrid solu- tion with Intel SGX and homomorphic encryption. URL CoRR abs/1703.02577 2017. arXiv:1703.02577, http://arxiv.org/abs/1703.02577.
[40] Wang W, Jiang Y, Shen Q, Huang W, Chen H, Wang S, Wang X, Tang H, Chen K, Lauter KE, Lin D. Toward scalable fully Ho- momorphic encryption through light trusted computing assistance. URL CoRR abs/1905.07766 2019. arXiv:1905.07766, http://arxiv.org/abs/1905.07766.
[41] Chen S, Zhang X, Reiter MK, Zhang Y. Detecting privileged side-channel attacks in shielded execution with DéJ'a vu, in: pro- ceedings of the 2017 ACM on asia conference on computer and com- munications security. In: ASIA CCS '17, ACM, New York, NY, USA; 2017. p. 7–18. https://doi.org/10.1145/3052973.3053007. http://doi.acm.org/10.1145/3052973.3053007.
[42] Stefanov E, Shi E. Path O-RAM: an extremely simple oblivious RAM protocol. URL CoRR abs/1202.5150 2012. http://arxiv.org/abs/. 1202.5150. arXiv:1202.5150.
[43] Wang X, Chan H, Shi E. Circuit ORAM: on tightness of the goldreich- ostrovsky lower bound. In: Proceedings of the 22Nd ACM SIGSAC con- ference on computer and communications security, CCS '15. New York, NY, USA: ACM; 2015. p. 850–61. https://doi.org/10.1145/2810103.2813634. http://doi.acm.org/10.1145/2810103.2813634.
[44] Sasy S, Gorbunov S, Fletcher CW. ZeroTrace : oblivi- ous memory primitives from Intel SGX. URL. In: 25th annual net- work and distributed system security symposium, NDSS 2018. San Diego: California, USA; February . p. 2018. 25/2018/02/ndss2018_02B-4_Sasy_paper.pdf, http://wp.internetsociety.org/ndss/wp-content/uploads/sites/.
[45] Mishra P, Poddar R, Chen J, Chiesa A, Popa RA. Oblix: an efficient oblivious search index. In: 2018 IEEE symposium on security and privacy (SP); 2018. p. 279–96.
[46] Marlinspike M. Technology preview: private contact discovery for Signal. 2017. https://signal.org/blog/private-contact-discovery/.
[47] van Schaik S, Kwong A, Genkin D, Yarom Y. SGAxe: how SGX fails in practice. 2020. https://sgaxeattack.com/.
[48] Van Bulck J, Moghimi D, Schwarz M, Lipp M, Minkin M, Genkin D, Yuval Y, Sunar B, Gruss D, Piessens F. LVI: hijacking transient ex- ecution through microarchitectural load value injection. In: 41th IEEE symposium on security and privacy (S&P'20); 2020.
[49] Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M. Meltdown: reading kernel memory from user space. In: 27th USENIX security sym- posium (USENIX security 18); 2018.