[Array 13 (2022) 100118](https://doi.org/10.1016/j.array.2021.100118)

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | Contents lists available at ScienceDirect |  |
| Array |
| journal homepage: [www.sciencedirect.com/journal/array](https://www.sciencedirect.com/journal/array) |

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

Lars Brennaa,\*, Isak Sunde Singhb, Håvard Dagenborg Johansenc, Dag Johansenc

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](mailto:lars.brenna@uit.no), [larslars.brenna@gmail.com](mailto:larslars.brenna@gmail.com) (L. Brenna).

<https://doi.org/10.1016/j.array.2021.100118>   
[Received 25 February 2021; Accepted 2 Dece](https://doi.org/10.1016/j.array.2021.100118)mber 2021   
Available online 28 December 2021   
2590-0056/© 2021 Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

*L. Brenna et al.*  *Array 13 (2022) 100118*

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 lan-guage 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 trans-form a ciphertext into a different ciphertext and then have it decrypted to a related plaintext. For instance, consider the following homomorphic encryption scheme:

*Ek*(*x*) ⊗ *Ek*(*y*) = *Ek*(*x, y*) (1)

*Ek*(*x*) is the encryption of the plaintext *x* with the key *k*, × is some binary operation between plaintexts, and ⊗ is a *lifted* version of × , operating in the ciphertext space. Note that the lifted operator ⊗ does not necessarily involve the same operations as the × operator, which implies it may have a higher complexity. Assume an attacker knows *x* and *y* in addition to their encryptions *Ek*(*x*) and *Ek*(*y*), and there exists some pair (*x, y*) such that *x* × *y* ∈*/*{*x, y*}. The attacker can then compute *Ek*(*x*) ⊗ *Ek*(*y*) to obtain a ciphertext C, that corresponds to the encryption of *x* × *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* × *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 cryp-tosystems [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.

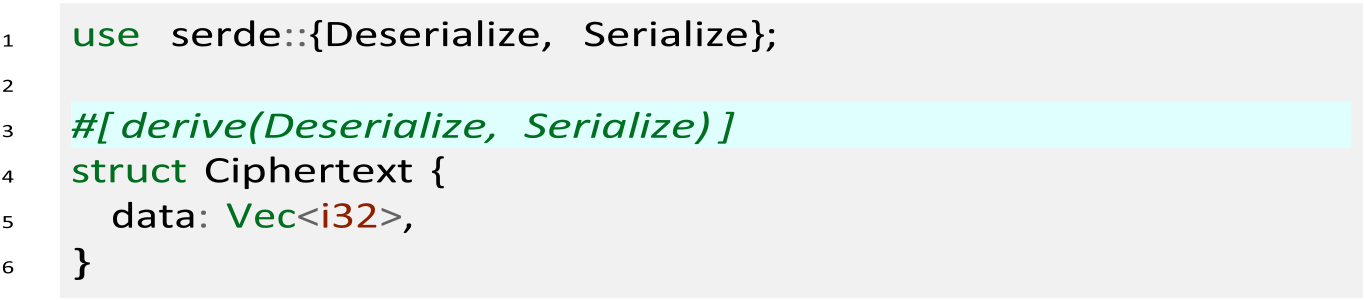
HE schemes encrypt their decryption key as part of the bootstrapping process, they have circular security properties.

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 def-initions 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-

2

*L. Brenna et al.*  *Array 13 (2022) 100118*



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

*own* the data they reference or whether the pointers reference memory Structures with pointer-fields in C++ do not specify whether they

given to it dur-ing initialization. For instance in the case of struct Data { val: Vec*<*i32*>*} versus struct Data { val: &mut [i32]} (lifetime anno-tations 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*

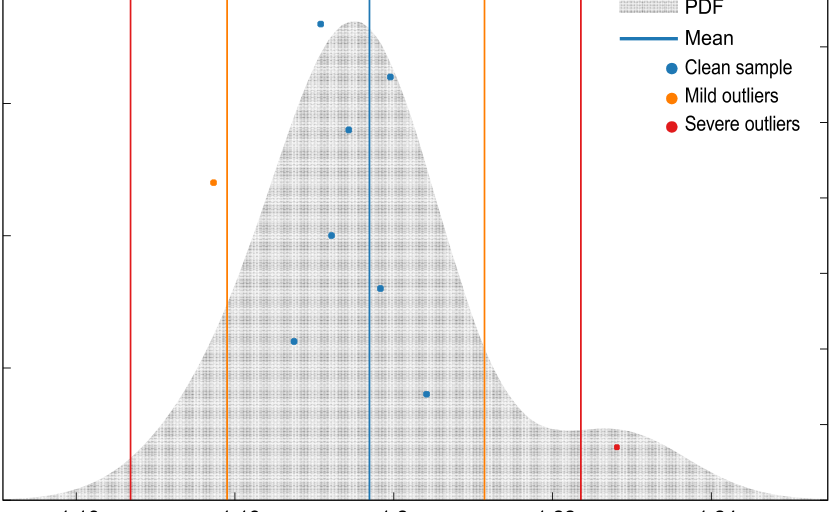
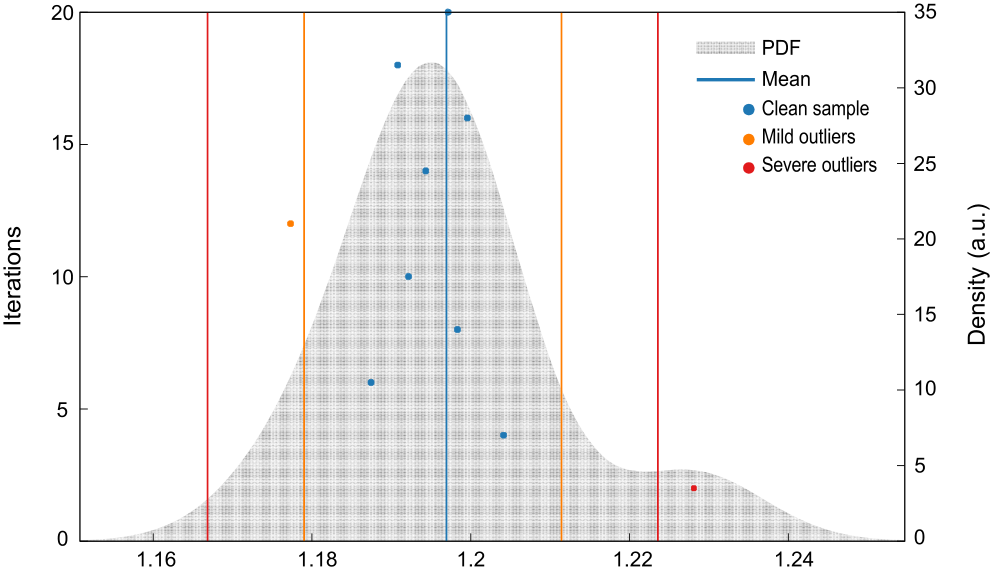
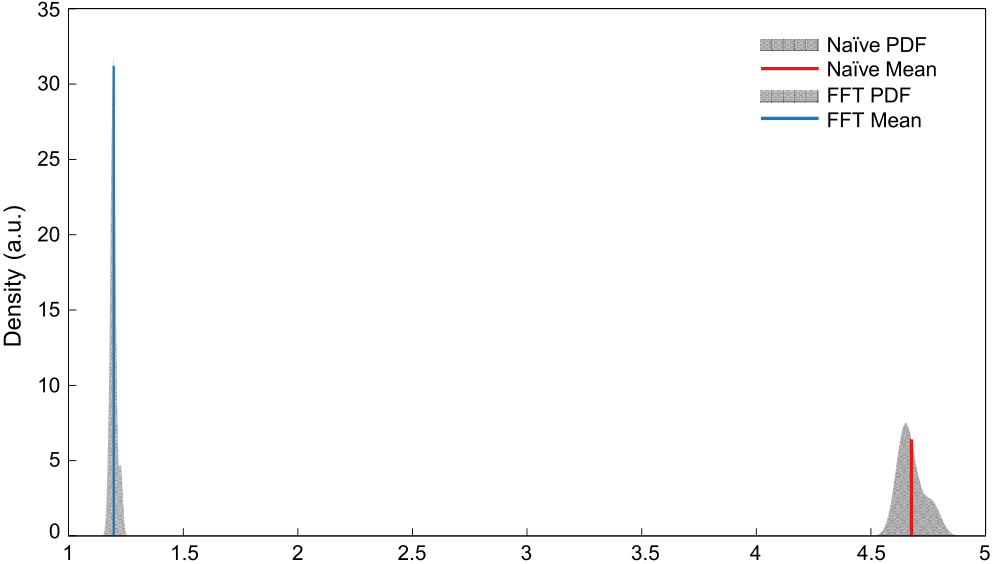
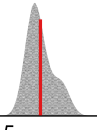
TFHE-rs supports creating keys of different security levels. Choosing pa-rameters for encryption schemes based on LWE is complicated, as choosing a parameter set with incompatible values might lead to an insecure or slow sys-tem. Our implementation currently supports the two parameter-sets defined in 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 ~ 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 theo-retically 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 seri-alization 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 li-brary, 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 serial-ization format. In TFHE-rs, a macro allows deriving the implementation automatically, such as (line 3 highlights derive macro):

3 <https://crates.io/crates/serde> or their homepage <https://serde.rs/>

3



*L. Brenna et al.*  *Array 13 (2022) 100118*

**Table 1**

Summary of micro benchmarks.

|  |  |  |
| --- | --- | --- |
|  | time (μs) | throughput (KiB s−1) |
| Encryption  Decryption  Key Generation | 1*.*654 50  0*.*797 62  527*.*67 | 73*.*916  148*.*79 |

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 genera-tion 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 boot-strapping 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 imple-mented in TFHE-rs. Firstly, it uses the Lagrange half-complex repre-sentation, 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.

4

*L. Brenna et al.*  *Array 13 (2022) 100118*

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 combina-tion step that scales linearly with the number of parties.

After this, the setup phase is complete. We then perform the com-parison circuit equivalent to computing *a* ≤ *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 inde-pendent and are thus evaluable in parallel, although our implementation performs them sequentially.

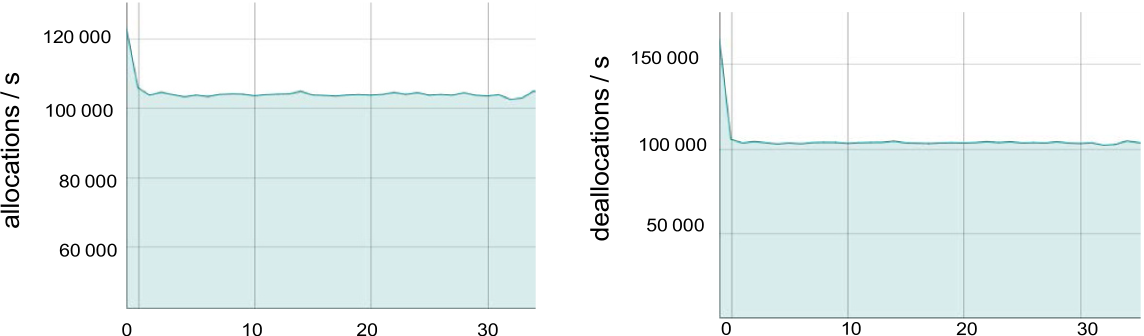
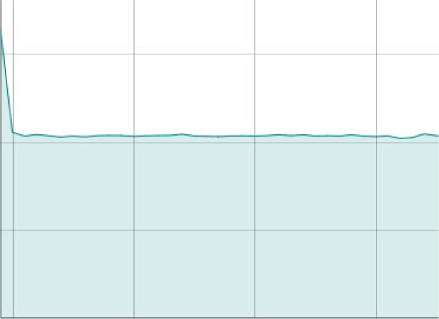
*4.4. TFHE-rs with and without SGX*

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 de-viation 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 be-tween 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

8 [https://github.com/koute/memory-profile](https://github.com/koute/memory-profiler)r.

5



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *L. Brenna et al.* |  |  |  | *Array 13 (2022) 100118* |

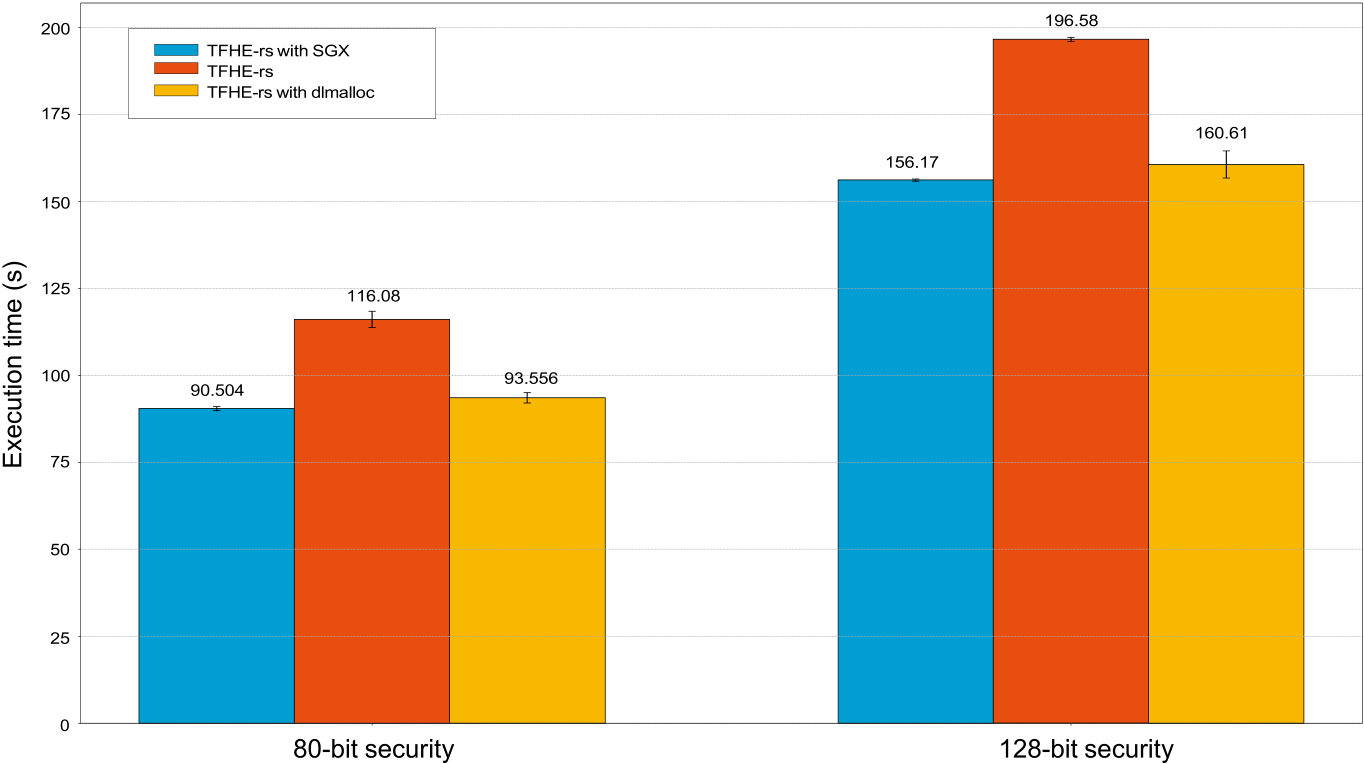
 





**Fig. 3.** Memory usage characteristics for FHE with the default system allocator.



**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 sen-sitive and often comes with strict regulations on how to process and store it. By combining Paillier encryption with SGX they created a sys-computing techniques. tem which achieved a 4*.*8 × speedup compared to existing secure TEEFHE [40] is an example of combining FHE with SGX by per-forming 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

6

*L. Brenna et al.*  *Array 13 (2022) 100118*

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 ~ 9 × to ~ 140 × 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 × .

**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

7

*L. Brenna et al.*  *Array 13 (2022) 100118*

[23] [Genkin D, Shamir A, Tromer E. RSA key extraction via low-bandwidth acoustic](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref23) [cryptanalysis. In: Garay JA, Gennaro R, editors. Advances in cryptology – CRYPTO](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref23) [2014, springer berlin heidelberg. Berlin: Heidel- berg; 2014. p. 444–61.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref23)

[24] [Chillotti I, Gama N, Georgieva M, Izabach](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref23)`[ene M. TFHE: fast fully hom](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref23)omorphic encryption over the Torus. J Cryptol 2019;33. [https://doi.org/10.1007/s00145-](https://doi.org/10.1007/s00145-019-09319-x) [019-09319-x](https://doi.org/10.1007/s00145-019-09319-x).

[25] [Chillotti I, G](https://doi.org/10.1007/s00145-019-09319-x)[ama N, Georgieva M, Izabach`ene M. TFHE: fast fully homomorphic](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref25) [encryption library. Technical Report August 2016 [Https://tfhe.github.io/tfhe/]](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref25).

[26] [Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref26) [approximate numbers. In: Takagi T, Peyrin T, editors. Advances in cryptology](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref26) [–](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref26) [ASIACRYPT 2017, springer international pub- lishing, cham; 2017. p. 409–37.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref26)

[27] [Brakerski Z, Gentry C, Vaikuntanathan V. Fully homomorphic encryp- tion with](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref26)out bootstrapping. Cryptology ePrint Archive, Report 2011/277 2011. [https://eprint.](https://eprint.iacr.org/2011/277) [iacr.org/2011/277](https://eprint.iacr.org/2011/277).

[28] [Lenstra AK. Key le](https://eprint.iacr.org/2011/277)[ngths. Contribution to the handbook of information security.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref28) [2010.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref28)

[29] [Chill](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref28)[otti I, Gama N, Georgieva M, Izabach`ene M. Faster fully homo- morphic](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref29) [encryption: bootstrapping in less than 0.1 seconds. In: Cheon JH, Takagi T, editors.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref29)

[Advances in cryptology – asiacrypt 2016. Berlin, Heidelberg: Springer Berlin](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref29) [Heidelberg; 2016. p. 3–33.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref29)

[30] [Yao AC. Protocols for secu](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref29)re 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/](https://doi.org/10.1109/SFCS.1982.88) [SFCS.1982.88](https://doi.org/10.1109/SFCS.1982.88). [https://doi.org/10.1109/SFCS.1982.88.](https://doi.org/10.1109/SFCS.1982.88)

[31] [Ioannidis I, Gr](https://doi.org/10.1109/SFCS.1982.88)[a](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref31)[ma A. An efficient protocol for Ya](https://doi.org/10.1109/SFCS.1982.88)o[’s mi](https://doi.org/10.1109/SFCS.1982.88)[llionaires’ problem, in: 36th](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref31) [annual Hawaii international conference on system sciences. Proceedings of the](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref31) [2003;2003:6.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref31)

[32] [Lin H-y, Tze](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref31)[ng W-g. An efficient solution to the millionaire](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref32)s[’ problem based on](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref32) [homomorphic encryption. In: In ACNS 2005, vol. 3531; 2005. p. 456–66. of](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref32) [Lecture.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref32)

[33] [Jakobss](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref32)[on M, Yung M. Proving without knowing: on oblivious, agnos- tic and](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref33) [blindfolded provers. In: Koblitz N, editor. Advances in cryptology — crypto ’96.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref33) [Berlin, Heidelberg: Springer Berlin Heidelberg; 1996. p. 186–200.](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref33)

[34] [Chen H, Chillotti I, Song Y. Multi-key homomophic encryption fr](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref33)om TFHE.

Cryptology ePrint Archive, Report 2019/116 2019. [https://eprint.iacr.org](https://eprint.iacr.org/2019/116) [/2019/116](https://eprint.iacr.org/2019/116).

[35] [Drucker N,](https://eprint.iacr.org/2019/116) [Gueron S. Achieving trustworthy homomorphic encryption by](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref35) [combining it with a trusted execution environment. JoWUA 2018;9:86–99](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref35).

[36] [Popa RA, Redfield CMS, Zeldovich N, Balakrishnan H. CryptDB: protectin](http://refhub.elsevier.com/S2590-0056(21)00056-4/sref35)g 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/10.1145/2043[556.2043566.](https://doi.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

8