HasTEE - Confidential Computing on Trusted Execution Environments with Haskell

ABHIROOP SARKAR, Chalmers University, Sweden
ROBERT KROOK, Chalmers University, Sweden
ALEJANDRO RUSSO, Chalmers University, Sweden
KOEN CLAESSEN, Chalmers University, Sweden

Growing concerns about data privacy and confidentiality have spurred the development of Confidential Computing, a security paradigm that facilitates deploying applications in low-trust environments like the cloud and IoT. Confidential Computing allows this low level of trust on various platforms through the support of hardware-enforced memory isolation units known as Trusted Execution Environments (TEEs). Popular TEEs, such as Intel SGX or ARM TrustZone, isolate applications from low-level system software with large codebases, such as operating systems and hypervisors, reducing the Trusted Computing Base (TCB) of applications.

However, the adoption of this technology has been obstructed by the awkward programming model enforced by most TEEs, which requires partitioning an existing application into trusted and untrusted components. The toolchain that supports these components relies entirely on low-level C/C++ libraries that are error-prone, memory-unsafe, and have the potential for accidental information leakage.

We address the above concerns through HasTEE, a Haskell library for programming TEEs. HasTEE presents a type-safe, high-level programming model that enables the execution of Haskell programs on the Intel SGX TEE. The library is capable of automatically partitioning a Haskell application, using the type system, and then running the trusted component of the application on an SGX *enclave* via our modified GHC runtime. Furthermore, the purity and expressive type system of Haskell allows HasTEE to enforce Information Flow Control (IFC) techniques that prevent unintentional leakage of confidential data from the TEE memory.

Our partitioning approach is notably lightweight and does not involve any modification of the Glasgow Haskell Compiler. We demonstrate the practicality of HasTEE across a variety of domains through case studies on (1) privacy-preserving machine learning, (2) an encrypted password wallet, and (3) a data clean room providing differential privacy.

CCS Concepts: • **Security and privacy → Trusted computing**; **Information flow control**; **Security in hardware**; • **Software and its engineering → Functional languages**; **Domain specific languages**.

Additional Key Words and Phrases: Trusted Execution Environment, Intel SGX, Haskell, Enclave

Authors' addresses: Abhiroop Sarkar, Chalmers University, Gothenburg, Sweden, sarkara@chalmers.se; Robert Krook, Chalmers University, Gothenburg, Sweden, krookr@chalmers.se; Alejandro Russo, Chalmers University, Gothenburg, Sweden, russo@chalmers.se; Koen Claessen, Chalmers University, Gothenburg, Sweden, koen@chalmers.se.

1 INTRODUCTION

Confidential Computing is an emerging security paradigm that is progressively transforming the semiconductor industry, with a consequential shift in how software is designed [Mulligan et al. 2021]. At its core, confidential computing aims to secure what is known as *data in use*.

Data in use refers to in-memory data that is distributed across the DRAM, cache lines, page tables and other CPU registers. While encryption has been fairly successful in protecting secrets for *data at rest* (such as databases and file systems) as well as *data in transit* (such as networks using TLS), the need for efficiency and performance has prevented encryption from effectively protecting *data in use*, which has seen an unprecedented rise in vulnerabilities in the last decade [Checkoway and Shacham 2013; Kocher et al. 2018; Lipp et al. 2018].

To protect *data in use*, hardware vendors such as Intel, ARM, and AMD have introduced the concept of a *Trusted Execution Environment (TEE)*, which provides hardware-enforced isolation for in-memory data. A TEE unit, such as the Intel SGX [Intel 2015] or ARM TrustZone [ARM 2004], provides a *disjoint* region of code and data memory that allows for the physical isolation of a program's execution and state from the underlying operating system, hypervisor, I/O peripherals, firmware, and even the physical compromise of a device. As such, the TEE hardware unit has been heralded as a leading contender to enforce a strong notion of *trust* in areas such as cloud computing [Baumann et al. 2015; Zegzhda et al. 2017], IoT [Lesjak et al. 2015] and blockchain [Bao et al. 2020].

However, despite providing such a strong threat model, the adoption of TEEs in modern software development has faced resistance due to their awkward programming model [Decentriq 2022]. At a high level, the programming model requires splitting the state of the program into trusted and untrusted components and dividing the entire logic into two separate software projects. This is often a complex and error-prone process. The issue is further exacerbated by the fact that the API provided by the hardware is entirely based on languages such as C/C++, which are low-level and can open further opportunities to exploit well-known memory-unsafe vulnerabilities such as return-oriented programming (ROP) [Shacham 2007].

Efforts have been made to port high-level managed languages such as GoTEE [Ghosn et al. 2019], a superset of Go, and J_E [Oak et al. 2021], a subset of Java, onto TEEs. Both efforts, however, make large modifications to the respective compiler to parse the source code for effective identification of the trusted and untrusted parts of the code.

Virtualization-based solutions, such as AMD SEV [AMD 2018], on the other hand, typically virtualize an entire application platform, such as the hypervisor. The tradeoff is that the trusted computing base becomes larger and the granularity of secure data becomes coarser.

Our contribution through this paper is the *HasTEE* Haskell library, which offers a high-level programming model for developing TEE applications. In contrast with

GoTEE and J_E, our implementation does not require any modification of the canonical Glasgow Haskell Compiler (GHC) [Jones et al. 1993]. The GHC runtime requires modification to run on the enclave but remains capable of hosting the entire Haskell language (with extensions) supported by GHC 8.8.

The partitioning of a TEE program is accomplished entirely through a 200-line library that uses the Haskell type system to distinguish between the trusted and untrusted components of a program. The granularity of secure data and code in HasTEE remains finer than virtualization-based solutions such as AMD SEV, while keeping a much smaller trusted code base. One benefit of running Haskell on a TEE is the ability to enforce language-based information flow control [Russo et al. 2008] on data that crosses the boundary between the trusted and untrusted parts of the application.

Listing 1 presents a sample password checker application written in HasTEE. HasTEE adopts the term *Enclave* from Intel's nomenclature to refer to the software that runs inside the TEE unit.

```
1  pwdChkr :: Enclave String -> String -> Enclave Bool
2  pwdChkr pwd guess = fmap (== guess) pwd
3
4  passwordChecker :: App Done
5  passwordChecker = do
6    paswd       <- enclaveConstant "secret"
7    enclaveFunc <- secure $ pwdChkr paswd
8    runClient $ do -- the Client monad
9      userInput <- liftIO $ putStrLn "Enter your password" >> liftIO
       getLine
10     res       <- onEnclave (enclaveFunc <.> userInput)
11     liftIO $ putStrLn $ "Your login attempt returned "   <> (show res)
```

Listing 1. A password checker written in HasTEE

HasTEE internally uses conditional compilation provided by GHC to swap in different implementations of the HasTEE API depending on whether it is compiling for the trusted execution environment or the untrusted client. The two compilations use modules that export the same API but differ in the underlying implementations of the functions.

The distinction between the trusted and untrusted parts of the application is done via the type system that encodes the former as the `Enclave` type (line 1) and the latter as the `Client` type (type inferred in line 8). At a high level, the untrusted *client* is in charge of driving the application, while the *enclave* is assigned the role of a computational and/or storage resource that services the client's requests instead of driving the program.

Line 6 holds the secret string that we want to protect. The `enclaveConstant` function accepts the secret string and gets compiled to a no-op term for the client. As a result, the untrusted client can never observe the secret string. The TEE, on the other hand, is the one that holds the string in memory. Line 7 uses the `secure` call to

obtain a reference to the *secure* `pwdChkr` function and then uses a combination of the `onEnclave` and `<.>` combinators (Line 10) to execute the function `pwdChkr` on the enclave and obtain the result on the client side.

HasTEE connects an application (`passwordChecker`) to Haskell's `main` method using the `runApp :: App a -> IO a` function that executes the application. We explain the HasTEE API in detail in Section 4.2 and the semantics in Section 4.3.

Partitioning is done by compiling Listing 1 twice using a conditional flag to assign different semantics to the two programs. Fig 1 shows the partitioning at a high level. This approach is inspired by work on the `Haste.App` library, used to partition a server and client [Ekblad and Claessen 2014] for web programming.

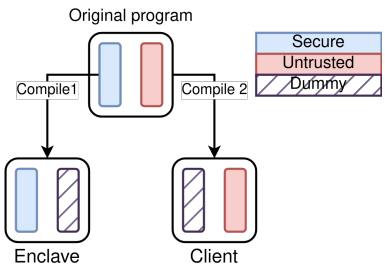

Fig. 1. The HasTEE partitioning scheme

For the information flow control (IFC), the underlying implementation of the `Enclave` type uses a restricted IO monad in the style of `MAC` monads [Russo 2015]. The monad significantly constrains the scope of side-effecting operations to protect the confidentiality of data within the enclave. A detailed account is given in Section 4.5. We summarise our contributions below.

Contributions

- **A type-safe, high-level programming model.** The HasTEE library enables developers to program a TEE environment, such as Intel SGX, using Haskell - a type-safe, memory-managed language whose expressive type system can be leveraged to enforce various security constraints. Additionally, HasTEE allows programming in a familiar client-server style programming model (Section 4.2), an improvement over the low-level Intel SGX APIs.
- **Automatic Partitioning.** A key part of programming TEEs, partitioning the trusted and untrusted part of the program, is done automatically using the type system (details in Section 4.4). Crucially, our approach does not require any modification of the GHC compiler and can be adapted to other programming languages, as long as their runtime can run on the desired TEE infrastructure.
- **Information Flow Control (IFC).** Drawing inspiration from *restricted* IO monad families in Haskell, we designed an `Enclave` monad that prevents

accidental leaks of secret data by TEE programmers (details in Section 4.5). Hence, our `Enclave` monad enables writing applications with a relatively low level of trust placed on the enclave programmer.

- **Practicality.** We illustrate the practicality of the HasTEE library through three case studies across different domains - (1) a Federated Learning example (Section 5.1), (2) an encrypted password wallet (Section 5.2) and (3) a *differentially-private* data clean room (Section 5.3). The examples also demonstrate the simplicity of TEE development enabled by HasTEE.

2 BACKGROUND

This section provides background information on the specific TEE environment that we target with the HasTEE library - Intel SGX.

Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) [Intel 2015] is a set of security-related instructions supported since Intel's sixth-generation Skylake processor, which can enhance the security of applications by providing a *secure enclave* for processing sensitive data. The enclave is a disjoint portion of memory separate from the DRAM, where sensitive data and code reside, beyond the influence of an untrusted operating system and other low-level software. Fig. 2 illustrates the difference in attack surface between an application without enclaves and one that has them.

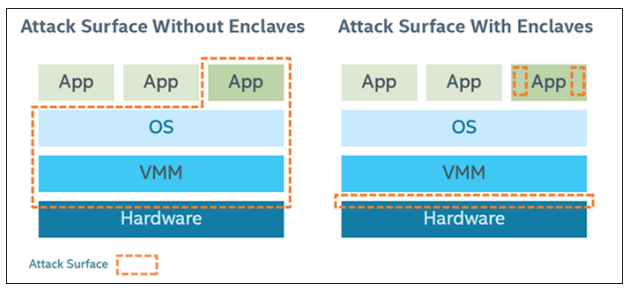

Fig. 2. Intel SGX Attack Surface (Image source [Intel 2018a])

Internally, SGX reserves a 128 MB segment of the physical memory called the *Processor Reserved Memory*. A subset of this memory holds the enclave memory pages in an area called *Enclave Page Cache (EPC)*. The CPU protects the confidentiality of the memory residing in shared cache lines by encrypting the cache when it gets evicted, and decrypting it as it is loaded into the enclave memory while the CPU runs in enclave mode. This incurs a four-fold performance hit [Zhao et al. 2016] but provides stricter hardware-enforced memory isolation.

Intel offers an SGX SDK [Intel 2016] for programming applications with enclaves. The SDK requires partitioning an application into two parts such that the sensitive

data resides in a separate trusted C/C++ project from the untrusted data and code. The SDK provides a specialised set of function calls, referred to as *ecall*, to access the enclave, as well as an *ocall* API for the enclave to communicate with the untrusted client.

The boundary between the client and enclave is defined using a specialised *Enclave Description Language (EDL)*. The SGX SDK parses EDL files using a special tool called *edger8r*, which is part of the SDK. This tool generates two *bridge* files that ensure data transfer between projects is done through *copying*, rather than *sharing* via pointers, to prevent an adversary from compromising the state of the enclave through pointer manipulation. Fig 3 shows the SDK's programming model.

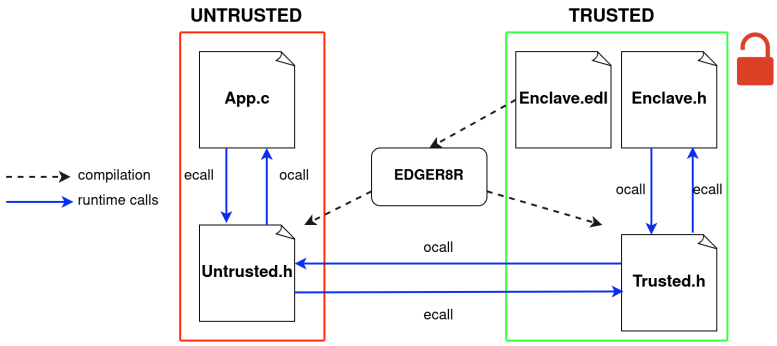

Fig. 3. Intel SGX SDK Programming Model

The primary objective of an application developer when working with enclaves is to minimize the *Trusted Computing Base* (TCB). By placing the operating system, hypervisor, and other system software outside the enclave, the size of the TCB is drastically reduced compared to an application without an enclave. For the very essential system software, such as `libc` (the C standard library), which is required to run any software application, the SGX SDK provides a minimal and lightweight implementation called `tlibc` [Intel 2018b].

Overall, the programming of an SGX enclave involves understanding a fairly complex control flow between the untrusted and the trusted project. The SDK provides an API of 200+ functions to manage the life cycle of an enclave. One of the key difficulties observed in programming enclaves was the awkwardness of enforcing this multi-project programming model on a typical software project. Additionally, the restricted `tlibc` library significantly hinders the ability to run complex applications on the enclaves, beyond those written in vanilla C/C++.

3 KEY IDEA: A TYPED, HIGH-LEVEL PROGRAMMING MODEL FOR ENCLAVES

The key contribution of this paper is the HasTEE library, which aims to simplify the programming of enclaves. The library contributes the following:

(1) HasTEE enables the programming of enclaves in a high-level, memory-managed, statically-typed programming language - Haskell. By running Haskell on an enclave, developers can utilize its expressive type system to type-check the safety of interactions between the client and the enclave.
(2) HasTEE allows expressing enclave-based programs in a familiar single-program, client-server-based programming model rather than the low-level, multi-project approach provided by the SGX SDK.
(3) HasTEE, being embedded in Haskell, enables the expression of language-based security constraints such as Information Flow Control (IFC) on sensitive data that moves in and out of the enclave memory.

One of the key challenges in accomplishing point (1) is running the GHC Haskell Runtime [Marlow et al. 2009] on an enclave. A Haskell program relies on the runtime for essential tasks such as memory allocation, concurrency, I/O management, etc. The GHC runtime heavily depends on well-known C standard libraries, such as `glibc` on Linux [GNUDevs 1991] and `msvcrt` on Windows [Microsoft 1994]. In contrast, the Intel SGX SDK, as discussed in the previous section, provides a much more restricted `libc` known as `tlibc`.

This results in the fact that several `libc` calls used by the GHC runtime such as `mmap`, `madvise`, `epoll`, `select` and 100+ other functions become unavailable. Even the core threading library used by the GHC runtime, `pthread`, has a much more restricted API on the SGX SDK. To solve this conundrum, we have patched portions of the GHC runtime and used functionalities from a library OS, Gramine [C. Tsai, Porter, et al. 2017], to enable the execution of GHC-compiled programs on the enclave.

The Programming Model and Partitioning

Once Haskell programs are capable of running on an enclave, there is the requirement of partitioning the original source program. As mentioned in point (2) above, HasTEE doesn't require the programmer to manually partition the program but uses a conditional compilation trick to partition the same compilation unit into two parts.

The conditional compilation tactic was first presented in Haste.App [Ekblad and Claessen 2014], which is a Haskell library for web programming. The library automatically partitions the application into a `Client` and `Server` type using a conditional compilation trick that swaps in different implementations of the terms corresponding to the above types. The function calls between the `Client` and `Server` are checked for type safety at compile time, where they are replaced with corresponding remote procedure calls (RPC) at runtime.

We adopt the same partitioning tactic owing to its simplicity, as it does not require any compiler extensions or elaborate dependency analysis passes to distinguish between the underlying types. The codebase involved in other complex partitioning approaches [Ghosn et al. 2019; Oak et al. 2021] becomes part of the Trusted Computing Base (TCB), creating a larger TCB. In contrast, the conditional compilation approach

does not add any code to the TCB. Fig 4 shows the partitioned software stack in the HasTEE approach.

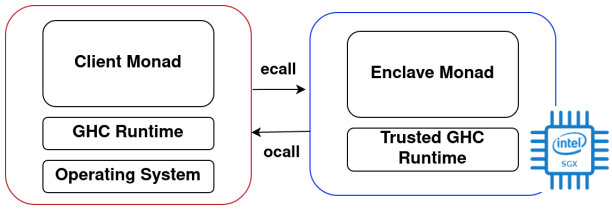

Fig. 4. The untrusted software (left) and the trusted software stack (right)

Post-partitioning, it is natural to use the client-server-style programming model for programming the enclave. In this model (example in Listing 1), the client remains the primary driver of the program while utilizing the enclave as a computational and/or storage resource, rather than making it the main driver of the program. At compile time, the source program gets type safety because of being written in Haskell, while at runtime, the HasTEE library handles the copying of the actual messages between the client and the enclave memory The detailed API is presented in Section 4.2.

Information Flow Control on Enclaves

As stated in point (3) above, being a Haskell library enables HasTEE to tap into the library-based Information Flow Control techniques in Haskell [Buiras et al. 2015; Russo 2015; Russo et al. 2008]. The IFC literature distinguishes between security-critical and security-relaxed monad families using a `Sec H` and a `Sec L` monad. We have a similar security-level hierarchy between the `Enclave` and `Client` monads, respectively. Accordingly, we design the `Enclave` monad such that it restricts the possible variants of I/O operations. Due to the security-critical nature of the Enclave monad, we include a `trust` operator, which is similar to the `endorse` function found in IFC literature.

TEE Independence

Another benefit of being a high-level programming model is that it provides an abstraction over low-level system APIs offered by the SGX SDK. As a result, the principles applied in programming Intel SGX, should translate to the programming of other popular TEEs, such as the ARM TrustZone. TrustZone employs a similar two-project approach as Intel SGX, along with a gateway description header file similar to the `.edl` file used in the SGX SDK. An important point to note here is that while the high-level API and programming model can be easily translated, considerable effort is required to port the GHC runtime to the TrustZone infrastructure.

4 DESIGN AND IMPLEMENTATION OF HASTEE

This section first outlines the high-level design and semantics of the HasTEE library. Later in this section, we will describe the internal implementation details of the library.

4.1 Threat Model

We begin by discussing the threat model of the HasTEE library. HasTEE has the very same threat model as that of Intel SGX. In this model, only the software running inside the enclave memory is trusted. All other application and system software, such as the operating system, hypervisors, driver firmware, etc., are considered compromised by an attacker. A very similar threat model is shared by a number of other work based on Intel SGX [Arnautov et al. 2016; Baumann et al. 2015; Ghosn et al. 2019; Lind et al. 2017].

In this work, we enhance the application-level security firstly by using a memory-safe language, Haskell, and secondly by introducing information flow control via the `Enclave` monad. Our implementation strategy of loading the GHC runtime on the enclave allows us to handle Iago attacks [Checkoway and Shacham 2013] (see Section 4.4.1). We trust the underlying implementation of the SGX hardware and software stack (such as `tlibc`) as provided by Intel. Known limitations of Intel SGX such as denial-of-service attacks and side-channel attacks [Schaik et al. 2022] are beyond the scope of this paper.

An ideally secure development process should include auditing the code running on the enclave either through static analyses or manual code reviews or both. The conciseness of Haskell codebases should generally facilitate the auditing process. However, the mechanisms for fail-proof audits are beyond the scope of this paper as well.

4.2 HasTEE API

We show the core API of the HasTEE library in Fig 5. The functions presented operate over three principal Haskell data types: (1) `Enclave`, (2) `Client`, and (3) `App`. All three types are instances of the `Monad` typeclass, which allows for the use of `do` notation when programming with them. One of the key differences in functionality provided by the `Client` and `Enclave` monads is the restriction on arbitrary IO actions enforced by the `Enclave` monad. The `App` monad sets up the infrastructure for communication between the `Client` and `Enclave` monad. We show a simple secure counter example written using most of the API in Listing 2.

Listing 2 internally gets partitioned into the trusted and untrusted components via conditional compilation. In line 3, `liftNewRef` is used to create a secure reference initialised to the value 0. Followed by that, the computation to increment this value inside the enclave is given in lines 4 - 7. Applying `secure` on the enclave computation (line 4) yields the type `App (Secure (Enclave Int))`. The `Secure` type is an internal representation used by HasTEE to represent a closure that is present in the enclave

```
-- mutable references
liftNewRef :: a → App (Enclave (Ref a))
readRef    :: Ref a → Enclave a
writeRef   :: Ref a → a → Enclave ()

-- get a reference to call a function inside the enclave
secure :: Securable a ⇒ a → App (Secure a)

-- runs the Client monad
runClient :: Client () → App Done

-- used for function application on the enclave
onEnclave :: Binary a ⇒ Secure (Enclave a) → Client a
(<.>)     :: Binary a ⇒ Secure (a → b) → a → Secure b

-- call this from `main` to run the App monad
runApp    :: App a → IO a
```

Fig. 5. The core HasTEE API

```
1  app :: App Done
2  app = do
3    enclaveRef <- liftNewRef 0 :: App (Enclave (Ref Int))
4    count <- secure $ do
5      r <- enclaveRef
6      v <- readRef r
7      writeRef r (v + 1) >> return v :: Enclave Int
8    runClient $ onEnclave count >>= (\v ->
9        liftIO $ print $ "Counter's #" ++ show v)
10
11 main = runApp app
```

Listing 2. A secure counter written in HasTEE (types annotated for clarity)

memory. Line 8 uses the critical `onEnclave` function to actually execute the enclave computation within the enclave memory and get the result back in the client memory. This resulting value, `v`, is displayed to the user.

The only function from Fig. 5 not used in Listing 2 is the `<.>` operator, used to collect arguments that are sent to the enclave. For example, an enclave function, `f`, that accepts two arguments, `arg1` and `arg2`, would be executed as `onEnclave (f <.> arg1 <.> arg2)`. Listing 1 in Section 1 shows a concrete usage of the operator. We have larger case studies in Section 5.

4.3 Operational Semantics of HasTEE

In this section, we provide a big-step operational semantics of the core functionalities within the HasTEE library. We present the operational semantics through the use of an evaluator (interpreter) for the core operators of HasTEE, which intends to show how the client and enclave memory evolve as a program executes. We show our *expression language* and the abstract machine values to which we evaluate below:

```haskell
type Name = String

data Exp = Lit Int | Var Name | Fun [Name] Exp | App Exp [Exp]
         | Let Name Exp Exp   | Plus Exp Exp
         -- HasTEE operators
         | Secure Exp  | OnEnclave  Exp | EnclaveApp Exp Exp -- (<.>)

data Value = IntVal Int | Closure [Name] Exp Env
           -- HasTEE values
           | SecureClosure Name [Value] | ArgList [Value] | Dummy
           -- Error values
           | Err ErrState
```

The `Exp` language above is a slightly modified version of lambda calculus with the restriction of allowing only fully applied function application. This restriction is done to reflect the nature of the HasTEE API, which through the type system, only permits fully saturated function application for functions residing in the enclave. The lambda calculus language is then extended with the core HasTEE operators.

In the `Value` type, the `Closure` constructor represents a very standard approach to capture closures. Owing to saturated function application, it captures a list of variable names rather than a single name. Notable in the `Value` type is the `SecureClosure` constructor that represents a closure residing in the enclave memory. This constructor does not capture the body of the closure as the body could hold any hidden state that lies protected within the enclave memory. The `SecureClosure` value is used by the `onEnclave` function to invoke functions residing in the enclave.

The `ArgList` constructor, as the name suggests, collects the arguments that are to be sent over to the enclave. The `<.>` operator, when applied to various arguments, builds up this list. Lastly, the `Dummy` value is used as a placeholder for operators that do not have any semantics depending on the client or the enclave memory. For instance, the `onEnclave` function has no meaning inside the `Enclave` monad, it is only usable from the `Client` monad. The `Dummy` value is an important value type that enables the conditional compilation trick in HasTEE by acting as a placeholder for meaningless functions in the respective client and enclave memory.

The evaluator operates on an environment that maps variable names to values. As there are two distinct memories here - the enclave memory and the client memory, we have two environments.

```haskell
type ClientEnv  = [(Name, Value)]
type EnclaveEnv = [(Name, Value)]
```

```
testProgram = let m = 3 in
              let f = λ x -> x + m in
              let y = secure f in
              onEnclave (y <.> 2)
```

Listing 3. A simple program for illustrating the operational semantics of HasTEE

Interestingly, the conditional compilation of HasTEE requires us to define an evaluator that operates in two passes. In the first pass, it runs a program and loads up the necessary elements in the enclave memory and then in the second pass, the loaded enclave memory is additionally passed to the client's evaluator. The following demonstrates this:

```
eval :: Exp -> Value
eval e =
  let newEnclaveEnv = snd $ evalState (evalEnclave e initEnclaveEnv)
                              (initStateVar initEnclaveEnv)
  in fst $ evalState (evalClient e initClientEnv) (initStateVar newEnclaveEnv)
  where
    initEnclaveEnv = []
    initClientEnv  = []

data StateVar = StateVar { varName  :: Int, encState :: EnclaveEnv }

initStateVar :: EnclaveEnv -> StateVar
initStateVar = StateVar 0
```

The `StateVar` type exists to generate variable names and to hold the enclave environment when the evaluator for the client is run. The `evalEnclave` and `evalClient` functions are presented in Fig 6 and Fig. 7, respectively. The type signature of both the functions are similar and they represent the change in state of the respective environments as a term of the `Exp` language is evaluated.

Two helper functions, `genEncVar` and `evalList` are not shown for concision. The first one is a standard function for generating unique variable names using the `Int` held by the `StateVar` type. The `evalList` function is effectively a fold function over a list of expressions using the respective evaluators. Appendix A contains the complete, executable semantics written in Haskell.

We demonstrate the semantics by examining a simple example program (Listing 3) written in the above expression language and discuss the differences between the client and enclave memories for this program.

Our semantic evaluator operates in two passes. In the first pass, the `evalEnclave` evaluator from Fig. 6 is run. Fig. 8a shows the state of the enclave environment after the evaluator has completed evaluating Listing 3. Notably, the variable `y` maps to a value with no semantic meaning, as the evaluator is already running in the secure memory.

```haskell
evalEnclave :: (MonadState StateVar m)
            ⇒ Exp → EnclaveEnv → m (Value, EnclaveEnv)
evalEnclave (Lit n) env    = pure (IntVal n, env)
evalEnclave (Var x) env    = pure (lookupVar x env, env)
evalEnclave (Fun xs e) env =
  pure (Closure xs e env, env)
evalEnclave (Let name e1 e2) env = do
  (e1', env') ← evalEnclave e1 env
  evalEnclave e2 ((name,e1'):env')
evalEnclave (App f args) env     = do
  (v1, env1)   ← evalEnclave f env
  (vals, env2) ← evalList args env1 []
  case v1 of
    Closure xs body ev →
      evalEnclave body ((zip xs vals) ++ ev)
    _ → pure (Err ENotClosure, env2)
evalEnclave (Plus e1 e2) env = do
  (v1, env1) ← evalEnclave e1 env
  (v2, env2) ← evalEnclave e2 env1
  case (v1, v2) of
    (IntVal a1, IntVal a2) → pure (IntVal (a1 + a2), env2)
    _                      → pure (Err ENotIntLit, env2)
evalEnclave (Secure e) env = do
  (val, env') ← evalEnclave e env
  varname     ← genEncVar
  let env''   = (varname, val):env'
  pure (Dummy, env'')
-- the following two are essentially no-ops
evalEnclave (OnEnclave e) env = evalEnclave e env
evalEnclave (EnclaveApp e1 e2) env = do
  (_, env1) ← evalEnclave e1 env
  (_, env2) ← evalEnclave e2 env1
  pure (Dummy, env2)
```

Fig. 6. Operational Semantics of the Enclave

In the second pass, the environment from Fig. 8a is additionally passed as a state variable to the evaluator `evalClient` from Fig. 7. Line 3 creates a different mapping from the enclave evaluator when evaluating the client, as shown in Fig 8b. The `<.>` operator is evaluated as the `EnclaveApp` constructor on lines 25-33 in Fig 7. It collects the arguments to generate `SecureClosure "`$EncVar_0$`" [Lit 2]`.

The most significant part of Listing 3 occurs on line 4 when the client evaluates the `onEnclave` call. The semantics for this evaluation can be found in lines 12-24 of Fig 7. The evaluator finds a reference $EncVar_0$ with no semantics in the client memory (as shown in Fig 8b). The evaluator then looks up the variable in the enclave environment (Fig 8a) and finds a `Closure` with a body. Crucially, **it evaluates the**

```
1  evalClient :: (MonadState StateVar m)
2             ⇒ Exp → ClientEnv → m (Value, ClientEnv)
3
4  {- evalClient for Lit, Var, Fun, Let, App, Plus not shown as
5     they have the identical semantics as evalEnclave above -}
6
7  evalClient (Secure e) env = do
8    (_, env') ← evalClient e env
9    varname   ← genEncVar
10   let env'' = (varname, Dummy):env'
11   pure (SecureClosure varname [], env'')
12 evalClient (OnEnclave e) env = do
13   (e', env1) ← evalClient e env
14   case e' of
15     SecureClosure varname vals → do
16       enclaveEnv ← gets encState
17       let func = lookupVar varname enclaveEnv
18       case func of
19         Closure vars body encEnv → do
20           (res,enclaveEnv') ←
21                 evalEnclave body ((zip vars vals) ++ encEnv)
22           pure (res, env1)
23         _ → pure (Err ENotClosure, env1)
24     _ → pure (Err ENotSecClos, env1)
25 evalClient (EnclaveApp e1 e2) env = do
26   (v1, env1) ← evalClient e1 env
27   (v2, env2) ← evalClient e2 env1
28   case v1 of
29     SecureClosure f args →
30       case v2 of
31         ArgList vals → pure (SecureClosure f (args ++ vals), env2)
32         v            → pure (SecureClosure f (args ++ [v]), env2)
33     v → pure (ArgList [v,v2], env2)
```

Fig. 7. Operational Semantics of the Client

`Closure` by invoking the `evalEnclave` function on line 21 of Fig. 7 using the enclave environment. This part models how the SGX hardware switches to the enclave memory when executing the secure function `f` rather than the client memory. An important point is generating an identical fresh variable name, $EncVar_0$, that the client uses to identify and call the functions in the enclave memory.

4.4 HasTEE implementation

We now discuss the implementation details required to support HasTEE.

$$
\begin{array}{c}
m \longmapsto 3 \\
f \longmapsto Closure\ [\text{"}x\text{"}]\ (x+m)\ [m \mapsto 3] \\
EncVar_0 \longmapsto Closure\ [\text{"}x\text{"}]\ (x+m)\ [m \mapsto 3] \\
y \longmapsto Dummy
\end{array}
$$

(a) Enclave Environment

$$
\begin{array}{c}
m \longmapsto 3 \\
f \longmapsto Closure\ [\text{"}x\text{"}]\ (x+m)\ [m \mapsto 3] \\
EncVar_0 \longmapsto Dummy \\
y \longmapsto SecureClosure\ \text{"}EncVar_0\text{"}\ []
\end{array}
$$

(b) Client Environment

Fig. 8. The Enclave Environment in (a) is loaded during the first pass of the evaluator, and the Client Environment is empty during this pass. In the second pass, the Client Environment (b) gets loaded while having access to the memory (a), as can be seen in Fig 7

4.4.1 Trusted GHC Runtime. One of the crucial challenges in implementing the HasTEE library is enabling Haskell programs to run within an Intel SGX enclave. As already discussed in Section 3, all Haskell programs compiled via the Glasgow Haskell Compiler (GHC), rely on the GHC runtime [Marlow et al. 2009] for crucial operations such as memory allocation and management, concurrency, I/O management, etc. As such, it is essential to port the GHC runtime in order to run Haskell programs on the enclave.

The GHC runtime is a complex software that is heavily optimized for specific platforms, such as Linux and Windows, to maximize its performance. For instance, on Linux, the runtime relies on a wide variety of specialised low-level routines from a C standard library, such as `glibc` [GNUDevs 1991] or `musl` [Felker 2005], to provide essential facilities like memory allocation, concurrency, and more. The challenge lies in porting the runtime due to the limited and constrained implementation of the C standard library in the SGX SDK, called `tlibc` [Intel 2018b]. Specifically, `tlibc` does not support some of the essential APIs required by the GHC runtime, including `mmap`, `madvise`, `munmap`, `select`, `poll`, a number of `pthread` APIs, operations related to timers, file reading, writing, and access control, and other functionalities that add up to 100+ functions.

Given the magnitude of engineering effort required to port the GHC runtime, we fall back on a library OS called Gramine [C. Tsai, Porter, et al. 2017]. Gramine internally intercepts all `libc` system calls within an application binary and maps them to a Platform Abstraction Layer (PAL) that utilizes a smaller ABI. In Gramine's case, this amounts to only 40 system calls that are executed through dynamic loading and runtime linking of a larger `libc` library, such as `glibc` or `musl`. Importantly, to protect the confidentiality and integrity of the enclave environment, Gramine uses a concept known as *shielded execution*, pioneered by the Haven system [Baumann et al. 2015], where a library is only loaded if its hash values are checked against a measurement taken at the time of initialisation. Shielded execution further protects applications against *Iago attacks* [Checkoway and Shacham 2013] in Gramine.

However, there are additional difficulties in loading the GHC runtime on the SGX enclave via Gramine. Owing to Gramine's diminished system ABI, it has a dummy

or incomplete implementation for several important system calls that the runtime requires. For instance, the absence of the `select`, `pselect`, and `poll` functions, which are used in the GHC IO manager, required us to modify the GHC I/O manager to manually manage the polling behavior through experimental heuristics. Similarly, the critical `mmap` operation in GHC uses specific flags (`MAP_ANONYMOUS`) that require modification. In addition, other calls, such as `madvise`, `getrusage`, and timer-based system calls, also require patching. We hope to quantify these modifications' performance in the future.

After the GHC runtime is loaded onto an enclave, communication between the untrusted and trusted parts of the application effectively occurs between two disjoint address spaces. Communication between them can happen over any binary interface, emulating a remote procedure call. As our implementation is in its early prototype stage, we transmit serialised data as TCP packets (Fig 9). A production implementation should communicate via the C ABI using the Foreign Function Interface (FFI) supported by Haskell.

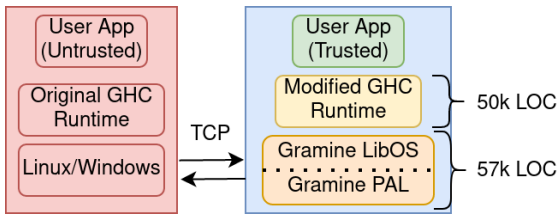

Fig. 9. The high-level overview of communication between the untrusted and trusted parts of the app

The Gramine approach results in an increase of 57,000 lines of code in the Trusted Computing Base (TCB) [C. Tsai, Porter, et al. 2017]. However, this is still an improvement over traditional operating systems, like Linux, with a TCB size of 27.8 million lines of code [Larabel 2020].

4.4.2 HasTEE Library. The API of the HasTEE library was already shown (Figure 5) and discussed in Section 4.2. The principal data types, `Enclave` and `Client`, have been implemented as wrappers around the IO monad, as shown below:

```
1 newtype Enclave a = Enclave (IO a) -- data constructor not exported
2 type Client = IO
```

A key distinction is that the `Enclave` data type does not instantiate the `MonadIO` typeclass, as a result of which arbitrary IO actions cannot be lifted inside the `Enclave` monad.However, it does instantiate a `RestrictedIO` typeclass that will be discussed in the following section. The conditional-compilation-based partitioning technique is achieved by having dummy implementations of certain data types in one of the modules, while the concrete implementation of those types is defined in the second module. We give an example of this using two different data types from the API.

```haskell
-- Enclave.hs
data Secure a = SecureDummy

type Ref a = IORef a
```

```haskell
-- Client.hs
data Secure a =
  Secure CallID [ByteString]
type Ref a = RefDummy
```

A notable aspect of the API is the `Securable` typeclass, which constrains the `secure` function and enables it to label functions with any number of arguments as residents of the enclave memory. The `Securable` typeclass accomplishes this using a well-known typeclass trick in Haskell, used to represent statically-typed variadic functions such as `printf` [Augustsson and Massey 2013].

The operational semantics presented in Section 4.3 should provide an intuition for the core implementation techniques used in the library. The actual Haskell code of the HasTEE library has been open-sourced[1]. Additionally, the Haste.App paper [Ekblad and Claessen 2014] provides more details on the techniques used.

4.5 Information Flow Control for Enclaves

The HasTEE library, being written in Haskell, allows using language-based Information Flow Control (IFC) techniques available in Haskell [Russo et al. 2008]. IFC approaches in Haskell aim to protect the confidentiality of data by encapsulating computations within a `Sec` monad. Typically, the monad employs a lattice of *labels* [Denning 1976] to model various security levels and then enforces policies on how data can flow between the levels. For a two-label lattice, where confidential data is marked with `H` and public data with `L`, a security policy known as *non-interference* is to prevent information flow from the secret to public channels [Goguen and Meseguer 1982]. In other words, $L \sqsubseteq L$, $H \sqsubseteq H$, $L \sqsubseteq H$, but $H \not\sqsubseteq L$, where $\sqsubseteq$ indicates the *flows to* relation.

A similar scenario arises in HasTEE, where the `Enclave` monad can be compared to a security-critical `Sec H` monad that attempts to prevent information leakage to a public `Sec L` channel represented by the `Client` monad. Enforcing the non-interference policy in this scenario would imply that no data can flow out of the `Enclave` monad to the `Client`, which would make the enclave very restrictive for any real-world use cases. As such, the IFC literature relaxes the non-interference policy by the means of *declassification* [Sabelfeld and Sands 2005], which allows controlled leak of data from `H` to `L`.

In the HasTEE API, the `onEnclave :: (Binary a) => Secure (Enclave a) -> (Client a)` function is an *escape hatch* [Hedin and Sabelfeld 2012] that allows the enclave to leak *any* data to the client. We prioritise the usability of the API and trust that the enclave programmer will make the `onEnclave` call when they are certain they want to intentionally leak information to a public channel. However, there is a hidden line of defence in the `onEnclave` function. If the programmer intends to transmit any user-defined data type to the untrusted client, they are required to provide an instance of the `Binary` typeclass. Writing this typeclass instance for some

[1] github link hidden for the review process

confidential data type, such as a private key, equips the confidential data with the capacity to leave the enclave boundary, which should be done in a highly controlled manner.

Besides the `onEnclave` function, the `Enclave` monad has occasional requirements to interact with general I/O facilities like file reading/writing or random number generation. For such operations, the `Enclave` monad would need a `MonadIO` instance in Haskell to perform any I/O operations. However, as discussed in the previous section, we do not provide the lenient `MonadIO` instance to the Enclave monad but instead, use a `RestrictedIO` typeclass to limit the types of I/O operations that an `Enclave` monad can do.

4.5.1 Restricted IO. `RestrictedIO`, shown in Listing 4, is a collection of typeclasses that constrains the variants of I/O operations possible inside an `Enclave` monad. For instance, if a programmer, through the usage of a malicious library, mistakenly attempts to leak confidential data through a network call, the typeclass would not allow this.

```
1 type RestrictedIO m = (EntropyIO m, UnsafeFileIO m) -- other
      typeclasses not shown
2
3 class EntropyIO (m :: Type -> Type) where
4     type Entropy m:: Type
5     genEntropyPool :: m (Entropy m)
6
7 class UnsafeFileIO (m :: Type -> Type) where
8     untrustedReadFile  :: FilePath -> m (Untrusted String)
```

Listing 4. The Restricted IO typeclass

This approach is invasive in that it restricts how a library (malicious or otherwise) that interacts with a HasTEE program conducts I/O operations. For instance, we had to modify the HsPaillier library [L.-T. Tsai and Sarkar 2016] that used the `genEntropy` function for random number generation. Initially, the library could use the Haskell IO monad freely, but to interact with a package written in HasTEE, it had to be modified to use the more restricted type class constraint (`EntropyIO`) for its *effectful* operations. This limits potential malicious behaviour within the library. Notably, our changes involve only five lines of code that instantiate the type class and generalize the type signature of effectful operations.

Another aspect of IFC captured in our system is the notion of *endorsement* [Hedin and Sabelfeld 2012], which is essentially the dual of declassification. Endorsement is concerned with the integrity, i.e., trustworthiness, of information. In our scenario, we utilize endorsement to ensure that the integrity of secrets is not compromised by data being introduced into the enclave.

HasTEE allows file reading operations inside the `Enclave` monad, which can potentially corrupt the enclave's data integrity. To control this, HasTEE provides two forms of file reading operation - (1) untrusted file read and (2) trusted encrypted file

reads. For (1), data can be read from untrusted files but the enclave programmer is required to manually endorse the untrusted data read using the `trust` operator. This provides an additional check before untrusted data interacts with the trusted domain. The types involved are shown below:

```
data Untrusted a -- | A wrapper over untrusted data

-- | Indicate trust in a potentially untrusted value
trust :: Untrusted a -> a
```

For point (2), HasTEE relies on an Intel SGX feature known as *sealing*. Every Intel SGX chip is embedded with a unique 128 bit key known as the Root Seal Key (RSK). The SGX enclave can use this RSK to encrypt trusted data that it wishes to persist on untrusted media. This process is known as sealing, and HasTEE provides a simple interface to seal as well as unseal the trusted data being persisted. The API is shown below:

```
data SecurePath = SecurePath String

securefile :: FilePath -> SecurePath
securefile fp = "/secure_location/" <> fp -- path hidden from the user

readSecure  :: SecurePath -> Enclave String
writeSecure :: SecurePath -> String -> Enclave ()
```

In the above, the `writeSecure` operation corresponds to *ciphertext declassification* [Askarov et al. 2008], while `readSecure` to an operation that applies automatic endorsement if the file can be decrypted successfully by the enclave RSK. If an attacker were to locate the secure location, the worst possible outcome would be the deletion of the file. However, the contents of the file cannot be read or modified outside the enclave, so the attacker would not be able to access the sensitive information stored within.

5 CASE STUDIES

In this section, we will demonstrate how HasTEE can be used to enforce data confidentiality using three case studies spread across a variety of domains.

5.1 Federated Learning

Federated Learning is an emerging *privacy-preserving* machine learning [Al-Rubaie and Chang 2019] approach that allows multiple parties to train a model without sharing the raw training data. A typical federated learning setup involves multiple decentralized edge devices holding local datasets, training a model locally and then aggregating the trained model on a cloud server. Fig. 10 shows the desired setup.

The setup in Fig. 10 above is facilitated by a combination of TEEs and homomorphic encryption. Homomorphic Encryption (HE) [Gentry 2009] is a form of encryption that enables direct computation on encrypted data, revealing the computation result

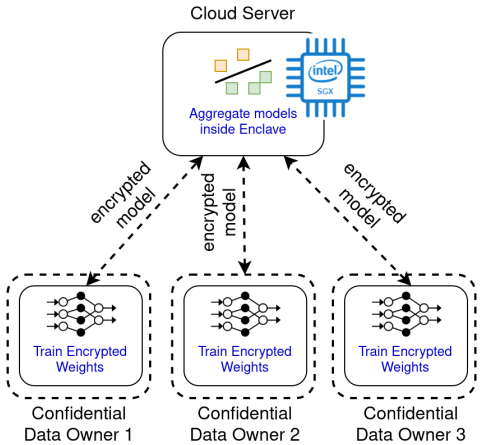

Fig. 10. A Federated Learning setup where the data owners are protecting their data and the ML model owner is protecting their model. The training with encrypted weights can be done using homomorphic encryption.

only to the decryption key owner. We emulate the very same setup for our case study where we have two mutually distrusting parties -

(1) Confidential data owner: The data owner could, for instance, be a hospital that does not want to send confidential patient data to an untrusted cloud server. They additionally want to protect their private data from other participants. The training process is conducted on a data owner's machine locally and only the encrypted weights are transferred between the cloud server and data owner so that the data privacy of the model is protected.
(2) ML model owner: The ML model owner wants to protect their model as it could be considered their intellectual property. They want the model to remain confidential in both the client machines where the training occurs, as well as from the cloud server, which may be an untrusted multi-tenant virtual machine. Weights from each data owner are aggregated and updated in an enclave of the cloud server before being sent back to the data owner. Although the model owner has the private key of HE, it has no chance to speculate on the data owners' private data from the weights.

The above setup only requires the cloud server supporting Intel SGX technology so that even mobile devices can participate in training as a worker role. We can very conveniently model this entire setup as three clients and a server with an enclave in HasTEE. For illustration purposes, we will use GHC's threads to represent the three clients instead of three separate data owner machines.

The server's state is modelled in Listing 5. Note that the state keeps the weights in plaintext form. The enclave state holds both its public and private keys. However,

```
data SrvSt = SrvSt { publicKey  :: PubKey, privateKey :: PrvKey
                   , updWts     :: Vector Double, numClients :: Int
                   , wtsDict    :: Map -- (key=Epoch,
                                       --  value=[Vector CipherText]
                                       -- )
                   }
```

Listing 5. The Federated Learning server state

only the public key should be allowed to move to the client. We enforce this by not providing an instance of the `Binary` typeclass for the private key. As discussed in Section 4.5, the lack of a `Binary` instance for the `privateKey` will prevent the enclave programmer from accidentally leaking the security-critical private key.

Listing 6 shows the API exposed to the client machine. Instead of the complex `SGX_ECALL` machinery, our API is expressed in idiomatic Haskell. Calling any function `f` from the record `api` with an argument `arg` in this API is expressed simply as `onEnclave ((f api) <.> arg)`.

```
type Accuracy = Double
type Loss     = Double
data API = API
  { aggregateModel :: Secure (Epoch -> Vector CipherText
                     -> Enclave (Maybe (Vector CipherText)))
  , validateModel  :: Secure (Enclave (Accuracy, Loss))
  , getPublicKey   :: Secure (Enclave PubKey)
  , reEncrypt      :: Secure (CipherText -> Enclave CipherText)
  }
```

Listing 6. The Federated Learning client API

We also show the main machine learning loop running on the data owner's machine in Listing 7. A number of important functions have been elided for brevity, but the key portions of the client-server interaction in HasTEE should be visible. The `Config` type holds the encrypted weights sent from the cloud server and after each epoch updates to the new aggregated value (Line 13). The value `x'` is the data set that the data owners are protecting and `y` is the result of the learning algorithm. The `adjustModelWithLearningRate` function (body elided, line 8) takes the computed gradient (line 7) and tries to converge on the desired result.

The client-server communication happens in both lines 9 and 10. On line 9 the server is communicated to aggregate models spread across different clients, with the server returning the encrypted updated weights `wt'`. We use a wrapper over `onEnclave`, called `retryOnEnclave` (body elided), that functionally allows the server to move in lock step with all the clients. Then in line 10, the server is communicated again to collect the accuracy and loss in the ongoing epoch number, which gets displayed in line 11. Finally, the loop continues in line 13.

```
{- The Config type is a record that stores configuration info such as
   the learning rate, the current epoch number, encrypted weights and
   the public key -}
handleSingleEpoch :: API
                  -> CurrentEpochNum
                  -> MaxEpochNum
                  -> Matrix Double
                  -> Vector Int
                  -> Config
                  -> Client Config
handleSingleEpoch api n m x' y cfg'
  | n == m    = return cfg'
  | otherwise = do
    grad   <- computeGradient api cfg' x' y
    cfgNew <- adjustModelWithLearningRate api (cfg' { iterN = n })
    grad
    wt'    <- retryOnEnclave $   (aggregateModel api)
                             <.> n
                             <.> (weights cfgNew)
    (acc, loss) <- onEnclave (validateModel api)
    printClient $  " Iteration no: " <> show n
                <> " Accuracy: "     <> show acc
                <> " Loss : "        <> show loss
    handleSingleEpoch api (n+1) m x' y (cfgNew { weights = wt' })
```

Listing 7. The key machine learning loop; a number of functions have been elided for brevity

Listing 7 above features a complex control flow with at least two interactions visible in the loop itself. Internally, both the `computeGradient` as well as `adjustModelWithLearning` functions also talk to the enclave, calling the `reEncrypt` function to remove noise from the homomorphic encryption operation. HasTEE can represent a fairly complex, asynchronous control flow as simple Haskell function calls.

In terms of Information Flow Control, there are two important aspects in this case study. Firstly, the `RestrictedIO` typeclass constrains potentially malicious libraries from misbehaving. For example, consider the library HsPaillier [L.-T. Tsai and Sarkar 2016], which implements the Paillier Cryptosystem [Paillier 1999] for partial homomorphic encryption. All effectful operations from this library, such as `genKey :: Int -> IO (PubKey, PrvKey)`, need to be rewritten for them to be usable within the `Enclave` monad. The following snippet shows our typeclass instantiation and a sample type signature change needed inside the library.

```
instance (IO ~ m) => EntropyIO m where
    type Entropy m = EntropyPool
    genEntropyPool = createEntropyPool

-- genKey :: Int -> IO (PubKey, PrvKey) -- original type
```

```haskell
-- | A single entry of authentication tokens
data Item = Item { title :: String
                 , username :: String
                 , password :: Password
                 }
  deriving (Show, Read)

-- | The secure wallet
data Wallet = Wallet { items :: [Item]
                     , size :: Int
                     , masterPassword :: Password
                     }
  deriving (Show, Read)
```

Listing 8. The definition of a password wallet as a regular Haskell data type.

```haskell
genKey :: (Monad m, EntropyIO m) => Int -> m (PubKey, PrvKey)
```

The second aspect of IFC arises when the client machine queries the server for accuracy and loss by asking it to validate the model. Internally, when this call is made to the server equipped with the enclave, the enclave has to read a file with test data. This test data resides outside of the enclave and is potentially an attack vector. In order to not inadvertently trust such an exposed source, the enclave uses the `untrustedReadFile` function from the `RestrictedIO` typeclass. The file is marked as `Untrusted` and requires explicit programmer *endorsement* via the `trust` operator as shown below. The compiler will not typecheck if the user does not trust the file source.

```haskell
validate :: Enclave (Ref SrvSt) -> Enclave (Accuracy, Loss)
validate srv_ref_st = do
  -- ...
  testDataSet <- untrustedReadFile testfile
  let (x, y) = parseDataSet $ trust testDataSet -- trust the contents
  -- ...
  return (acc, loss)
```

Overall the case study constitutes only 500 lines of code. It naturally fits into the client-server programming model, and the usage of Haskell provides type safety and enables IFC-based security.

5.2 Encrypted Password Wallet

For this case study, we use HasTEE to implement a secure password wallet that stores authentication tokens in encrypted form on the disk. An authentication token can be retrieved from the wallet if the right master password is supplied. The definition of a password wallet used by the case study follows in Listing 8.

```
-- | Secure file path to the wallet
wallet :: SecureFilePath
wallet = secureFile "wallet.seal"

-- | Try to load the secure wallet into the enclave
loadWallet :: Enclave (Maybe Wallet)
loadWallet = do b <- doesSecureFileExist wallet
                if b then do contents <- readSecure wallet
                             return $ readMaybe contents
                     else return Nothing

-- | Store the wallet on disk in encrypted form
saveWallet :: Wallet -> Enclave ReturnCode
saveWallet w = writeSecure wallet (show w) >> return Success
```

Listing 9. The code that stores the wallet in encrypted form, and loads it up into the enclave memory. The only part of this code that indicates that encryption & decryption are taking place is the fact that we use a secure filepath. The programmer is relieved from dealing with encryption keys.

The `Show` and `Read` instances are used to convert a wallet to and from a string. This allows us to write the wallet to disk, and by writing to a secure file path we ensure that the stored wallet is encrypted, as described in section 4.5.1. By omitting a `Binary` instance we ensure that the wallet is not inadvertently leaked to the client directly. The code in Listing 9 implements the functions that store and load the wallet. We emphasize that the code does not need to explicitly reason about encryption and decryption, except for defining the secure file path.

Our password wallet has the following features - (1) adding an authentication token, (2) retrieving a password, (3) deleting a token and (4) changing the master password. It is designed as a command-line utility where the commands are handled by an untrusted client and the passwords are protected by the enclave. The complete implementation is roughly 200 lines of Haskell code.

The hardware-enforced security provided by our secure wallet makes it a natural fit for designing password wallets that are protected by biometrics. A similar approach is used on modern iPhones, where passwords are stored in a secure enclave [Apple 2021] to ensure confidentiality, and the user's biometric data is used as the master password. In our case, the usage of a high-level language like Haskell enables expressing this relatively complex application concisely.

5.3 Data Clean Room with Differential Privacy

A *Data Clean Room* (DCR) [AWS 2022] is a technology that provides aggregated and anonymised user information to protect user privacy while providing advertisers and analytic firms with non-personally identifiable information to target a specific demographic with advertising campaigns and analytics-based services.

A DCR can be provisioned with user data, but very strict privacy controls ensure that user data does not leave the DCR. The DCR can, however, compute aggregated results based on the user data within, and release that. The DCR can further employ *differential privacy* [Dwork 2006] as a security guarantee. Differential privacy is a guarantee that states that a query over a data set does not produce a result that makes it possible to say whether a specific individual was in the data set or not. This is done by adding a calibrated amount of noise to the result. The amount of noise can be calibrated to increase privacy (add more noise) or increase accuracy (add less noise).

Our third case study implements a DCR within an SGX enclave using HasTEE. To protect user confidentiality, the analytics applied in the DCR use basic differential privacy techniques (written in Haskell). User records are encrypted before they are provisioned to the DCR, after which we use the *Laplace Mechanism* [Dwork and Roth 2014] when performing counting queries to add noise to the result. The mechanism introduces noise by sampling a Laplace distribution. The code implementing the Laplace mechanism can be found in the appendix, in Listing 14. The definition of user records is shown in Listing 10.

```
data User = User { name :: String
                 , occupation :: Occupation
                 , salary :: Integer
                 , gender :: Gender
                 , age :: Integer
                 , origin :: Country
                 }
  deriving (Show, Read)
-- | The client can encrypt a user
encryptUser :: User -> PubKey -> Client [CipherText]
-- | The enclave can decrypt a user
decryptUser :: [CipherText]
            -> PrvKey
            -> PubKey
            -> Enclave (Maybe User)
```

Listing 10. Definition of user records that are recorded in the enclave. User records should only enter the enclave in encrypted form, so we expose two functions for encrypting and decrypting a user record.

The DCR does not provide a `Binary` instance for the `User` type to ensure that the user records are not transferred to the enclave via plain serialisation. Instead, we expose functions that encrypt and decrypt users. The `[CipherText]` type has a `Binary` instance, allowing for an encrypted user to be sent to the enclave.

The Laplace Mechanism used for adding noise requires a source of randomness. In our implementation, we use Haskell's `System.Random` package, which internally reads from `/dev/urandom`. Although the `/dev/urandom` file is located outside of the enclave, the Gramine library OS intercepts the application's read calls to `/dev/urandom`, and instead of reading from the file directly, it samples randomness

```
app :: App Done
app = do
    ref    <- liftNewRef undefined
    initSt <- secure $ initEnclave ref
    pkey   <- secure $ getPublicKey ref
    prov'  <- secure $ provisionUserEnclave ref
    lm     <- secure $ laplaceMechanism ref $ salaryWithin 10000 50000
    dataset <- liftIO $ sequence $ replicate 500 (generate arbitrary)
    runClient $ do
        onEnclave $ initSt <.> 0.1 -- initialize enclave with
                                   -- privacy budget
        key <- onEnclave pkey      -- fetch public key
        mapM_ (\u -> do ct <- encryptUser u key
                        onEnclave $ prov' <.> ct) dataset -- provision
                                                          -- users
        result <- onEnclave lm -- run the salary query
        liftIO $ putStrLn $ concat [ "randomized response: "
                                   , show result
                                   ]
```

Listing 12. The client running the salaryWithin query over the data set in the clean room.

from a secure source within the enclave. This secure source of randomness can be accessed through the `RestrictedIO` (4.5.1) typeclass.

An example query that one might want to run over the data set is to see how many individuals in the data set have a salary in a specific range. Constructing the query that answers this for a single individual is shown in Listing 11.

```
salaryWithin :: Integer -> Integer -> User -> Bool
salaryWithin l h u = l <= salary u && salary u <= h
```

Listing 11. A query that checks whether an individual has a salary within a certain bound.

The HasTEE code that implements the main method of this example is shown in listing 12. Lines 3 to 8 specify the API of the data clean room. The DCR's API supports (1) initialisation, (2) fetching of the public key, (3) provisioning user data to the enclave, and (4) executing the salary query. Line 8 is used to generate some arbitrary users (for testing), after which the client code takes over. The client initializes the DCR and fetches its public key. After this, the users are encrypted and sent to the DCR. On line 15 the salary query is executed in the DCR, and then the result is printed.

Generating arbitrary users to test the setup is done purely for illustration purposes. In a more faithful implementation, the client would relay the public key to data owners that would then send already encrypted user records to the client, which provisions them to the DCR. Owing to HasTEE's client-server programming model and the use of a high-level language like Haskell, the implementation becomes very compact with roughly 200 LOC.

6 EVALUATION

6.1 Discussion

Through the case studies presented in Section 5, we have demonstrated the applicability of HasTEE in a variety of domains, including privacy-preserving machine learning, hardware-encrypted wallets, and data clean rooms with differential privacy. In all of the case studies shown above, we rely on a particular TEE implementation - Intel SGX.

Considering the shortcomings of SGX, it has been shown to be vulnerable to a wide variety of side-channel attacks [Schaik et al. 2022]. HasTEE's API and implementation, however, are much more high-level and are not reliant on anything SGX-specific. A TEE design, such as ARM TrustZone, which provides a disjoint cache hierarchy, could alleviate several side-channel attacks affecting Intel SGX. The main effort then lies in porting the GHC runtime to the TrustZone infrastructure.

The concision and brevity provided by Haskell are notable in all three case studies. In contrast to development on the Intel C/C++ SGX SDK, HasTEE's high-level programming model entirely abstracts away the complexity of dealing with the low-level `edl` files in the SGX SDK. The remote procedure calls that happen between the untrusted client and trusted enclave are typechecked in Haskell, unlike the SGX SDK.

In terms of Information Flow Control, HasTEE's API represents a significant improvement over the SGX SDK. The latter does not warn or typecheck a program against accidental data leaks. On the other hand, HasTEE has stronger compile-time guarantees. For instance, in all three case studies, the lack of the `Binary` typeclass constraint would by construction prevent accidental leakage of the secret data from the enclave.

Additionally, in all three case studies, we observe the `RestrictedIO` typeclass constraining the I/O operations possible in the Enclave monad. Notably, in the federated learning example, we had to modify the homomorphic encryption library to limit its types that involved the `IO` monad. Finally, in the password wallet example, the `readSecure` and `writeSecure` relieves the enclave programmer from the burden of key management.

6.2 Comparing HasTEE to GoTEE and J_E

Implementation. Table 11 presents a comparison between HasTEE and its two closest counterparts - GoTEE [Ghosn et al. 2019] and J_E [Oak et al. 2021]. While both GoTEE and J_E had to modify the respective compilers, HasTEE required no modifications to the compiler. The specific runtime used by JE is not mentioned in the paper [Oak et al. 2021]; however, it suggests that no modification of the runtime was required, as it was run on a large virtualized host - SGX-LKL [Priebe et al. 2019]. In contrast, the runtimes for HasTEE and GoTEE required modification. GoTEE required significant modifications to the Golang runtime system to enable communication between the trusted and untrusted memory. The modifications required by HasTEE

Framework	HasTEE	GoTEE	J_E
IFC support	Standard declassification	None	Robust declassification
Partitioning scheme	Type-based	Process-based	Annotation-based
Modified compiler	No	Yes	Yes
Modified runtime system	Yes	Yes	No
Secure by construction	Yes	No	No
Trusted Components	GHC compiler, GHC runtime, Gramine	GoTEE compiler, GoTEE runtime	Java parser and partitioner, Jif compiler, JVM, SGX-LKL[Priebe et al. 2019]
Programming model	Client-server	Synchronous Message-Passing	Using the object-framework provided by Java

Fig. 11. Comparison of HasTEE, GoTee, and J_E. We specify the core components involved in the Trusted Computing Base in all three frameworks.

are all related to running any general Haskell program in the enclave (Section 4.4.1) rather than the library's primitives.

Secure by construction. Both GoTEE and J_E use sophisticated static analysis passes and program transformations to partition a program into its two components. These passes have to properly identify which code should go into each respective component and generate the correct RPC code. This is a complicated but very important process. In contrast, HasTEE does not require such analyses. HasTEE makes the assumption that the computation in the `Enclave` monad is secure, and computation in the `Client` monad is untrusted. Since the two components are not meant to rely on each other's logic, the same program is compiled twice. When the `Enclave` is compiled the `Client` monad is swapped for a dummy implementation, and vice versa. Fig. 1 in the Introduction (Sec 1) conveyed this general idea.

7 RELATED WORK

In this section, we will compare the key aspects of HasTEE with its closely related counterparts.

Programming Language support for TEE programming. HasTEE presents, to the best of our knowledge, one of the first instances of a functional language, Haskell,

running on a TEE environment. Among imperative and object-oriented languages, GoTEE [Ghosn et al. 2019] extends the Go programming language with a feature called *secured routines* to model message-passing between an untrusted client and a trusted enclave. Similarly, J_E [Oak et al. 2021] adds security-based annotations to a subset of Java, which allows partitioning Java applications into trusted and untrusted components and then enforcing information flow control across the trusted boundary. Civet [C. Tsai, Son, et al. 2020] is another example of a partitioned Java runtime for enclaves.

In the Rust ecosystem, the Rust-SGX [Wang et al. 2019] project provides foreign-function interface (FFI) bindings to the C/C++ Intel SGX SDK. The goal of the project is different from HasTEE in that it does not aim to introduce any programming model for programming TEEs. The main goal of Rust SGX is to introduce application-level memory safety when programming with the low-level SGX SDK. HasTEE provides memory safety by the virtue of running on an already memory-safe language, Haskell. TrustJS [Goltzsche et al. 2017] takes a similar FFI-based approach as Rust-SGX for programming enclaves with JavaScript.

An important project in this space is the WebAssembly (WASM) initiative [Rossberg 2019]. There have been WASM projects, both academic, such as Twine [Ménétrey et al. 2021], as well as commercial, such as Enarx [Red Hat 2019], aimed at allowing WASM runtimes to operate within SGX enclaves. Our initial approach was to use the experimental Haskell WASM backend [Tweag.io 2022] to run Haskell on SGX enclaves. However, the aforementioned runtimes are not supported by GHC and lack several key features required for loading Haskell onto an enclave.

Partitioning application and programming model. An important contribution of HasTEE is the seamless program partitioning and familiar client-server-based programming model for enclaves. The partitioning approach in HasTEE has been adapted from the Haste.App library [Ekblad and Claessen 2014]. The most well-known automatic partitioning tool for C programs on an SGX enclave is Glamdring [Lind et al. 2017]. The general idea of partitioning a single program, irrespective of the application domain, has been studied as multitier programming [Weisenburger et al. 2021]. Among the existing approaches to multitier programming, HasTEE provides a lightweight alternative that does not require any compiler modification or elaborate dataflow analysis to partition the program.

In terms of various programming models, GoTEE utilizes a synchronous message-passing-based approach. A similar but asynchronous message-passing-based model is given by EActors [Sartakov et al. 2018], an actor-model-based framework for confidential computing, written in C.

Intel SGX application development. The partitioning-based two-project programming model of Intel SGX has been a significant obstacle to the technology's adoption. As such, there have been attempts to virtualize entire platforms within the enclave memory. Haven [Baumann et al. 2015] virtualizes the entire Windows operating system as well as an entire SQL server application running on top of it. SCONE [Arnautov et al. 2016] virtualizes a Docker container instance within an SGX

enclave. The Gramine [C. Tsai, Porter, et al. 2017] library operating system, which is used in this work, is also an example of lightweight virtualization.

While a larger body of work has been dedicated to Intel SGX, AMD introduced TEEs in the form of AMD SEV [AMD 2018], which is natively a virtualization-based approach. Virtualization can result in drastically increasing the size of the Trusted Computing Base (TCB). Hence, in HasTEE, we chose the approach with the smallest TCB of 57k lines of code (Gramine). We intend to eventually move away from Gramine and make the GHC runtime a standalone library inside the SGX enclave (see Section 8).

Information Flow Control for enclaves. HasTEE draws inspiration from the library-based lightweight Information Flow Control (IFC) approaches in Haskell [Buiras et al. 2015; Russo 2015; Russo et al. 2008] for its IFC implementation. This approach relies on the purity of Haskell to detect and stop malicious behaviour. In contrast, J_E [Oak et al. 2021] compiles down to the Jif Java compiler [Myers 1999], which statically checks confidentiality and integrity policies. J_E employs a stronger IFC policy based on robust declassification [Myers et al. 2004]. This policy ensures that low-integrity data cannot influence the declassification of secret data. We relax this policy in favour of a more user-friendly API.

Another interesting line of work is Moat [Sinha et al. 2015], which formally models and verifies enclave programs running on Intel SGX such that data confidentiality is respected. It uses IFC to enforce the policies and automated theorem proving to verify the policy enforcement mechanism.

8 FUTURE WORK

This paper lays the foundation for several threads of future work.

- The IFC scheme in HasTEE operates on two security levels - private (`Enclave`) and public (`Client`). A natural extension of this work is to enable the use of a hierarchical security *lattice* consisting of multiple security levels, where each security level is mapped to unique enclaves, allowing multi-enclave IFC.
- Another promising line of work is *Remote Attestation* [Knauth et al. 2018], which allows an SGX enclave to prove its identity to a challenger using the private key embedded in the enclave. This unique identification property can facilitate secure communication between multiple enclaves, both in a multi-enclave and a distributed-enclave setting. It would also be interesting to see if there could be language-level support for remote attestation.
- Currently, it is unclear how to bring legacy packages into the enclave without extensive modification. Additionally, the `RestrictedIO` typeclass makes any library that uses IO unusable without modification. Future work should look into more principled ways of bringing legacy packages into the enclave while retaining the ability to analyze their effects and understand their implications for a security policy.

- A significant obstacle when developing HasTEE was running Haskell programs on an Intel SGX enclave. While the GHC runtime is extensively optimized for performance, having a more compact and portable runtime of GHC that sacrifices some performance by using a restricted set of `libc` operations could result in a considerably smaller Trusted Computing Base. A more portable runtime would facilitate HasTEE experiments on other TEE infrastructures such as ARM TrustZone and RISC-V PMP [RISC-V 2017].

9 CONCLUSION

In this paper, we introduce HasTEE - a type-safe library in Haskell that enables the programming of Trusted Execution Environments using a familiar client-server-based programming model. HasTEE raises the level of abstraction over the low-level and error-prone programming APIs provided by TEE SDKs. Additionally, the partitioning framework proposed by HasTEE is secure by construction, which is an improvement over other high-level imperative language frameworks.

HasTEE being written in a pure and statically-typed functional language allows enforcing language-based information flow control mechanisms that protect data confidentiality. We have demonstrated through three diverse case studies how HasTEE's IFC mechanism can help prevent accidental data leakage while producing concise code. We hope that HasTEE will open up future research avenues at the intersection of hardware-based and programming-language-based security.

REFERENCES

Mohammad Al-Rubaie and J. Morris Chang. 2019. "Privacy-Preserving Machine Learning: Threats and Solutions." *IEEE Secur. Priv.*, 17, 2, 49–58. DOI: 10.1109/MSEC.2018.2888775.

AMD. 2018. *AMD*. https://developer.amd.com/sev/. (accessed: 16.02.2023).

Apple. 2021. *Apple Platform Security*. https://support.apple.com/guide/security/face-id-and-touch-id-security-sec067eb0c9e/web. (accessed: 16.02.2023).

ARM. 2004. *ARM TrustZone*. https://www.arm.com/technologies/trustzone-for-cortex-a. (accessed: 16.02.2023).

Sergei Arnautov et al.. 2016. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. 2008. "Cryptographically-masked flows." *Theor. Comput. Sci.*, 402, 2-3, 82–101. DOI: 10.1016/j.tcs.2008.04.028.

Lennart Augustsson and Bart Massey. 2013. *Text.Printf - printf in Haskell*. https://hackage.haskell.org/package/base-4.17.0.0/docs/Text-Printf.html#t:PrintfType. (accessed: 16.02.2023).

Amazon AWS. 2022. *AWS Clean Room*. https://aws.amazon.com/clean-rooms/. (accessed: 16.02.2023).

Zijian Bao, Qinghao Wang, Wenbo Shi, Lei Wang, Hong Lei, and Bangdao Chen. 2020. "When Blockchain Meets SGX: An Overview, Challenges, and Open Issues." *IEEE Access*, 8, 170404–170420. DOI: 10.1109/ACCESS.2020.3024254.

Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. "Shielding Applications from an Untrusted Cloud with Haven." *ACM Trans. Comput. Syst.*, 33, 3, 8:1–8:26. DOI: 10.1145/2799647.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. "HLIO: mixing static and dynamic typing for information-flow control in Haskell." In: *Proceedings of the 20th ACM SIGPLAN International Conference*

on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. Ed. by Kathleen Fisher and John H. Reppy. ACM, 289–301. DOI: 10.1145/2784731.2784758.

Stephen Checkoway and Hovav Shacham. 2013. "Iago attacks: why the system call API is a bad untrusted RPC interface." In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. Ed. by Vivek Sarkar and Rastislav Bodik. ACM, 253–264. DOI: 10.1145/2451116.2451145.

Decentriq. 2022. *Decentriq*. https://blog-decentriq-com.cdn.ampproject.org/c/s/blog.decentriq.com/swiss-cheese-to-cheddar-securing-amd-sev-snp-early-boot-2. (accessed: 16.02.2023).

Dorothy E. Denning. 1976. "A Lattice Model of Secure Information Flow." *Commun. ACM*, 19, 5, 236–243. DOI: 10.1145/360051.360056.

Cynthia Dwork. 2006. "Differential Privacy." In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Vol. 4052. Springer, 1–12. DOI: 10.1007/11787006_1.

Cynthia Dwork and Aaron Roth. 2014. "The Algorithmic Foundations of Differential Privacy." *Found. Trends Theor. Comput. Sci.*, 9, 3-4, 211–407. DOI: 10.1561/0400000042.

Anton Ekblad and Koen Claessen. 2014. "A seamless, client-centric programming model for type safe web applications." In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 79–89. DOI: 10.1145/2633357.2633367.

Richard Felker. 2005. *musl C Library*. https://musl.libc.org/. (accessed: 16.02.2023).

Craig Gentry. 2009. "Fully homomorphic encryption using ideal lattices." In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. Ed. by Michael Mitzenmacher. ACM, 169–178. DOI: 10.1145/1536414.1536440.

Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. "Secured Routines: Language-based Construction of Trusted Execution Environments." In: *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. Ed. by Dahlia Malkhi and Dan Tsafrir. USENIX Association, 571–586. https://www.usenix.org/conference/atc19/presentation/ghosn.

GNUDevs. 1991. *glibc - The GNU C Library*. https://www.gnu.org/software/libc/. (accessed: 16.02.2023).

Joseph A. Goguen and José Meseguer. 1982. "Security Policies and Security Models." In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. DOI: 10.1109/SP.1982.10014.

David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter R. Pietzuch, and Rüdiger Kapitza. 2017. "TrustJS: Trusted Client-side Execution of JavaScript." In: *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. Ed. by Cristiano Giuffrida and Angelos Stavrou. ACM, 7:1–7:6. DOI: 10.1145/3065913.3065917.

Daniel Hedin and Andrei Sabelfeld. 2012. "A Perspective on Information-Flow Control." In: *Software Safety and Security - Tools for Analysis and Verification*. NATO Science for Peace and Security Series - D: Information and Communication Security. Vol. 33. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. IOS Press, 319–347. DOI: 10.3233/978-1-61499-028-4-319.

Intel. 2016. *Intel SGX SDK*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html. (accessed: 16.02.2023).

Intel. 2015. *Intel Software Guard Extension*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. (accessed: 16.02.2023).

Intel. 2018a. *SGX Tutorial*. https://www.intel.com/content/www/us/en/developer/articles/training/intel-software-guard-extensions-tutorial-part-1-foundation.html. (accessed: 16.02.2023).

Intel. 2018b. *tlibc - an alternative to glibc*. https://github.com/intel/linux-sgx/tree/master/common/inc/tlibc. (accessed: 16.02.2023).

SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. "The Glasgow Haskell compiler: a technical overview." In: *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*. Vol. 93.

Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. "Integrating Remote Attestation with Transport Layer Security." *CoRR*, abs/1801.05863. http://arxiv.org/abs/1801.05863 arXiv: 1801.05863.

Paul Kocher et al.. 2018. "Spectre Attacks: Exploiting Speculative Execution." *CoRR*, abs/1801.01203. http://arxiv.org/abs/1801.01203 arXiv: 1801.01203.

Michael Larabel. 2020. *Linux Stats*. https://www.phoronix.com/news/Linux-Git-Stats-EOY2019. (accessed: 16.02.2023).

Christian M. Lesjak, Daniel M. Hein, and Johannes Winter. 2015. "Hardware-security technologies for industrial IoT: TrustZone and security controller." In: *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society, Yokohama, Japan, November 9-12, 2015*. IEEE, 2589–2595. DOI: 10.1109/IECON.2015.7392493.

Joshua Lind et al.. 2017. "Glamdring: Automatic Application Partitioning for Intel SGX." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

Moritz Lipp et al.. 2018. "Meltdown." *CoRR*, abs/1801.01207. http://arxiv.org/abs/1801.01207 arXiv: 1801.01207.

Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. 2009. "Runtime support for multicore Haskell." In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 65–78. DOI: 10.1145/1596550.1596563.

Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. "Twine: An Embedded Trusted Runtime for WebAssembly." In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 205–216. DOI: 10.1109/ICDE51399.2021.00025.

Microsoft. 1994. *glibc - The GNU C Library*. C%20runtime%20(CRT)%20and%20C++%20standard%20library%20(STL). (accessed: 16.02.2023).

Dominic P. Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo J. M. Vincent. 2021. "Confidential Computing - a brave new world." In: *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*. IEEE, 132–138. DOI: 10.1109/SEED51797.2021.00025.

Andrew C. Myers. 1999. "JFlow: Practical Mostly-Static Information Flow Control." In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 228–241. DOI: 10.1145/292540.292561.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. 2004. "Enforcing Robust Declassification." In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 172–186. DOI: 10.1109/CSFW.2004.9.

Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi. 2021. "Language Support for Secure Software Development with Enclaves." In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. DOI: 10.1109/CSF51468.2021.00037.

Pascal Paillier. 1999. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." In: *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding* (Lecture Notes in Computer Science). Ed. by Jacques Stern. Vol. 1592. Springer, 223–238. DOI: 10.1007/3-540-48910-X_16.

Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. 2019. "SGX-LKL: Securing the Host OS Interface for Trusted Execution." *CoRR*, abs/1908.11143. http://arxiv.org/abs/1908.11143 arXiv: 1908.11143.

Inc. Red Hat. 2019. *Enarx: Confidential Computing with WebAssembly*. https://enarx.dev/. (accessed: 16.02.2023).

RISC-V. 2017. *RISC-V: Physical Memory Protection*. https://riscv.org/blog/2022/04/xuantie-virtualzone-risc-v-based-security-extensions-xuan-jian-alibaba/. (accessed: 16.02.2023).

Andreas Rossberg. Dec. 5, 2019. *WebAssembly Core Specification*. W3C, (Dec. 5, 2019). https://www.w3.org/TR/wasm-core-1/.

Alejandro Russo. 2015. "Functional pearl: two can keep a secret, if one of them uses Haskell." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 280–288. DOI: 10.1145/2784731.2784756.

Alejandro Russo, Koen Claessen, and John Hughes. 2008. "A library for light-weight information-flow security in haskell." In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 13–24. DOI: 10.1145/1411286.1411289.

Andrei Sabelfeld and David Sands. 2005. "Dimensions and Principles of Declassification." In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 255–269. DOI: 10.1109/CSFW.2005.15.

Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. "EActors: Fast and flexible trusted computing using SGX." In: *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*. Ed. by Paulo Ferreira and Liuba Shrira. ACM, 187–200. DOI: 10.1145/3274808.3274823.

Stephan van Schaik et al.. 2022. "SoK: SGX.Fail: How Stuff Get eXposed." In.

Hovav Shacham. 2007. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 552–561. DOI: 10.1145/1315245.1315313.

Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2015. "Moat: Verifying Confidentiality of Enclave Programs." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 1169–1184. DOI: 10.1145/2810103.2813608.

Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves." In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 505–522. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai.

Li-Ting Tsai and Abhiroop Sarkar. 2016. *HsPaillier - Partial Homomorphic Encryption in Haskell*. https://hackage.haskell.org/package/Paillier. (accessed: 16.02.2023).

Tweag.io. 2022. *Haskell WASM backend*. https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc/. (accessed: 16.02.2023).

Huibo Wang et al.. 2019. "Towards Memory Safe Enclave Programming with Rust-SGX." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2333–2350. DOI: 10.1145/3319535.3354241.

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2021. "A Survey of Multitier Programming." *ACM Comput. Surv.*, 53, 4, 81:1–81:35. DOI: 10.1145/3397495.

Dmitry P. Zegzhda, E. S. Usov, V. A. Nikol'skii, and Evgeny Pavlenko. 2017. "Use of Intel SGX to ensure the confidentiality of data of cloud users." *Autom. Control. Comput. Sci.*, 51, 8, 848–854. DOI: 10.3103/S0146411617080284.

Chongchong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and Chunxiao Xing. 2016. "On the Performance of Intel SGX." In: *13th Web Information Systems and Applications Conference, WISA 2016, Wuhan, China, September 23-25, 2016*. IEEE, 184–187. DOI: 10.1109/WISA.2016.45.

A EXECUTABLE OPERATIONAL SEMANTICS OF HASTEE IN HASKELL

```haskell
{-# LANGUAGE FlexibleContexts #-}
module HasTEEOrig where

import Control.Monad.State.Class
import Control.Monad.State.Strict



type Name = String

data Exp = Lit Int
         | Var Name
         | Fun [Name] Exp
         | App Exp [Exp]
         | Let Name Exp Exp
         | Plus Exp Exp

         -- HasTEE operators
         | Remote    Exp
         | OnServer  Exp
         | RemoteApp Exp Exp -- (<.>)
         deriving (Show)

data Value = IntVal Int
           | Closure [Name] Exp Env
           -- HasTEE values
           | RemoteClosure Name [Value]
           | ArgList [Value]
           | Dummy

           -- Error values
           | Err ErrState
           deriving (Show)

data ErrState = ENotClosure
              | EVarNotFound
              | ENotRemClos
              | ENotIntLit

instance Show ErrState where
  show ENotClosure  = "Closure not found"
  show EVarNotFound = "Variable not in environment"
  show ENotRemClos  = "Remote Closure not found"
  show ENotIntLit   = "Not an integer literal"

type Env = [(Name, Value)]
```

```
47
48 type ClientEnv  = Env
49 type EnclaveEnv = Env
50
51
52 type VarName = Int
53
54
55 data StateVar =
56   StateVar { varName  :: Int
57            , encState :: EnclaveEnv
58            }
59
60 initStateVar :: EnclaveEnv -> StateVar
61 initStateVar = StateVar 0
62
63
64 eval :: Exp -> Value
65 eval e =
66   let newEnclaveEnv = snd $
67                       evalState (evalEnclave e initEnclaveEnv)
68                       (initStateVar initEnclaveEnv)
69   in fst $ evalState (evalClient e initClientEnv) (initStateVar
       newEnclaveEnv)
70   where
71     initEnclaveEnv = []
72     initClientEnv  = []
73
74 genRemVar :: (MonadState StateVar m) => m String
75 genRemVar = do
76   n <- gets varName
77   modify $ \s -> s {varName = 1 + n}
78   pure ("RemVar" <> show n)
79
80 evalList :: (MonadState StateVar m) => [Exp] -> Env -> [Value] -> m ([
       Value], Env)
81 evalList []     e vals = pure (reverse vals, e)
82 evalList (e1:es) env xs = do
83   (v, e) <- evalEnclave e1 env
84   evalList es e (v:xs)
85
86
87 evalEnclave :: (MonadState StateVar m)
88             => Exp -> EnclaveEnv -> m (Value, EnclaveEnv)
89 evalEnclave (Lit n) env = pure (IntVal n, env)
90 evalEnclave (Var x) env = pure (lookupVar x env, env)
```

```
91 evalEnclave (Fun xs e) env =
92   pure (Closure xs e env, env)
93 evalEnclave (Let name e1 e2) env = do
94   (e1', env') <- evalEnclave e1 env
95   evalEnclave e2 ((name,e1'):env')
96 evalEnclave (App f args) env = do
97   (v1, env1) <- evalEnclave f env
98   (vals, env2) <- evalList args env1 []
99   case v1 of
100    Closure xs body ev ->
101      evalEnclave body ((zip xs vals) ++ ev)
102    _ -> pure (Err ENotClosure, env2)
103 evalEnclave (Plus e1 e2) env = do
104   (v1, env1) <- evalEnclave e1 env
105   (v2, env2) <- evalEnclave e2 env1
106   case (v1, v2) of
107    (IntVal a1, IntVal a2) -> pure (IntVal (a1 + a2), env2)
108    _ -> pure (Err ENotIntLit, env2)
109
110 evalEnclave (Remote e) env = do
111   (val, env') <- evalEnclave e env
112   varname     <- genRemVar
113   let env'' = (varname, val):env'
114   pure (Dummy, env'')
115 -- the following two are the essentially no-ops
116 evalEnclave (OnServer e) env = evalEnclave e env
117 evalEnclave (RemoteApp e1 e2) env = do
118   (_, env1) <- evalEnclave e1 env
119   (_, env2) <- evalEnclave e2 env1
120   pure (Dummy, env2)
121
122 evalList2 :: (MonadState StateVar m) => [Exp] -> Env -> [Value] -> m
      ([Value], Env)
123 evalList2 []      e vals = pure (reverse vals, e)
124 evalList2 (e1:es) env xs = do
125   (v, e) <- evalClient e1 env
126   evalList2 es e (v:xs)
127
128
129 evalClient :: (MonadState StateVar m)
130            => Exp -> ClientEnv -> m (Value, ClientEnv)
131 evalClient (Lit n) env = pure (IntVal n, env)
132 evalClient (Var x) env = pure (lookupVar x env, env)
133 evalClient (Fun xs e) env =
134   pure (Closure xs e env, env)
135 evalClient (Let name e1 e2) env = do
```

```
136   (e1', env') <- evalClient e1 env
137   evalClient e2 ((name,e1'):env')
138 evalClient (App f args) env = do
139   (v1, env1) <- evalClient f env
140   (v2, env2) <- evalList2 args env1 []
141   case v1 of
142     Closure xs body ev ->
143       evalClient body ((zip xs v2) ++ ev)
144     _ -> pure (Err ENotClosure, env2)
145 evalClient (Plus e1 e2) env = do
146   (v1, env1) <- evalClient e1 env
147   (v2, env2) <- evalClient e2 env1
148   case (v1, v2) of
149     (IntVal a1, IntVal a2) -> pure (IntVal (a1 + a2), env2)
150     _ -> pure (Err ENotIntLit, env2)
151
152
153 evalClient (Remote e) env = do
154   (_, env') <- evalClient e env
155   varname     <- genRemVar
156   let env'' = (varname, Dummy):env'
157   pure (RemoteClosure varname [], env'')
158 evalClient (OnServer e) env = do
159   (e', env1) <- evalClient e env
160   case e' of
161     RemoteClosure varname vals -> do
162       enclaveEnv <- gets encState
163       let func = lookupVar varname enclaveEnv
164       case func of
165         Closure vars body environ -> do
166           (res,enclaveEnv') <- evalEnclave body ((zip vars vals) ++
      environ)
167           pure (res, env1)
168         _ -> pure (Err ENotClosure, env1)
169     _ -> pure (Err ENotRemClos, env1)
170 evalClient (RemoteApp e1 e2) env = do
171   (v1, env1) <- evalClient e1 env
172   (v2, env2) <- evalClient e2 env1
173   case v1 of
174     RemoteClosure f args ->
175       case v2 of
176         ArgList vals -> pure (RemoteClosure f (args ++ vals), env2)
177         v -> pure (RemoteClosure f (args ++ [v]), env2)
178     v -> pure (ArgList [v,v2], env2)
179
180
```

```
-- onServer (f == RemoteClosure f [])
-- onServer (f <.> arg == RA f arg == RemoteClosure f [arg])
-- onServer (f <.> arg1 <.> arg2 == RA f (RA arg1 arg2) == RC f [arg1, arg2])

lookupVar :: String -> [(String, Value)] -> Value
lookupVar _ [] = Err EVarNotFound
lookupVar x ((y, v) : env) =
  if x == y then v else lookupVar x env
```

Listing 13. Operational Semantics of HasTEE

B DATA CLEAN ROOM CODE

```
countingQuery :: Enclave (Ref CleanRoomSt) -> (User -> Bool) -> Enclave Int
countingQuery refst q = do
    st <- readRef =<< refst
    return $ length $ filter id $ map q (users st)

laplaceDistribution :: Enclave (Ref CleanRoomSt) -> Double -> Enclave Double
laplaceDistribution refst b = do
    z <- int2Double <$> getRandom refst (0,1)
    u <- ((/) 1000 . int2Double) <$> getRandom refst (1,1000)
    return $ (2 * z - 1) * (b * log u)

laplaceMechanism :: Enclave (Ref CleanRoomSt) -> (User -> Bool) -> Enclave Double
laplaceMechanism refst q = do
    st <- readRef =<< refst
    true <- int2Double <$> countingQuery refst q
    noise <- laplaceDistribution refst (1 / (epsilon st))
    return $ true + noise
```

Listing 14. Code for the laplace mechanism.