HasTEE$^+$: Confidential Cloud Computing and Analytics with Haskell

Abhiroop Sarkar[0000−0002−8991−9472] and Alejandro Russo[0000−0002−4338−6316]

Chalmers University, Gothenburg, Sweden
{sarkara,russo}@chalmers.se

Abstract. Confidential computing is a security paradigm that enables the protection of confidential code and data in a co-tenanted cloud deployment using specialized hardware isolation units called Trusted Execution Environments (TEEs). By integrating TEEs with a Remote Attestation protocol, confidential computing allows a third party to establish the integrity of an *enclave* hosted within an untrusted cloud. However, TEE solutions, such as Intel SGX and ARM TrustZone, offer low-level C/C++-based toolchains that are susceptible to inherent memory safety vulnerabilities and lack language constructs to monitor explicit and implicit information-flow leaks. Moreover, the toolchains involve complex multi-project hierarchies and the deployment of hand-written attestation protocols for verifying *enclave* integrity.
We address the above with HasTEE$^+$, a domain-specific language (DSL) embedded in Haskell that enables programming TEEs in a high-level language with strong type-safety. HasTEE$^+$ assists in multi-tier cloud application development by (1) introducing a *tierless* programming model for expressing distributed client-server interactions as a single program, (2) integrating a general remote-attestation architecture that removes the necessity to write application-specific cross-cutting attestation code, and (3) employing a dynamic information flow control mechanism to prevent explicit as well as implicit data leaks. We demonstrate the practicality of HasTEE$^+$ through a case study on confidential data analytics, presenting a data-sharing pattern applicable to mutually distrustful participants and providing overall performance metrics.

Keywords: Confidential Computing · Trusted Computing · Trusted Execution Environments · Information Flow Control · Attestation · Haskell.

1 Introduction

Confidential computing [28] is an emerging security paradigm that ensures the isolation of sensitive computations and data during processing, shielding them from potential threats within the underlying infrastructure. It accomplishes this by employing specialised hardware isolation units known as Trusted Execution Environments (TEEs). A TEE unit, such as the Intel SGX [24] or ARM TrustZone [1], 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, BIOS and other firmware.

TEEs further allow a remote party to establish *trust* by providing a measurement of the sensitive code and data, composing a signed *attestation report* that can be verified. As such, TEEs have been heralded as a leading contender to enforce a strong notion of *trust* within cloud computing infrastructure[4,33,50], particularly in regulated industries such as healthcare, law and finance [12].

However, an obstacle in the wide-scale adoption of confidential computing has been the awkward programming model of TEEs [47]. TEEs, such as Intel SGX, involve partitioning the state of the program into a trusted project linked with Intel-supplied restricted C standard library [19] and an untrusted project that communicates with the trusted project using custom data copying protocols [18]. The complexity is compounded for distributed, multi-tiered cloud applications due to the semantic friction in adhering to various data formats and protocols across multiple projects [32], resulting in increased developer effort [30].

Furthermore, a well-known class of security vulnerabilities [8,39] arise from the memory-unsafety of the TEE projects. Given the security-critical nature of TEE applications, efforts have been made to introduce Rust-based [48] and Golang-based SDKs [13] aimed at mitigating memory unsafety vulnerabilities. However, the same applications remain vulnerable to unintended information leaks [34]. Consider the following *trusted* function hosted within a TEE:

```c
#include "library.h" //provides `int computeIdx(char *)`
char *secret = "...";
int public_arr [15] = ....;
void trusted_func(char *userinput, int inputsize) {
   if (memcmp(secret, userinput, inputsize) == 0) {
     int val = computeIdx(userinput);
     ocall_printf("%d\n", public_arr[val]); // handwritten printf routine
   } else { ocall_printf("0\n"); }
}
```

In the above program, at least five attack vectors are present - (1) the `inputsize` parameter can be abused to cause a buffer-overflow attack, (2) the `userinput` parameter can be tampered by a malicious operating system to force an incorrect branching, (3) even with the correct `inputsize` and `userInput`, the attacker can observe `stdout` to learn which branch was taken, (4) the trusted library function - `computeIdx` - could intentionally (if written by a malicious party) or accidentally leak `secret`, and (5) finally the attacker can use timing-based side channels to learn the branching information or even the `secret` [7].

Our contribution through this work is – *HasTEE*$^+$ – a Domain Specific Language (DSL) embedded in Haskell and targeted towards confidential computing. HasTEE$^+$ is designed to mitigate at least four of the five attack vectors mentioned above. Additionally, it offers a *tierless* programming model, thereby simplifying the development of distributed, multi-tiered cloud applications.

HasTEE$^+$ builds on the HasTEE [37] project, which crucially provides a *Glasgow Haskell Compiler (GHC)* runtime [23] capable of running Haskell on

Intel SGX machines. While using a memory-safe language like Haskell or Rust mitigates attack vector (1), all the other attack vectors remain open in the HasTEE project. HasTEE$^+$ notably adds support for a general one-time-effort remote-attestation infrastructure that helps mitigate the attack vector (2). While the underlying protocol employs Intel's RA-TLS [21], HasTEE$^+$ ensures that programmers are not required to create custom attestation code for capturing and sending measurements or conducting integrity checks [22].

For attack vectors (3) and (4), HasTEE$^+$ adds support for a dynamic information flow control (IFC) mechanism based on the Labeled IO Monad (LIO) [43]. Accordingly, we adopt a *floating-label* approach from OSes such as HiStar [51], enabling HasTEE$^+$ to relax some of the impractical I/O restrictions in the original HasTEE project. Side-channel attacks (attack vector (5)) remain outside the scope of HasTEE$^+$.

Concerning the complex programming model, modern TEE incarnations like AMD SEV-SNP [38] and Intel TDX [20] introduce a virtualization-based solution, opting to virtualize the entire project instead of partitioning it into trusted and untrusted components. At the cost of an increased trusted code base, this approach simplifies the TEE project layout. Nevertheless, it remains vulnerable to the complexities of a multi-tiered cloud application, as well as all five of the aforementioned attack vectors.

The *tierless* programming model of HasTEE$^+$ expresses multi-client-server projects as a single program and uses the Haskell type system to distinguish individual clients. A separate monadic type, such as `Client "client1" a`, demarcates each individual client, while HasTEE's multi-compilation tactic partitions the program. This programming model, evaluated on Intel SGX, remains applicable on newer Intel TDX machines.

Contributions. We summarize the key contributions of HasTEE$^+$ here:

– HasTEE$^+$ introduces a *tierless* DSL (Section 3.1), capable of expressing multi-tiered confidential computing applications as a single program, increasing an application's comprehensibility and reducing developer effort.
– HasTEE$^+$ incorporates a remote attestation design (Section 3.2) that relieves programmers from crafting custom integrity checks and attestation setups.
– HasTEE$^+$ integrates dynamic information flow control mechanisms (Section 3.3) to prevent explicit and implicit information leaks from applications.
– We use HasTEE$^+$'s IFC mechanism and cryptography in a *data clean room* case study (Section 4), showing a general data sharing pattern for conducting analytics on confidential data among mutually distrustful participants.

2 Threat Model

We build upon the threat model of the HasTEE project [37] and other related works on Intel SGX [3,4,13,48]. In such a threat model, an attacker attempts to compromise the code and data memory within the TEE. The attacker has administrative access to the operating system, hypervisor and other related system software hosted on a malicious cloud service.

We expand the above threat model to include an *active attacker* attempting to compromise the integrity of the data flowing into the TEE, as well as a *passive attacker* who observes the public channels that the trusted software interacts with to learn more about its behaviour. The threat model terminology is adopted from the J_E project [31], and related attacks are discussed in subsequent sections. Another class of potential threats emerge from the inclusion of public software libraries into TEE software, such as cryptography libraries, which might accidentally or intentionally leak secrets [25].

HasTEE$^+$'s attestation infrastructure, based on Intel's RA-TLS protocol [21] accounts for masquerading attacks [15,45]. Availability attacks such as denial-of-service and hardware side-channel attacks are outside the scope of this work.

3 The HasTEE$^+$ DSL

We illustrate the key APIs of HasTEE$^+$ using a password checker application in Listing 1, and explain the individual types and functions in the subsequent sections. Notably, the entire application, consisting of a separate client and server, can be expressed within 27 lines of Haskell code (excluding import declarations).

Listing 1 shows the user *Alice* storing her password in the TEE memory and deploying the trusted function `pwdChecker` to conduct a password check. The application uses three key types – `EnclaveDC`, `Client`, and `App`, adapted from HasTEE [37]. All three types implements a monadic interface, denoted as `m`, constructed using fundamental operations `return :: a -> m a` and `(>>=) :: m a -> (a -> m b) -> m b` (read as *bind*). The `return x` operation produces a computation returning the value of x without side effects, while the `(>>=)` function sequences computations and their associated side effects. In Haskell, we often use the *do-notation* to express such monadic computations.

The `EnclaveDC` monad represents the trusted computations that get loaded onto a TEE. The name *Enclave* alludes to an Intel SGX *enclave*, while *DC* stands for *Disjunction Category*, which we further explain in Section 3.3.

The client-side of the application is represented by the namesake `Client` monad that captures a *type-level* string - `"client"`. The type-level string allows the Haskell type-system to distinguish between multiple clients and serves as an identifier for the monadic computation runner function - `runAppRA` on line 27.

The monad `App` serves as a staging area where the enclave data (`"password"`) and the enclave computation (`pwdChecker`) are loaded into trusted memory. Also, the `client` function is provided with the `API` it will use to communicate with the TEE within the `App` monad. We provide the type signatures of a simplified subset of the HasTEE$^+$ APIs used in Listing 1 for loading trusted data and computations, and running the computations, in Fig. 1 below.

The application begins with the `app` function (lines 17-25). The functions `inEnclaveLabeledConstant` and `inEnclave` are used to load the trusted data (line 19) and the trusted function (line 21) in the enclave, respectively. The `runClient` at line 22 runs the monadic `Client` computations. Note, there is no

```
 1 pwdChecker :: EnclaveDC (DCLabeled String) -> String -> EnclaveDC Bool
 2 pwdChecker pwd guess = do
 3   l_pwd <- pwd
 4   priv  <- getPrivilege
 5   p     <- unlabelP priv l_pwd
 6   if p == guess then return True else return False
 7
 8 data API = API { checkpwd :: Secure (String -> EnclaveDC Bool) }
 9
10 client :: API -> Client "client" ()
11 client api = do
12   liftIO $ putStrLn "Enter your password:"
13   userInput <- liftIO getLine
14   res <- gatewayRA ((checkpwd api) <@> userInput)
15   liftIO $ putStrLn ("Login returned " ++ show res)
16
17 app :: App Done
18 app = do
19   pwd   <- inEnclaveLabeledConstant pwdLabel "password"
20   let priv = toCNF "Alice"
21   efunc <- inEnclave (dcDefaultState priv) $ pwdChecker pwd
22   runClient (client (API efunc))
23   where
24     pwdLabel :: DCLabel
25     pwdLabel = "Alice" %% "Alice" :: DCLabel
26
27 main = runAppRA "client" app >> return ()
```

Listing 1: A password checker application in HasTEE$^+$

```
inEnclave :: Label l => LIOState l p -> a -> App (Secure a)
inEnclaveLabeledConstant
          :: Label l => l -> a -> App (EnclaveDC (DCLabeled a))
gatewayRA :: (Binary a, Label l)
          => Secure (Enclave l p a) -> Client loc a
(<@>)     :: Binary a => Secure (a -> b) -> a -> Secure b
runClient :: Client loc a -> App Done
runAppRA  :: Identifier   -> App a -> IO a
```

Fig. 1. HasTEE$^+$ APIs for loading data and computations on the TEE and invoking the TEE (parameterized types simplified and typeclass constraints omitted for brevity).

equivalent `runEnclave`, as in our programming model *a client functions as the main application driver, while the enclave serves as a computational service.*

The client and server communicate with each other through a user-defined `API` type (line 8) that encapsulates a remote closure, represented using the

`Secure` type constructor. This closure is constructed on line 21 with the `inEnclave` function whose type signature can be found in Fig. 1. The parameter `LIOState l p` and the typeclass constraint `Label l` are explained in Section 3.3.

The `client` function (lines 10-15) has access to the remote closure through the `API` type. The remote function is invoked on line 14 using the `<@>` operator to emulate function application and the `gatewayRA` function executes the remote function call. The respective type signatures are specified in Fig. 1.

A notable type in the `pwdChecker` function is `DCLabeled String` that captures the password string but is labeled with ownership information of user *Alice*. The labeling happens on line 19 using the `inEnclaveLabeledConstant` function and the label `pwdLabel` (lines 24,25). The body of `pwdChecker` uses certain IFC operations - `getPrivilege` and `unlabelP`, which we elaborate on in Section 3.3.

The structure of Listing 1 represents the standard style of writing HasTEE$^+$ programs, where the monadic types `EnclaveDC`, `Client` and `App` indicate the partitioning within the program body. We discuss the partitioning tactic and the underlying representation to capture multiple clients in the next section.

3.1 Tierless client-server programming

HasTEE$^+$ builds on the partitioning strategy of HasTEE [37] but generalizes it for multiple clients. The strategy involves compiling the program `n` times for `n` parties. The compilation of the enclave program substitutes dummy implementation for all client monads. Similarly, each client compilation substitutes a dummy value for the enclave monad. The distinction between each client is done using a type-level string identifier, such as `"client1"`. At runtime, this identifier is used to dynamically dispatch the correct client, as shown in Fig. 2.

The dynamic identifier-based dispatch of the client computation is inspired from the HasChor library [40] for choreographic programming. In this approach, a `Client` monad is parameterized with a type-level location string, which is used at runtime to execute the desired `Client` computation (see Listing 2).

```haskell
data Client (loc :: Symbol) a where
  Client :: (KnownSymbol loc) => Proxy loc -> IO a -> Client loc a

symbolVal :: forall (n :: Symbol) proxy. KnownSymbol n => proxy n -> String

runClient :: Client loc a -> App Done
runClient (Client loc cl) = App $ do
  location <- get -- the underlying App monad captures the location
  if ((symbolVal loc) == location)
  then liftIO cl >> return Done
  else return Done -- cl not executed
```

Listing 2: The underlying Client monad in HasTEE$^+$

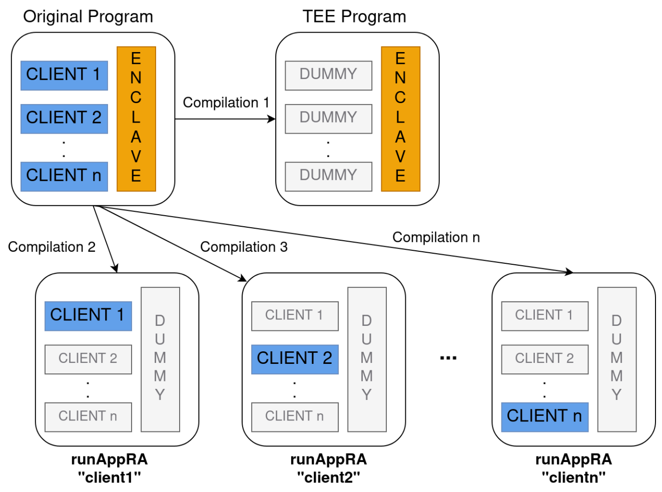

Fig. 2. HasTEE$^+$'s partitioning uses multiple compilations to create binaries that can dynamically dispatch the code for only one concerned monad based on a string identifier

The function `symbolVal` is provided by GHC to reflect types as terms at runtime, provided the types are constrained by the `KnownSymbol` typeclass. The `runClient` function in Listing 2 queries the App monad that captures the string within the underlying `App` monad. The `runClient` implementation for the `EnclaveDC` module is simply implemented as `runClient _ = return Done`, which amounts to a dummy implementation. A case study involving multiple clients is demonstrated in Section 4.

For the remote function invocation, the `inEnclave` function internally builds a *dispatch table* mapping an integer identifier to each enclave function. The client only gets access to the integer identifier. It uses the `<@>` operator to gather the function argument and the `gatewayRA` function to serialise the arguments, make the remote function call (specifying the identifier), and obtain the result computed on the remote enclave machine. Note the `Binary` typeclass constraint (Fig. 1) on both of the remote invocation functions for binary serialisation.

The complete implementation details of HasTEE$^+$ has been made publicly available[1]. Further details on the operational semantics of the general partitioning strategy can be found in the HasTEE[37] paper.

3.2 Remote Attestation via a Monitoring Server

A key component of confidential computing is remote attestation, which establishes *trust* on a TEE within a malicious environment. In HasTEE$^+$, we conduct our experiments on Intel SGX enclaves, and hence our infrastructure is integrated with the SGX attestation protocol. The low-level protocol is a multi-step

[1] https://github.com/Abhiroop/HasTEE

process [21] that begins with the client sending a nonce to the TEE, the TEE then creates a manifest file that includes an ephemeral key to encrypt future communication. Next, the TEE generates an *attestation report* that summarizes the enclave and platform state. A quoting enclave on the same machine verifies and signs the report, now called a *quote*, and returns it to the client. The client then communicates with the Intel Attestation Service (IAS) to verify the quote.

The API for this interface is quite low-level and involves programming at the level of a device driver (`/dev/attestation`). Intel's RA-TLS protocol abstracts over the low-level APIs and presents an API deeply tied to the TLS protocol. RA-TLS operates by extending an X.509 certificate to incorporate the attestation report within an unused X.509 extension field. During TLS connection setup, it uses the TLS handshake to transmit the *quote*, calculated using the protocol described earlier [21]. The enclave programmer, working with RA-TLS, interacts with a modified TLS implementation such as Mbed TLS [2]. Now, the focus shifts back to dealing with low-level socket-programming APIs, such as:

```
int (*ra_tls_create_key_and_crt_der_f)(uint8_t** der_key, size_t* der_key_size,
                                       uint8_t** der_crt, size_t* der_crt_size);
void (*ra_tls_set_measurement_callback_f)(int (*f_cb)(const char* mrenclave,
           const char* mrsigner, const char* isv_prod_id, const char* isv_svn));
```

Once again, managing these APIs is *error-prone* and *memory unsafe*. Additionally, it requires constructing *underspecified protocols*. Most importantly, the programmer is burdened with handling *cross-cutting concerns* that are irrelevant to the application code.

In HasTEE$^+$, we abstract over Intel's RA-TLS protocol. As mentioned earlier, clients always serve as the primary program *driver*, while the enclave functions as a computational *service*. The *enclave-as-a-service* model is implemented by representing the entire enclave program as an infinitely running server. The server is implemented in C using the Mbed TLS library [2], which can parse and verify the modified X.509 certificate. Internally, when the enclave runs, it spawns the C server hosted on the enclave memory but using separate memory pages. We use GHC's Foreign Function Interface to establish a communication channel between the C and Haskell heaps. Listing 3 shows a high-level overview of the implementation.

The Mbed TLS-based C server module acts as a *monitor* for the enclave application. All dataflows between the clients and the enclave pass through this module, which conducts integrity checks on incoming data at this point. Fig. 3 shows the HasTEE$^+$ general monitoring architecture.

There are two distinct attackers targeting the dataflow - (1) a malicious OS snooping or tampering with the data flowing into the enclave, and (2) a malicious client, potentially colluding with the OS, repeatedly sending garbage inputs to observe the behaviour of the enclave. The TLS channel specifically prevents the first attack. For the second attack, we use public-key-cryptography-based digital signatures to verify the identity of the client making the request.

For instance, in Listing 1, we provision the public key for user *Alice* during enclave boot time and disallow password checks from other malicious clients.

```
runAppRA :: Identifier -> App a -> IO a
runAppRA ident (App s) = do
  (a, vTable) <- runStateT s (initAppState ident)
  flagptr  <- malloc :: IO (Ptr CInt)
  dataptr  <- mallocBytes dataPacketSize :: IO (Ptr CChar)
  _    <- forkOS (startmbedTLSSERVER_ffi tid flagptr dataptr)
  result <- try (loop vTable flagptr dataptr) -- exception handler
  -- exception handling and freeing C pointers
  return a
  where
    loop :: [(CallID, Method)] -> Ptr CInt -> Ptr CChar -> IO ()
    loop vTable flagptr dataptr = do -- body elided
      -- non-blocking loop that gets woken when data arrives;
      -- `flagptr` indicates data arrival; read data from `dataptr`
      -- invoke the correct method from the lookup table `vTable`
      loop vTable flagptr dataptr -- continue the event loop
-- implemented in C
startmbedTLSSERVER_ffi :: ThreadId -> Ptr CInt -> Ptr CChar -> IO ()
```

Listing 3: High-level template of the `runAppRA` function for the enclave

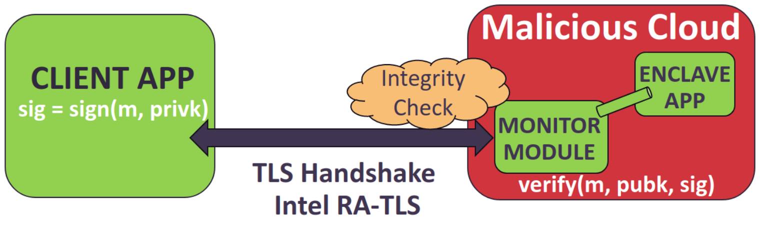

Fig. 3. HasTEE+'s remote attestation infrastructure abstracts over Intel's RA-TLS protocol and supports establishing the identity of the client and the server

While Intel-SGX also offers mutual attestation services, they depend on the client machine supporting an SGX enclave. Considering Intel's recent deprecation of SGX services on desktops and other client devices [47], our scheme aligns well.

3.3 Dynamic Information Flow Control

While programming TEEs using a memory-safe and type-safe language inherently provides stronger guarantees than programming in C/C++, such TEE applications remain vulnerable to unintended information leaks. To mitigate such explicit and implicit information leaks, HasTEE$^+$ integrates a dynamic information flow control mechanism within the underlying `EnclaveDC` monad. For instance, the trusted function `pwdChecker` from Listing 1 operates in this monad and captures a *labeled* string - the user password. The internal representation of the types `EnclaveDC a` and `DCLabeled a` is as follows:

```
newtype Label l => Enclave l p a = Enclave (IORef (LIOState l p) -> IO a)
```

```
data Labeled l t where
  LabeledTCB :: (Label l, Binary l, Binary t) => l -> t -> Labeled l t

type EnclaveDC = Enclave DCLabel DCPriv
type DCLabeled = Labeled DCLabel

data LIOState l p = LIOState { lioCurLabel :: l, lioClearance :: l
                             , lioOutLabel :: l, lioPrivilege :: Priv p}
```

The above representation is inspired by the LIO Haskell library [43]. The `Enclave` monad is parameterized by the label type `l` and privilege type `p`, wherein the `Label` typeclass captures a lattice [9] with partial order $\sqsubseteq$ governing the allowed flows. The `Enclave` monad employs a *floating label* approach, inspired by the HiStar OS [51]. In this approach, the *computational context* retains a current label L_{cur}. Upon reading sensitive data labeled L, it *taints* the context with the label $L_{cur} \sqcup L$, where $\sqcup$ denotes the least upper bound. The floating label approach restricts subsequent effects and enforces the lattice property, thereby preventing information leaks from higher-classified data to lower contexts in the lattice—a principle known as non-interference [14]. In the `LIOState` type, `lioCurLabel` represents L_{cur}, and `lioClearance` imposes an upper bound on the upward flow of a computational context within the lattice.

The `EnclaveDC` type specialises the `Enclave` monad to use *disjunction category (DC)* labels [42]. A DC Label captures both the *confidentiality* [9] and *integrity* [6] as a tuple. It employs the notion of mutually distrusting *principals*, whose conjunction represents restrictions on both the confidentiality and integrity of the data. An example label type is found in Listing 1, where `pwdLabel` constructs a DC Label using the tuple-construction operator `%%` and a string representation of the principal *Alice* to give `"Alice" %% "Alice"` (line 25).

A notable component in the DC Label system is the notion of a *privilege*, which is the type parameter `p` in the `Enclave` monad. In most real-world scenarios, the strict enforcement of non-interference is impractical, and privileges allow relaxing this policy by defining a more flexible ordering relation, $\sqsubseteq_P$. In HasTEE$^+$'s DCLabel implementation, we use a *conjunctive normal form (CNF)* representation for each disjunctive category of confidentiality and integrity. Hence, given a boolean formula P representing the privileges and two labels $< C_1, I_1 >$ and $< C_2, I_2 >$, the $\sqsubseteq_P$ is defined as shown in the formula.

$$\frac{P \wedge C_2 \Longrightarrow C_1 \qquad P \wedge I_1 \Longrightarrow I_2}{< C_1, I_1 > \sqsubseteq_P < C_2, I_2 >}$$

In Listing 1, we use the `toCNF` function (line 20) to generate a privilege for *Alice* to declassify the password, provisioning it to the `Enclave` monad at boot time (line 21). This allows the `pwdChecker` function to invoke `getPrivilege` (line 4) and then use that privilege to call the `unLabelP` function (line 5), which internally computes the $\sqsubseteq_P$ formula shown above. The `unlabelP` function and a related set of core APIs for HasTEE$^+$'s IFC enforcement is shown in Fig. 4.

```
unlabel  :: Label l => Labeled l a -> Enclave l p a
unlabelP :: PrivDesc l p => Priv p -> Labeled l a -> Enclave l p a
label    :: (Label l, Binary l, Binary a)
         => l -> a -> Enclave l p (Labeled l a)
labelP   :: (PrivDesc l p, Binary l, Binary a)
         => Priv p -> l -> a -> Enclave l p (Labeled l a)
taint    :: Label l => l -> Enclave l p ()
taintP   :: PrivDesc l p => Priv p -> l -> Enclave l p ()
```

Fig. 4. Core HasTEE$^+$ APIs for Information Flow Control

Implementation note: for prototyping, we represent principals and corresponding privileges using `String`s. In practice, a 512-bit private-key-hash is recommended.

The operations shown above dynamically compute the $\sqsubseteq$ and $\sqsubseteq_P$ relation to determine allowed information flows. The `PrivDesc` typeclass permits delegating privileges akin to the *acts for* relation described in the Myers-Liskov labeling model [29]. The type `Labeled l a` exists to allow labeling values to labels other than L_{cur}. Labeled data can be used to indicate data ownership and hence we provide additional APIs for the `Client` monad to `label`, serialise and `unlabel` data, inspired by *labeled communication* in the COWL system [44].

An implementation challenge arises when integrating the dynamic IFC mechanism with our partitioning tactic, specifically within the `inEnclave :: Label l => LIOState l p -> a -> App (Secure a)` function. This function is used to mark a function as trusted and move it into the TEE. The polymorphic type `a` encodes any general function of the form `a`$_1$ `->` `a`$_2$ `-> ... a`$_n$ `-> Enclave l b`. However, the type-checker is unable to unify the `l` in `LIOState l p` and the `l` in `Enclave l b`. Due to the dynamic nature of our IFC mechanism, a user can mistakenly supply a different label type at runtime, preventing the type-checker from producing a witness. Accordingly, we use the `Data.Dynamic` module of GHC to *dynamically type* the `LIOState l p` term. Thus, before evaluating the LIO computation, our evaluator dynamically checks for matching types and, on success, executes the monadic computation. A notable aspect is that both the program partitioning and IFC enforcement are *implemented as a Haskell library*, which allows us to use these features of the language.

4 Case Study: A Confidential Data-Analytics Pattern

We present a case study in privacy-preserving data analytics, illustrating how a group of mutually distrusting parties can perform analytics without revealing their data to each other. We use the core features of HasTEE$^+$ that we have discussed so far - tierless programming, remote attestation and dynamic IFC with privileges for declassification.

Fig. 5 shows the overall confidential analytics setup. The analytics is carried out in a *data clean room (DCR)* - a TEE hosted on a public cloud that aggregates data from multiple parties without revealing the actual data. In Fig. 5, the notation $\{a, b, ...\}$ denotes the state (in-memory and persistent) of that party. The figure shows two distinct sets of participants - the *Data Providers (P)* and the *Analytics Consumers (C)*. The setup does not limit the number of participants, allowing m data providers and n analytics consumers, where $m, n \in \mathbb{N}$. Additionally, there are *no restrictions requiring the data providers and the analytics consumer to be different*, and hence in some cases $P = C$.

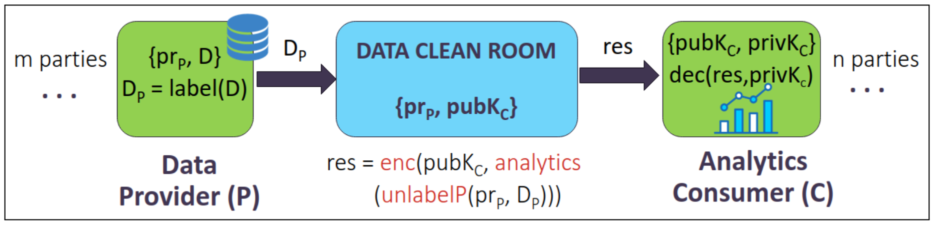

Fig. 5. A *Data Clean Room (DCR)* pattern with m data providers (P) and n analytics consumers (C). P labels its data as D_P and sends it to the DCR, which loads C's public key $pubK_C$ as well as *privilege* pr_P using a closure. The functions *enc* and *dec* handle encryption and decryption, and *analytics* refers to any general query.

In this setup, the data providers *label* their data before sending it to the DCR. The DCR is loaded with analytics queries from C after P reviews both the schema and the specific queries requested by C. The DCR is provisioned with C's public key, limiting access to the computed analytics solely to C. We notably use Haskell's partial application to load the privilege pr_P, enabling data unlabeling while restricting the locations of the `unlabelP` operation.

Consider a synthetic data analytics example with two data providers P_1 and P_2 that both store confidential data regarding COVID strains and the corresponding age of patients. An analytics consumer C_1 wishes to aggregate this data and derive the correlation between mean age and the respective COVID strain. We add a constraint that the analytics should only aggregate COVID strains that are common to both P_1 and P_2 (*private set intersection* [11]).

The DCR exposes two API calls for communication. The first, `datasend`, accepts a `DCLabeled Row` as an argument and is used by P_1 and P_2 to send a row labeled with their respective DC Label. The DCR stores the in-memory data in HasTEE$^+$'s mutable reference type `DCRef a`. Reading and writing occur via `readRef` and `writeRef`, raising the context's label accordingly. We omit the body of `datasend` for brevity. The full Haskell program is publicly available [36].

Listing 4 (lines 1-7) shows the second interface to the DCR, `runQuery`, used by C_1 to run the analytics query. A notable operation happens in line 5 where the `unLabelFunc` function is applied to each row of the database, labeled with either P_1's or P_2's DC label. `unlabelFunc` inspects the label and accordingly uses the correct privilege to declassify the data. Note, if during the whole computation, a row's label gets tainted by both P_1 and P_2, the `unlabel` function (not `unlabelP`,

```
1  runQuery :: EnclaveDC (DCRef DB) -> PublicKey
2           -> Priv CNF -> Priv CNF -> EnclaveDC ResultEncrypted
3  runQuery enc_ref_db pubKC1 priv1 priv2  = do
4    labeled_rows <- join $ readRef <$> enc_ref_db
5    rows         <- mapM (unlabelFunc priv1 priv2) labeled_rows
6    res_enc      <- encrypt_TCB pubKC1 (toStrict $ encode $ query rows)
7    return res_enc -- encryption error-handling elided
8
9  unlabelFunc p1 p2 lrow =
10   case extractOrgName (labelOf lrow) of
11     "P1" -> unlabelP p1 lrow
12     "P2" -> unlabelP p2 lrow
13     _      -> unlabel lrow -- label will float up
14
15 data API = API { datasend :: Secure (DCLabeled Row -> EnclaveDC ())
16                , runQ     :: Secure (EnclaveDC ResultEncrypted }
17
18 app = do db     <- liftNewRef dcPublic database
19          sfunc  <- inEnclave dcDefaultState $ sendData db
20          pubKC1 <- liftIO $ read <$> readFile "ssl/public.key"
21          p1Priv <- liftIO $ privInit (toCNF p1)
22          p2Priv <- liftIO $ privInit (toCNF p2)
23          qfunc  <- inEnclave dcDefaultState $ runQuery db pubKC1 p1Priv p2Priv
24          let api = API sfunc qfunc
25          runClient (client1 api)
26          runClient (client2 api)
27          runClient (client3 api)
```

Listing 4: `runQuery` unlabels the data, runs the query and encrypts the result; The `app :: App Done` computation captures the three clients and the enclave

see line 13) is invoked, floating the context high enough that writes to public channels are no longer possible.

In the absence of privileges, the `EnclaveDC` monad obeys general non-interference [14]. Hence, privileges, which allow *declassification* and *endorsement*, must be handed out with caution and used in limited places. In `app`, the privileges are created (lines 21, 22) and are partially applied to the `runQuery` function (line 23). As a result, the enclave loads a partially applied closure, `runQuery db pubKC1 p1Priv p2Priv`, and it is limited to using the privileges solely within `runQuery` and its *callees*. An interesting future work would be using Haskell's linear type support [5] to limit the copying of privileges and make them *unforgeable*.

The `app` function demonstrates the overall *tierless* nature of our DSL. It describes three clients and the enclave as a single program without specifying complex data copying protocols or involving multi-project hierarchies. We elide the body of the clients P_1 and P_2, involving data retrieval from their databases, labeling, and sending it to DCR, while C_1 calls `runQuery` and decrypts the result. We also omit the `query` function's implementation, responsible for executing private set intersection and returning results in a structured format. The interested reader can find the entire program hosted publicly [36]. In-transit security, enclave-integrity and client-integrity checks are implicitly enforced on all communication through HasTEE^{+}'s remote attestation infrastructure.

4.1 Security Analysis

Privacy Protection. The data clean room ensures privacy through - (1) *run-time security*, provided by the TEE's isolation of trusted code and data; (2) *in-transit security*, ensured by the RA-TLS protocol; (3) *enclave integrity*, established through remote attestation; (4) *client integrity*, provided with digital signatures checked by a monitor (Section 3.2); and (5) *information flow control*, implemented using a mix of IFC mechanisms and controlled privilege delegation.

Why is the result encrypted if HasTEE^{+}'s monitor already does client-integrity checks? This is necessary due to two distinct attacker models: *open-world attacks* and *closed-world attacks*. The digital signature verification in the monitor protects against open-world attacks, where an unknown malicious attacker outside our described system attempts to communicate with the DCR. On the other hand, in a closed-world attack, one of the participating entities, say P_1, may maliciously query the DCR for analytics, even though P_1 is intended to be merely a data provider. Although the monitor will allow this communication, the encryption will protect the data privacy.

Declassification Dimensions [35] Classifying the DCR along the four dimensions - **Who**: Integrity checks in HasTEE^{+}, RA-TLS and data labeling constrain the *who* dimension, allowing the DCR alone to declassify the data; **What**: The combination of the Haskell type system and dynamic privileges aim to allow declassification only for the analytics query; **Where**: The partial-application-based privilege loading was done to restrict this dimension, ensuring that only `runQuery` and its subsequent *callees* can declassify; **When** This is currently not captured but it is fairly straightforward to implement a *relative declassification* [35] policy where the analytics is released only after a certain set of data uploads succeeds, especially using the tierless nature of our DSL.

While HasTEE^{+}'s privacy protection mechanism provides useful guardrails against information leaks, we emphasize the importance of *auditing* the trusted code by all concerned parties to ensure the privileges are not misused.

5 Performance Evaluations

Here, we present performance microbenchmarks that allow for quantifying the overheads associated with various features in HasTEE^{+}. For benchmarking, we use the password checker example from Listing 1. We evaluate three particular sources of overheads - (1) *dynamic checks for information flow control*, (2) *remote attestation* and (3) *client-integrity checks* performed by the monitoring module.

We plot the overheads associated with each feature in Fig. 6. The X-axis captures the mean response-time in executing variants of the operation `gatewayRA`

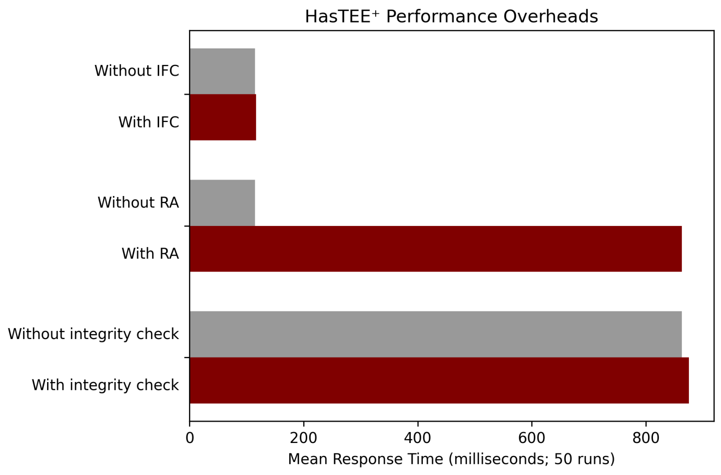

Fig. 6. Performance Overheads in HasTEE$^+$. The X-axis represents the mean response time for a query, both with and without the desired feature enabled, measured in milliseconds. The response time is averaged over 50 runs for each measure.

`((checkpwd api) <@> userInput)` (line 14) from Listing 1. A *gateway* call involves serialising the function arguments, making a remote procedure call to the enclave, executing the enclave computation, and then deserialising the result to the client. Measurement were conducted on an Azure Standard DC1s v2 (1 vcpu, 4 GiB memory) SGX machine, using the Intel SGX SDK for Linux.

Dynamic IFC Checks Overhead. In Fig. 6, the first group measures the overhead due to the runtime checks for IFC. The difference arises from extra conditional statements checking legal data flow policies. For Listing 1, the overhead is 2 milliseconds when compared to the non-IFC version. However, more complex applications (like Section 4) might incur slightly higher overheads.

Remote Attestation (RA) Overhead. The second group shows the RA overhead, demonstrating a considerable jump in response time with RA enabled. Much of the latency is attributed to the underlying *RA-TLS* protocol, implementing TLS version 1.2. In contrast, the non-RA baseline employs plain TCP for communication, using Linux's `send/recv` to enhance communication speed. The RA version's mean latency is 863 milliseconds, improvable by establishing a secure channel instead of initiating the entire handshake protocol each time.

Integrity-check Overhead. The client-integrity check is built on top of the HasTEE$^+$ RA infrastructure. As a result, for the baseline we use RA measurements from the second group and incorporate integrity checks on top of RA. The overhead on top of RA is minimal, in the order of 15 milliseconds.

Discussion. The measurements in Fig 6 show that each HasTEE$^+$ feature incurs maximum overheads in the order of hundreds of milliseconds. The significant response time increase for RA is mainly due to the complex TLS handshake involving multiple hops and communication with the Intel Attestation Service.

Given the security-critical nature of confidential computing and considering slowdowns due to general network latency, we posit that *HasTEE$^+$'s overheads are acceptable, making it a practical choice for security-critical applications.*

6 Related Work

We have already discussed projects closely related to HasTEE$^+$, including HasTEE [37], GoTEE [13], J_E [31], etc., in Sections 1 and 2. Here, we highlight additional related work that is relevant to the broader contributions of HasTEE$^+$.

Tierless Programming for Enclaves To the best of our knowledge, HasTEE$^+$ is one of the first practical programming frameworks to introduce the notion of *tierless* programming for confidential computing applications. Weisenburger et al. [49] provide a survey of general multi-tier programming approaches. Among the surveyed approaches, the HasTEE$^+$ DSL draws inspiration from the *Haste framework* [10] and functional choreographic programming [40].

Remote Attestation Infrastructure. HasTEE$^+$ is built on top of the Intel RA-TLS protocol [21] for *binary attestation.* In contrast, GuaranTEE [27] proposes a *control-flow attestation* technique based on two enclaves, which can be adapted quite naturally to the HasTEE$^+$ RA infrastructure.

Information Flow Control for TEEs. Gollamudi et al. proposed the first use of IFC to protect against low-level attackers in TEEs with the IMP$_E$ calculus [16], followed by a more general security calculus, DFLATE [17], for distributed TEE applications. In contrast to their work, HasTEE$^+$ does not require language-level modifications or type-system extensions. Instead, it conveniently enforces *IFC as a library* in an existing programming language. At the OS level, Deluminator [46] offers OS abstractions and userspace APIs for trace-based tracking of IFC violations in compartmentalized hardware, such as TEEs. Note that Deluminator is a reporting tool and not an *enforcement* mechanism, in contrast to HasTEE$^+$. Another application of IFC to TEEs is Moat [41], which formally verifies the confidentiality of enclave programs by proving the non-interference property [14].

Confidential Data Analytics. Referring to our confidential analytics pattern in Section 4, another proposed design pattern is Privacy Preserving Federated Learning [26], tailored to machine learning attacker models. We believe such threat models can be naturally integrated with our proposed design pattern.

7 Conclusion

We introduced HasTEE$^+$, a *tierless* confidential computing DSL that enforces dynamic information flow control, along with strong client-integrity and enclave-integrity checks. We also proposed a general confidential analytics pattern, expressed as a single program in HasTEE$^+$. Additionally, we presented performance evaluations that demonstrate acceptable overheads. Our evaluations, while conducted on Intel SGX, illustrate HasTEE$^+$'s general applicability to ARM TrustZone, AMD SEV, and Intel TDX machines. Furthermore, our library-based partitioning and IFC approach is extendable to other programming languages.

References

1. ARM: ARM TrustZone (2004), https://www.arm.com/technologies/trustzone-for-cortex-a
2. ARM: Mbed TLS (2009), https://tls.mbed.org
3. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D.M., Kapitza, R., Pietzuch, P.R., Fetzer, C.: SCONE: secure linux containers with intel SGX. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 689–703. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
4. Baumann, A., Peinado, M., Hunt, G.C.: Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. **33**(3), 8:1–8:26 (2015). https://doi.org/10.1145/2799647, https://doi.org/10.1145/2799647
5. Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093, https://doi.org/10.1145/3158093
6. Biba, K.J.: Integrity considerations for secure computer systems. Tech. rep., MITRE Corp. (04 1977)
7. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.: Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. IEEE Secur. Priv. **18**(3), 28–37 (2020). https://doi.org/10.1109/MSEC.2019.2963021, https://doi.org/10.1109/MSEC.2019.2963021
8. Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00. vol. 2, pp. 119–129. IEEE (2000)
9. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056, https://doi.org/10.1145/360051.360056
10. Ekblad, A., Claessen, K.: A seamless, client-centric programming model for type safe web applications. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. pp. 79–89. ACM (2014). https://doi.org/10.1145/2633357.2633367, https://doi.org/10.1145/2633357.2633367
11. Escalera, D.M., Agudo, I., López, J.: Private set intersection: A systematic literature review. Comput. Sci. Rev. **49**, 100567 (2023). https://doi.org/10.1016/J.COSREV.2023.100567, https://doi.org/10.1016/j.cosrev.2023.100567
12. Geppert, T., Deml, S., Sturzenegger, D., Ebert, N.: Trusted execution environments: Applications and organizational challenges. Frontiers Comput. Sci. **4** (2022). https://doi.org/10.3389/FCOMP.2022.930741, https://doi.org/10.3389/fcomp.2022.930741
13. Ghosn, A., Larus, J.R., Bugnion, E.: Secured routines: Language-based construction of trusted execution environments. In: Malkhi, D., Tsafrir, D. (eds.) 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. pp. 571–586. USENIX Association (2019), https://www.usenix.org/conference/atc19/presentation/ghosn

14. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014, https://doi.org/10.1109/SP.1982.10014
15. Goldman, K., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. In: Proceedings of the first ACM workshop on Scalable trusted computing. pp. 21–24 (2006)
16. Gollamudi, A., Chong, S.: Automatic enforcement of expressive security policies using enclaves. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 494–513. ACM (2016). https://doi.org/10.1145/2983990.2984002, https://doi.org/10.1145/2983990.2984002
17. Gollamudi, A., Chong, S., Arden, O.: Information flow control for distributed trusted execution environments. In: 32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019. pp. 304–318 (2019). https://doi.org/10.1109/CSF.2019.00028, https://doi.org/10.1109/CSF.2019.00028
18. Intel: Intel SGX Intro: Passing Data Between App and Enclave (2016), https://www.intel.com/content/www/us/en/developer/articles/technical/sgx-intro-passing-data-between-app-and-enclave.html
19. Intel: tlibc - an alternative to glibc (2018), https://github.com/intel/linux-sgx/tree/master/common/inc/tlibc
20. Intel: Intel Trust Domain Extensions (2021), https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
21. Knauth, T., Steiner, M., Chakrabarti, S., Lei, L., Xing, C., Vij, M.: Integrating remote attestation with transport layer security. arXiv preprint arXiv:1801.05863 (2018)
22. LinuxSGX: Linux SGX Remote Attestation (2017), https://github.com/svartkanin/linux-sgx-remoteattestation/blob/master/Application/isv_enclave/isv_enclave.cpp#L152-L308
23. Marlow, S., Jones, S.L.P., Singh, S.: Runtime support for multicore haskell. In: Hutton, G., Tolmach, A.P. (eds.) Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009. pp. 65–78. ACM (2009). https://doi.org/10.1145/1596550.1596563, https://doi.org/10.1145/1596550.1596563
24. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Lee, R.B., Shi, W. (eds.) HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013. p. 10. ACM (2013). https://doi.org/10.1145/2487726.2488368, https://doi.org/10.1145/2487726.2488368
25. Microsoft: Windows cryptoapi spoofing vulnerability (2020), https://nvd.nist.gov/vuln/detail/CVE-2020-0601
26. Mo, F., Haddadi, H., Katevas, K., Marin, E., Perino, D., Kourtellis, N.: PPFL: privacy-preserving federated learning with trusted execution environments. In: Banerjee, S., Mottola, L., Zhou, X. (eds.) MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021. pp. 94–108.

ACM (2021). https://doi.org/10.1145/3458864.3466628, https://doi.org/10.1145/3458864.3466628

27. Morbitzer, M., Kopf, B., Zieris, P.: Guarantee: Introducing control-flow attestation for trusted execution environments. In: 16th IEEE International Conference on Cloud Computing, CLOUD 2023, Chicago, IL, USA, July 2-8, 2023. pp. 547–553. IEEE (2023). https://doi.org/10.1109/CLOUD60044.2023.00073, https://doi.org/10.1109/CLOUD60044.2023.00073
28. Mulligan, D.P., Petri, G., Spinale, N., Stockwell, G., Vincent, H.J.M.: 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. pp. 132–138. IEEE (2021). https://doi.org/10.1109/SEED51797.2021.00025, https://doi.org/10.1109/SEED51797.2021.00025
29. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000). https://doi.org/10.1145/363516.363526, https://doi.org/10.1145/363516.363526
30. Northwood, C.: The full stack developer: your essential guide to the everyday skills expected of a modern full stack web developer. Springer (2018)
31. Oak, A., Ahmadian, A.M., Balliu, M., Salvaneschi, G.: Language support for secure software development with enclaves. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00037, https://doi.org/10.1109/CSF51468.2021.00037
32. Ramsingh, A., Singer, J., Trinder, P.: Do fewer tiers mean fewer tears? eliminating web stack components to improve interoperability. CoRR **abs/2207.08019** (2022). https://doi.org/10.48550/ARXIV.2207.08019, https://doi.org/10.48550/arXiv.2207.08019
33. Russinovich, M., Costa, M., Fournet, C., Chisnall, D., Delignat-Lavaud, A., Clebsch, S., Vaswani, K., Bhatia, V.: Toward confidential cloud computing. Commun. ACM **64**(6), 54–61 (2021). https://doi.org/10.1145/3453930, https://doi.org/10.1145/3453930
34. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121, https://doi.org/10.1109/JSAC.2002.806121
35. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). https://doi.org/10.3233/JCS-2009-0352, https://doi.org/10.3233/JCS-2009-0352
36. Sarkar, A.: Confidential private set intersection with hastee (2023), https://github.com/Abhiroop/HasTEE/blob/lio-ifc/app/Main.hs
37. Sarkar, A., Krook, R., Russo, A., Claessen, K.: HasTEE: Programming Trusted Execution Environments with Haskell. In: McDonell, T.L., Vazou, N. (eds.) Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023. pp. 72–88. ACM (2023). https://doi.org/10.1145/3609026.3609731, https://doi.org/10.1145/3609026.3609731
38. Sev-Snp, A.: Strengthening vm isolation with integrity protection and more. White Paper, January **53**, 1450–1465 (2020)
39. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications

Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 552–561. ACM (2007), https://doi.org/10.1145/1315245.1315313
40. Shen, G., Kashiwa, S., Kuper, L.: Haschor: Functional choreographic programming for all (functional pearl). Proc. ACM Program. Lang. **7**(ICFP), 541–565 (2023). https://doi.org/10.1145/3607849, https://doi.org/10.1145/3607849
41. Sinha, R., Rajamani, S.K., Seshia, S.A., Vaswani, K.: Moat: Verifying confidentiality of enclave programs. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1169–1184. ACM (2015), https://doi.org/10.1145/2810103.2813608
42. Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Disjunction category labels. In: Laud, P. (ed.) Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7161, pp. 223–239. Springer (2011), https://doi.org/10.1007/978-3-642-29615-4_16
43. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. In: Claessen, K. (ed.) Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011. pp. 95–106. ACM (2011), https://doi.org/10.1145/2034675.2034688
44. Stefan, D., Yang, E.Z., Marchenko, P., Russo, A., Herman, D., Karp, B., Mazières, D.: Protecting users by confining javascript with COWL. In: Flinn, J., Levy, H. (eds.) 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014. pp. 131–146. USENIX Association (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan
45. Stumpf, F., Tafreschi, O., Röder, P., Eckert, C., et al.: A robust integrity reporting protocol for remote attestation. In: Proceedings of the Workshop on Advances in Trusted Computing (WATC). p. 65 (2006)
46. Tarkhani, Z., Madhavapeddy, A.: Information flow tracking for heterogeneous compartmentalized software. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023. pp. 564–579. ACM (2023), https://doi.org/10.1145/3607199.3607235
47. Vault, H.: Intel SGX deprecation review (2022), https://hardenedvault.net/blog/2022-01-15-sgx-deprecated/
48. Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., Lin, Z.: Towards memory safe enclave programming with rust-sgx. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2333–2350. ACM (2019), https://doi.org/10.1145/3319535.3354241
49. Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. ACM Comput. Surv. **53**(4), 81:1–81:35 (2021), https://doi.org/10.1145/3397495
50. Zegzhda, D.P., Usov, E.S., Nikol'skii, V.A., Pavlenko, E.: Use of intel SGX to ensure the confidentiality of data of cloud users. Autom. Control. Comput. Sci. **51**(8), 848–854 (2017), https://doi.org/10.3103/S0146411617080284
51. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in histar. In: Bershad, B.N., Mogul, J.C. (eds.) 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA. pp. 263–278. USENIX Association (2006), http://www.usenix.org/events/osdi06/tech/zeldovich.html