2019 IEEE 32nd Computer Security Foundations Symposium (CSF)

Information Flow Control for Distributed Trusted Execution Environments

Anitha Gollamudi
Harvard University

Stephen Chong
Harvard University

Owen Arden
University of California, Santa Cruz

***Abstract*—Distributed applications cannot assume that their security policies will be enforced on untrusted hosts. Trusted execution environments (TEEs) combined with cryptographic mechanisms enable execution of known code on an untrusted host and the exchange of confidential and authenticated messages with it. TEEs do not, however, establish the *trustworthiness* of code executing in a TEE. Thus, developing secure applications using TEEs requires specialized expertise and careful auditing.**

This paper presents DFLATE, a core security calculus for distributed applications with TEEs. DFLATE offers high-level abstractions that reflect both the guarantees and limitations of the underlying security mechanisms they are based on. The accuracy of these abstractions is exhibited by asymmetry between confidentiality and integrity in our formal results: DFLATE enforces a strong form of noninterference for confidentiality, but only a weak form for integrity. This reflects the asymmetry of the security guarantees of a TEE: a malicious host cannot access secrets in the TEE or modify its contents, but they can suppress or manipulate the sequence of its inputs and outputs. Therefore DFLATE cannot protect against the suppression of high-integrity messages, but when these messages are delivered, their contents cannot have been influenced by an attacker.

***Index Terms*—information flow control, language-based security, trusted execution environment, enclaves, distributed systems security**

I. Introduction

Many applications rely on security checks in compilers and runtime systems to enforce security policies. In distributed decentralized settings (where applications are distributed, entities involved in the application may be mutually distrusting, and no single node is trusted by all entities), the effectiveness of such checks is limited: local security checks cannot ensure that a remote host will protect confidential information it receives. Encryption can ensure that an untrusted host cannot reveal secrets, but it also prevents the host from performing general computation on encrypted data.[1] Lack of trust between entities may require data to be hosted separately from computations that use it, hurting performance. Worse, it is possible that no entity is sufficiently trusted to both access the data and compute the result, limiting what the application can do.

Trusted Execution Environments (TEEs) such as SGX [29, 4] and Sanctum [14] address some of these limitations with application *enclaves*. An enclave is a protected user-level process that is strongly isolated from the OS and other applications by trusted hardware. Remote nodes can verify the integrity of code running in an enclave using a *remote attestation protocol*. Once verified, the remote node knows that runtime security checks are still present in the code, and that static properties verified during compilation are still valid.

TEEs by themselves are insufficient to enforce security policies. For instance, inputs and outputs to TEEs must be correctly encrypted, signed, decrypted, and verified to protect against malicious hosts. Even with correct use of cryptography, the application must be written to ensure that it does not inappropriately reveal confidential information nor allow entities to inappropriately influence computations. Although previous work has combined techniques to enforce strong application-level confidentiality and integrity guarantees with correct-by-construction use of cryptography [28, 41, 39], no such previous work supports TEEs, and extending them to do so is nontrivial.

This work presents Distributed Flow-Limited Authorization for Trusted Execution (DFLATE), a core calculus for secure decentralized distributed applications. DFLATE extends the Flow-Limited Authorization Calculus (FLAC) [5] with distributed execution, communication channels, concurrency, and TEEs. DFLATE's type system enforces confidentiality and integrity guarantees that are consistent with standard cryptographic mechanisms and TEE platforms.

To better understand how TEEs work, and the challenges in building secure applications that use them, consider an example of a simple distributed application, illustrated in Figure 1. Here, "Enclave" refers to code running in a TEE on Bob's node. The only way for Alice to interact with the enclave is via Bob, whom Alice does not trust. To establish the authenticity of the enclave, Alice uses a *remote attestation protocol*. First, Alice requests a remote attestation from Bob (message 0), who requests a *secure measurement* of the enclave code from the TEE: a cryptographic hash of the loaded binary (message 1). This hash, as well as additional parameters for establishing a secure channel, is signed by a key that has been securely provisioned to the TEE (message 2). Next, Bob relays the signed message to Alice (message 3), who inspects the measurement to ensure the expected code is running, and verifies the signature to ensure it is from an authentic TEE.

Once the signature is verified, Alice uses the security parameters included in the message to establish a secure authenticated channel to the enclave. Alice uses this channel to provide decryption and signing keys to the enclave (messages 4 and 5). Later, she can use these keys to exchange encrypted and signed inputs and outputs with the enclave (messages 6-9) without repeating the remote attestation protocol.

Omitting or improperly executing any of the above steps can

[1]Fully homomorphic encryption [21] can permit such computation, but at great cost to performance.

DOI 10.1109/CSF.2019.00028

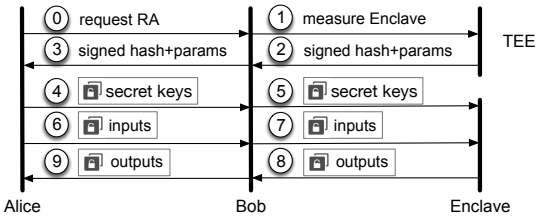


Fig. 1: Remote enclave attestation and secret provisioning.

undermine Alice's security. If the remote attestation is omitted, Alice has no guarantee that the enclave code (and not some malicious version of it) is running nor that code execution is protected from Bob. If Alice fails to encrypt (or sign) inputs to the enclave, or uses keys that are accessible to Bob, then Bob can learn (or modify) the inputs. Similarly, if the enclave fails to properly encrypt and sign outputs, Bob may be able to read or modify them.

Fortunately, the security and correctness of the first three messages is mostly independent of the application. So a relatively simple (but trustworthy) library API or language extension can provide remote attestation capabilities to applications and eliminate programmer errors.

But even with remote attestation and proper encryption and authentication, Alice's security may still be undermined. Although Bob cannot decrypt messages between Alice and the enclave, he does see each encrypted message when it is transmitted and may be able to infer secret information based on the sequence of exchanged messages. For example, the pseudocode below sends an encrypted and signed message `msg` from within an enclave over channel `ch` if `h` is true.

```
if h then send ch (enc (sign msg)) else ()
```

Because of the TEE and the cryptographic mechanisms, Bob cannot directly access `h` or `msg`, but he can infer the value of `h` based on whether a message is sent. The above code might also be problematic in terms of integrity: if Bob can influence the value of `h`, he can suppress the message. Similar code might permit Bob to replay messages or permute the message order.

Information-flow control (IFC) is well suited to protect against these kinds of vulnerabilities because it enables end-to-end semantic guarantees such as *noninterference*, which ensures an attacker cannot infer secret information from public outputs. However, existing IFC languages cannot precisely model the security guarantees and limitations of TEEs.

There are two key challenges to enforcing IFC in a decentralized distributed setting that employs encryption, signatures, and TEEs. The first challenge is to (symbolically) represent the security guarantees of the cryptographic mechanisms without abstracting away the power of the attacker to permute, suppress, or infer secrets from the message sequence. Security models of existing distributed IFC systems (Fabric [28] and DStar [41]) are insufficiently precise. Encryption and digital signatures allow secret or high-integrity messages to be sent over untrusted channels. For example, Alice could sign and send a message to the enclave over a channel controlled by Bob; if the enclave receives the message it knows (by verifying the signature) that it is from Alice, even though Bob could suppress the message. In Fabric and DStar, the only sound policy (that doesn't ignore a potential attack) expresses that both Alice and Bob might have influenced the message. In other words, they are too coarse-grained to distinguish the attacker's influence on control flow from its influence on data flow. Consequently, their enforcement mechanisms cannot determine if code respects the programmer's intended policy.

This scenario arises in any nontrivial application using TEEs, since the main benefit of TEEs is to run computation on potentially malicious nodes. So IFC must be able to reason about protected data flowing through untrusted nodes.

The second challenge is to design high-level abstractions that accurately reflect the guarantees of TEEs in a decentralized distributed setting. Currently, developers integrate TEEs into their applications using low-level library APIs. Using these libraries correctly may require a different skill set from that needed for the rest of the application. A better approach would be to design high-level programming abstractions for TEEs that don't burden the developer with low-level implementation details. Code expressed with these abstractions can be used to synthesize low-level implementations, shifting trust from application developers to the compiler.

Finding the right security abstraction for TEEs in decentralized settings is challenging. TEEs ensure that specific code is running securely, but, as discussed above, do not ensure the trustworthiness of the code. So different entities may trust different enclaves (perhaps based on who wrote the enclave code or analysis of the code). TEE mechanisms don't hide the existence of messages to and from an enclave, nor guarantee message delivery. A suitable TEE abstraction must reflect these limitations on communication and allow entities to express their trust in specific TEEs and entities.

DFLATE addresses these challenges. DFLATE has sufficiently fine-grained information-flow control to distinguish (and usefully reason about) important TEE use cases. DFLATE provides language abstractions for TEEs, distribution, and security principals that can ensure security while enabling applications to benefit from the powerful features of TEEs.

DFLATE is the first language to enforce end-to-end information security for distributed applications with TEEs. We prove that well-typed DFLATE programs enjoy *noninterference* guarantees. Confidentiality noninterference [22] ensures that an attacker cannot infer secret information from public outputs. Integrity noninterference ensures that an attacker cannot influence high-integrity outputs by modifying low-integrity inputs.

Integrity is dual to confidentiality [10], and thus most systems that protect confidentiality noninterference also protect integrity noninterference. However, DFLATE provides asymmetric guarantees for confidentiality and integrity. This asymmetry reflects inherent limitations of TEEs. The confidentiality and integrity of the *contents* of inputs and outputs to TEEs can be cryptographically protected, but neither the TEE itself nor cryptographic mechanisms can prevent the host of the TEE from suppressing or manipulating the sequence of inputs and outputs. Hence, we derive strong noninterference results for confidentiality, but weaker results for integrity that hold only

when messages are not suppressed.

II. Motivating the DFLATE design

DFLATE is a high-level language designed to be implemented using cryptographic mechanisms and trusted execution environments. Designing an IFC model in this setting is subtly different than designing a general IFC model. In this section we motivate three design features of DFLATE that are informed by cryptography and TEEs.

A. Fine-grained policies for secure communication

Suppose Alice sends a message to Carol via Bob, who is only partially trusted by Alice and Carol. Figure 2(a) illustrates the scenario where no cryptographic mechanisms are used to enforce information security, similar to the model of Fabric and DStar. Sending message A1 to Carol is secure only if Alice permits Bob to learn the contents of A1 and Carol permits Bob to (potentially) modify the contents of A1 en route. Figure 2(b) illustrates the same scenario, but Alice additionally signs the message and encrypts it with Carol's public key. In this case, Bob can neither learn nor modify the contents of A1. However, Bob does learn of the existence of A1. Furthermore, although Bob cannot modify A1, he could replace it with a previously signed message A2, or could choose to send no message at all.

Most existing IFC abstractions do not distinguish these two scenarios and instead enforce policies conservatively using checks similar to Figure 2(a). This lack of precision effectively ignores guarantees offered by cryptographic mechanisms for communication over untrusted channels.

DFLATE distinguishes the ability to disclose or modify messages sent over a channel from the ability to observe channel traffic and influence or suppress message sequences. In DFLATE, the security of a channel is specified using two policies. One policy governs the confidentiality and integrity of the contents of messages sent over the channel, and the other governs the confidentiality and integrity of contexts in which the channel may be used. A node may receive a message that it can't read or modify; this can be enforced by signing and encrypting the message. A node should *not* send a message to an untrusted node in a secret context (even if the message is public), and should not rely on a message from an untrusted node in a high-integrity context (even if the message contents are trusted).

B. Decentralized and distributed trust management

DFLATE's abstractions are based directly on the capabilities of the underlying cryptographic and TEE mechanisms, which allows stronger assumptions and finer-grained reasoning about what information flows and actions are possible than most previous IFC models. Two places where DFLATE's design is influenced by the underlying mechanisms are *clearance bounds* and *computation principals*.

DFLATE's type system associates a *clearance bound* [36] with every node, which restricts what data may be used and produced by computations on that node. Based on trust relationships between the node and other principals, the clearance bound reflects which cryptographic keys the node has access to, and thus models the ability of a node to digitally sign values and decrypt encrypted values.[2] In Figure 2(b), Bob does not have access to Alice's decryption key, so any computation that attempts to read and compute with Alice's data would exceed Bob's clearance. Similarly, Bob would be unable to produce a new message with Alice's integrity using a DFLATE program, modeling Bob's inability to access Alice's signing key.

For each source-level DFLATE computation e that will execute in a TEE, DFLATE defines a unique *computation principal* t. Code running in a TEE is subject to clearance bounds of the computation principal rather than of the node executing the TEE. DFLATE permits principals to express their trust in code running in a TEE by expressing trust in the corresponding computation principal t. Therefore Alice can express trust in an enclave running on Bob's node, allowing it to perform computation on her secrets even if Bob is not trusted to do so. DFLATE also provides protection in the other direction: if Bob doesn't trust Alice or the enclave, Alice can't use the enclave to leak Bob's secrets or influence his data.

C. Observability of TEE interactions

TEEs introduce additional subtlety into information flow control design. TEEs provide guarantees similar to those of a trusted third party, but executing code in a TEE on an untrusted node is not equivalent to executing code on a trusted node.

Consider our previous examples, but where Bob executes application code in an enclave E (Figure 2(c)). Although the code executes within an enclave, Bob can still observe and manipulate incoming and outgoing messages, as in Figure 2(b).

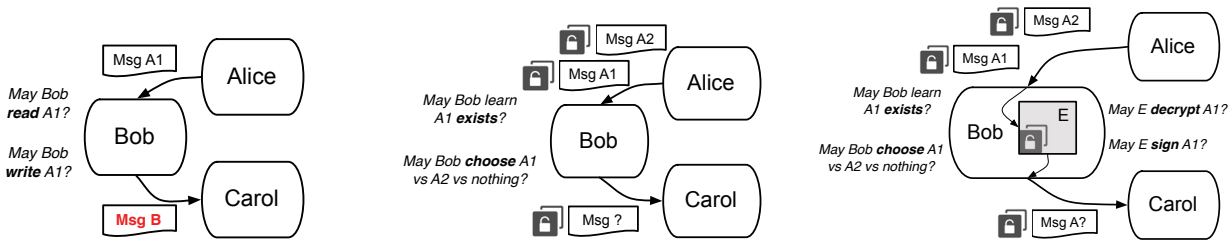
Most distributed IFC approaches (e.g., [28, 41]) ignore an attacker's ability to analyze traffic over communication channels. This is somewhat defensible for attackers with a limited view of the network, or when nodes use obfuscating techniques like TOR [15]. With TEEs, however, ignoring this ability is not as reasonable: in Figure 2(c), Bob is the *only* available conduit to E. Communicating over an observed untrusted channel is fundamental to the TEE abstraction. DFLATE ensures that programs capture the ability of a host to mediate communication with its enclaves, and enables reasoning about the security of these situations. For node-to-node communication, DFLATE makes similar assumptions to previous models: only the sender and the receiver observe the communication.

III. The DFLATE language

A. FLAM principal algebra

Security policies in DFLATE are based on the Flow-Limited Authorization Model (FLAM) [6], a principal algebra and logic for reasoning simultaneously about authorization and information flow control policies. Entities in a distributed application (e.g., Alice, Bob, etc.) are represented by names in a set of primitive principals $\mathcal{N}$. The FLAM algebra provides operations for constructing composite principals from this base set. A FLAM principal refers to the authority of the entity (or entities) represented by that principal. A principal p's authority consists of *confidentiality authority*, the authority necessary to learn

[2]Using LIO-style clearance bounds as a proxy for access to cryptographic keys was first introduced in CLIO [39].



(a) Security checks without cryptography. (b) Security checks with cryptography. (c) Security checks with an enclave on Bob.

Fig. 2: Information flow checks with and without cryptographic mechanisms.

p's secrets, and *integrity authority*, the authority necessary to influence information trusted by p. *Authority projections* of the form p^{π} where $\pi \in \{\rightarrow, \leftarrow\}$ allow us to represent the partial authority of a principal. For example, the principal $p^{\rightarrow}$ denotes a principal with only the confidentiality authority of p, and $p^{\leftarrow}$ denotes a principal with only the integrity authority of p.[3] The combined authority of principals p and q is represented by the conjunction $p \wedge q$, and the selective authority of principals p and q (i.e., the individual authority of either p or q) is represented by the disjunction $p \vee q$. The universally trusted principal (with the *most* authority) is represented by $\top$, and the universally distrusted principal (with the *least* authority) is $\bot$.

The complete set $\mathcal{P}$ of FLAM principals for any setting is given by the closure of the operations $\wedge$, $\vee$, $\leftarrow$, and $\rightarrow$ over the set of primitive principals $\mathcal{N}$, extended with $\top$, and $\bot$. Principals in this set are related by a preorder $\succcurlyeq$, the "acts for" relation, which orders principals by increasing trust. The equivalence classes[4] of $\succcurlyeq$ form a distributive lattice with $\top$ and $\bot$ as most and least trusted elements, and with $\wedge$ and $\vee$ as join and meet operations.

The trust ordering $\succcurlyeq$ also induces an ordering on principals specifying safe information flows. We write $p \sqsubseteq q$ when information labeled p may safely flow to principal q. The *flows-to* relation also forms a distributive lattice with $\bot^{\rightarrow} \wedge \top^{\leftarrow}$ (public and trusted) as the least restrictive element, and $\top^{\rightarrow} \wedge \bot^{\leftarrow}$ (secret and untrusted) as the most restrictive element. The flows-to relation and joins and meets in the information flow lattice are defined in terms of their authority lattice counterparts:

$$\begin{array}{lcl} p \sqsubseteq q & \triangleq & p^{\leftarrow} \succcurlyeq q^{\leftarrow} \text{ and } q^{\rightarrow} \succcurlyeq p^{\rightarrow} \\ p \sqcup q & \triangleq & (p^{\rightarrow} \wedge q^{\rightarrow}) \wedge (p^{\leftarrow} \vee q^{\leftarrow}) \\ p \sqcap q & \triangleq & (p^{\rightarrow} \vee q^{\rightarrow}) \wedge (p^{\leftarrow} \wedge q^{\leftarrow}) \end{array}$$

Every principal p is equivalent (under the trust ordering) to a principal in *normal form*, $q^{\rightarrow} \wedge r^{\leftarrow}$, i.e., the conjunction of a confidentiality authority and an integrity authority. The *voice* of a principal p, $\nabla(p)$, is the integrity authority necessary to act on the behalf of the principal. Formally, if $q^{\rightarrow} \wedge r^{\leftarrow}$ is the normal form of p, then $\nabla(p) = \nabla(q^{\rightarrow} \wedge r^{\leftarrow}) = q^{\leftarrow} \wedge r^{\leftarrow}$.

[3]One way to remember what each arrow means is to think of confidentiality as secrets "coming from" p, and integrity as information "accepted by" p.

[4]Principals p and q are in the same equivalence class if and only if $p \succcurlyeq q$ and $q \succcurlyeq p$.

$n \in \mathcal{N}$ (primitive principals) $\quad x \in \mathcal{V}$ (variables)
$ch \in \mathcal{V}_C$ (channel variables) $\quad \nu \in \mathcal{C}$ (channel values)

$$\begin{array}{rcl} p, \ell, \mathit{pc} & ::= & n \mid \top \mid \bot \mid p^{\rightarrow} \mid p^{\leftarrow} \mid p \wedge p \mid p \vee p \\ v & ::= & () \mid \langle v, v \rangle \mid \langle p \succcurlyeq p \rangle \mid \mathtt{inj}_i\ v \mid \lambda(x{:}\tau)[\mathit{pc}, \Theta, \mathcal{P}, \Pi].\ e \\ & \mid & \Lambda X[\mathit{pc}, \Theta, \mathcal{P}, \Pi].\ e \mid \overline{\eta}_{\ell}\ v \mid v\ \mathtt{where}\ v \\ e & ::= & x \mid v \mid e\ e \mid \langle e, e \rangle \mid \eta_{\ell}\ e \mid e\ \tau \mid \mathtt{proj}_i\ e \\ & \mid & \mathtt{inj}_i\ e \mid \mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\ e \mid \mathtt{inj}_2(x).\ e \\ & \mid & \mathtt{bind}\ x = e\ \mathtt{in}\ e \mid \mathtt{assume}\ e\ \mathtt{in}\ e \\ & \mid & \mathtt{send}\ ch\ e\ \mathtt{then}\ e \mid \mathtt{recv}\ ch\ \mathtt{as}\ x\ \mathtt{in}\ e \\ & \mid & \mathtt{TEE}^t\ s \mid \mathtt{spawn}\ @n\ (ch[\mathit{pc}; \tau],\ ch[\mathit{pc}; \tau]).\ e\ \mathtt{then}\ e \\ & \mid & \mathtt{runTEE}^t\ s \mid \overline{\mathtt{send}}\ ch\ v\ \mathtt{then}\ e \mid e\ \mathtt{where}\ e \\ \tau & ::= & p \succcurlyeq p \mid \mathtt{unit} \mid \tau + \tau \mid \tau \times \tau \\ & \mid & \tau \xrightarrow{\mathit{pc}, \Theta, \mathcal{P}, \Pi} \tau \mid \ell\ \mathtt{says}\ \tau \mid X \mid \forall X[\mathit{pc}, \Theta, \mathcal{P}, \Pi].\ \tau \\ c & ::= & \mathtt{chan}_{p \rightarrow q}\ \mathit{pc}\ \tau \mid \mathtt{chan}_{p \leftarrow q}\ \mathit{pc}\ \tau \end{array}$$

Fig. 3: DFLATE syntax

B. DFLATE syntax and local semantics

The DFLATE language is inspired by the Flow-Limited Authorization Calculus (FLAC) [5]. Like FLAC, DFLATE is a core calculus and secure programming model that enforces strong information security guarantees. DFLATE extends FLAC with distributed computation, communication, and TEEs, and the DFLATE type system is more compatible with implementations that use cryptographic enforcement mechanisms. This makes DFLATE a more suitable basis for the formal analysis of decentralized distributed applications, or as a core programming model for a general-purpose secure distributed programming language.

Figure 3 shows the DFLATE syntax. Principals are used both to specify the information flow policies on data and to represent the authority of the entities in the distributed application. Metavariables p, q, ℓ, and *pc* range over principals. We assume the set of primitive principals $\mathcal{N}$ includes computation principals t and nodes n, representing, respectively, code executed in a TEE and host machines. Nodes and computation principals represent the places where computation occurs, and we use metavariable *pl* to range over them.

Metavariables v and e range over values and expressions. (Shaded values and expressions are not part of the surface syntax but arise during evaluation.) DFLATE includes standard

[DE-APP] $\mathit{pl}, D \vdash (\lambda(x{:}\tau)[\mathit{pc}, \Theta, \mathcal{P}, \Pi].\, e)\ v \longrightarrow e\{v/x\}$

[DE-UNITM] $\mathit{pl}, D \vdash \eta_\ell\ v \longrightarrow \overline{\eta}_\ell\ v$

[DE-BINDM] $\mathit{pl}, D \vdash \texttt{bind}\ x = \overline{\eta}_\ell\ v\ \texttt{in}\ e \longrightarrow e\{v/x\}$

[DE-ASSUME] $\mathit{pl}, D \vdash \texttt{assume}\ v\ \texttt{in}\ e \longrightarrow e\ \texttt{where}\ v$

[DE-TEE] $\mathit{pl}, D \vdash \texttt{TEE}^t\ s \longrightarrow \texttt{runTEE}^t\ s$

[DE-WHERE] $$\frac{\mathit{pl}, D \cdot v \vdash e \longrightarrow e'}{\mathit{pl}, D \vdash e\ \texttt{where}\ v \longrightarrow e'\ \texttt{where}\ v}$$

[DE-SEND] $$\frac{v' = \mathsf{export\_del}(D, v) \qquad \mathsf{noTEE}(v)}{\mathit{pl}, D \vdash \texttt{send}\ \nu\ v\ \texttt{then}\ e \longrightarrow \overline{\texttt{send}}\ \nu\ v'\ \texttt{then}\ e}$$

export_del is a function such that if $D = \langle p_1 \succcurlyeq q_1\rangle \cdot \ldots \cdot \langle p_n \succcurlyeq q_n\rangle$ then $\mathsf{export\_del}(D, e) = e\ \texttt{where}\langle p_1 \succcurlyeq q_1\rangle \ldots \texttt{where}\langle p_n \succcurlyeq q_n\rangle$.

Fig. 4: Selected DFLATE sequential evaluation rules

syntax for variables, tuples, projections of types, tagged unions, case expressions, function application, and type-abstraction application. Term and type abstractions have annotations (principal *pc*, channel environment Θ, set of places $\mathcal{P}$, and delegation context Π) that restrict how abstractions can be applied; we discuss this further when we present the type system. We explain non-standard parts of the syntax below, as they arise.

The operational semantics for DFLATE uses two judgments: one for sequential semantics and one for distributed semantics (see Section III-C). Sequential semantic judgment $\mathit{pl}, D \vdash e \longrightarrow e'$ indicates that at place *pl*, under *delegation sequence* D, expression e takes a small step to e'. Figure 4 presents some of the inference rules for this judgment. [5]

A delegation sequence is a sequence of delegations $\langle p \succcurlyeq q\rangle$, indicating that principal q has delegated its authority to principal p. We assume that there is a well-known initial delegation sequence D_{init}. Expression $\texttt{assume}\ \langle p \succcurlyeq q\rangle\ \texttt{in}\ e$ adds delegation $\langle p \succcurlyeq q\rangle$ to the delegation sequence used to evaluate e. This can be thought of as an annotation indicating that more information flows are permitted during the computation e. However, note that the type system ensures that it is secure to add this delegation, i.e., that the delegation to add and the decision to add it have sufficiently high integrity. We use term $e\ \texttt{where}\ v$ to record that delegation v holds for evaluation of e. Rules DE-ASSUME and DE-WHERE show how these terms operate. (Runtime representation of the delegation sequence and `where` terms are needed only for proof purposes and do not need to be present in an implementation of DFLATE.)

The monadic unit term $\eta_\ell\ e$ protects e at security level ℓ. This syntax is similar to that used by DCC [2] and FLAC [5], but the DFLATE type system treats monadic terms slightly differently in order to better model cryptographic protection mechanisms. The protection mechanism is left abstract, but DFLATE's design is consistent with standard cryptographic mechanisms like semantically secure asymmetric encryption [31] and existentially unforgeable signature schemes [23]. Intuitively $\eta_\ell\ e$ evaluates e to a value, and then encrypts and signs the value

[5] All inference rules are given in the accompanying technical report [25].

[PAR-STEP] $$\frac{n, D_{init} \vdash e \longrightarrow e'}{\mathcal{D} \parallel \langle n, e\rangle \Longrightarrow \mathcal{D} \parallel \langle n, e'\rangle}$$

[PAR-SPAWN]
$$\frac{\begin{array}{c} e = E[\texttt{spawn}\ @n'\ (ch_1[\mathit{pc}_1; \tau_1],\ ch_2[\mathit{pc}_2; \tau_2]).\ e_1\ \texttt{then}\ e_2] \\ \nu_1, \nu_2\ \text{fresh channels} \\ e' = E[e_2\{\nu_1/ch_1\}\{\nu_2/ch_2\}] \qquad e'_1 = e_1\{\nu_1/ch_1\}\{\nu_2/ch_2\} \end{array}}{\mathcal{D} \parallel \langle n, e\rangle \Longrightarrow \mathcal{D} \parallel \langle n, e'\rangle \parallel \langle n', e'_1\rangle}$$

[PAR-SEND-RECV]
$$\begin{array}{c} \mathcal{D} \parallel \langle n_1, E_1[\overline{\texttt{send}}\ \nu\ v\ \texttt{then}\ e_1]\rangle \parallel \langle n_2, E_2[\texttt{recv}\ \nu\ \texttt{as}\ x\ \texttt{in}\ e_2]\rangle \\ \Longrightarrow \mathcal{D} \parallel \langle n_1, E_1[e_1]\rangle \parallel \langle n_2, E_2[e_2\{v/x\}]\rangle \end{array}$$

Fig. 5: DFLATE distributed semantics

with keys appropriate for ℓ to protect it. Protected value $\overline{\eta}_\ell\ v$ represents the encrypted and signed value v (see rule DE-UNITM). For example, $\overline{\eta}_{\texttt{Alice}^{\leftarrow} \wedge \texttt{Bob}^{\rightarrow}}\ v$ represents value v signed by `Alice` and encrypted with `Bob`'s key. A protected value may flow to places that would be insecure for the unprotected value to go. A protected value can be used only via a monadic bind term $\texttt{bind}\ x = \overline{\eta}_\ell\ v\ \texttt{in}\ e$, which binds v to variable x in e (rule DE-BINDM). This is analogous to decrypting and verifying the signature of protected value $\overline{\eta}_\ell\ v$.

Expression $\texttt{TEE}^t\ s$ represents a TEE that will execute computation s. Syntactic category s (omitted in Figure 3) consists of expressions without `TEE` or `spawn` terms. This syntactically prevents nested or forking TEE code and reflects restrictions in existing TEE mechanisms. Each expression s is uniquely identified by a computation principal t, which can be thought of as a hash of the code s and can be used to identify a TEE. Assuming that t uniquely identifies s is compatible with the trust assumptions of most TEE designs: code is securely measured and the hash is unique up to collisions, which occur with negligible probability. Rule DE-TEE evaluates the source-level TEE term to the intermediate value $\texttt{runTEE}^t\ s$. Note that the t in $\texttt{runTEE}^t\ s$ is related to the source-level expression s; additional steps evaluate s, but t remains fixed.

C. Distributed semantics

Process $\langle n, e\rangle$ is expression e running on node n. A *distributed configuration* $\langle n_1, e_1\rangle \parallel \cdots \parallel \langle n_m, e_m\rangle$ is the parallel composition of processes $\langle n_i, e_i\rangle$. Without loss of generality, we assume that each node n_i in a distributed configuration is unique. We assume standard structural equivalence for distributed configurations and use metavariable $\mathcal{D}$ to range over distributed configurations.

Rules for distributed configurations are presented in Figure 5 and have the form $\mathcal{D} \Longrightarrow \mathcal{D}'$. Some of the rules use evaluation contexts [16] for sequential evaluation: $E[e]$ is an expression with subexpression e that can be reduced. Evaluation contexts are standard and defined in the accompanying technical report [25].

Rule PAR-STEP states that a distributed configuration takes a step whenever one of its processes takes a step. Note that the sequential evaluation of a process uses the initial delega-

tion sequence D_{init}, although the process may use additional delegations via `assume` and `where` terms.

Processes communicate via channels. Channel endpoints are unidirectional: an endpoint can be used to send or receive values, but not both. Communication is synchronous: a send blocks until there is a matching receive, and a receive blocks until a message is available. Channels are not first-class and we ensure that a channel endpoint is used by at most one process. This restriction prevents certain races that are both difficult for programmers to reason about as well as potential covert channels. Indeed, even though the distributed semantics are non-deterministic, because of the careful management of channel endpoints, there can never be a race between two sends on the same channel or between two receives on the same channel. Thus our distributed semantics is confluent.

Rule PAR-SEND-RECV matches up a process that is sending on channel ν with a process that is receiving on ν. For bookkeeping purposes in the proof, the value sent over the channel is a `where` term that includes all delegations in use by the sender. Rule DE-SEND ensures that delegations are included in the value before the communication occurs. We allow closures to be sent over channels, but the type system carefully ensures that the closure can not contain inappropriate channel endpoints nor can the closure contain TEE code.

Term `spawn @`n $(ch_1[pc_1;\tau_1],\ ch_2[pc_2;\tau_2])$. e_1 `then` e_2 spawns expression e_1 as a new process on node n and continues as e_2 (rule PAR-SPAWN). Expressions e_1 and e_2 may refer to channel variables ch_1 and ch_2, which, when the process is spawned, will be replaced with fresh channels, enabling the parent and child processes to communicate. Channel type annotations pc_1, τ_1, pc_2, and τ_2 restrict how channels may be used. (Spawn expressions have an additional form to facilitate spawning a process that executes a TEE and creating a channel between the parent process and the TEE. Details are in the accompanying technical report [25].)

IV. THREAT MODEL

The DFLATE type system statically enforces information-flow control policies on data processed by DFLATE programs. In order to understand various design choices in the type system, it is necessary to understand the attacker model.

We assume that some conjunction of principals (denoted $\mathcal{A}$ for "attacker") are malicious. Since nodes are principals, this also permits us to express that nodes are compromised. Intuitively, the security guarantees that we will provide (Section VII) are based on the idea that "you can be hurt only by those you trust." That is, if $\mathcal{A}$ is the malicious principal and p is a "good principal" (i.e., a principal that doesn't trust $\mathcal{A}$), then $\mathcal{A}$ can not violate the security concerns of p. We assume all processes start execution with common trust assumptions, i.e., the initial delegation sequence D_{init}.

For confidentiality, we assume that a good principal provides confidential input to a program and that the attacker observes the output of the program (namely, the final value computed on a compromised node). For integrity, we assume that the attacker provides untrustworthy input to a program and that a good principal consumes the output of the program. We thus use an "input/output" observational model.

We also consider a stronger observational model for confidentiality, where the attacker is able to observe the execution trace on a compromised node. However, the attacker cannot observe the contents of a ciphertext for which it does not have a decryption key: the attacker cannot distinguish values $\overline{\eta}_\ell\ v$ and $\overline{\eta}_\ell\ v'$ (which represent values v and v' encrypted and signed by principal ℓ) if the attacker does not have the authority to decrypt $\overline{\eta}_\ell\ v$ and $\overline{\eta}_\ell\ v'$. Similarly, the attacker cannot observe the contents of a TEE unless it has sufficient authority to access the keys of the corresponding computation principal.

We ignore covert channels, including timing, termination, memory accesses by TEEs, and speculative-execution channels. Orthogonal techniques (e.g., [42, 32, 27]) can mitigate some of these concerns, and we expect some covert channels related to TEEs to be addressed in future TEE designs.

We assume that cryptographic mechanisms, TEE implementations, and the compiler and runtime system are correct. We assume that node-to-node communication is secure and unobserved by other principals, i.e., we do not consider network-level adversaries. Tools such as Tor cite can be used to make it harder for network-level adversaries to observe the presence of node-to-node communication. We do, however, assume that communication with a TEE is observed by the host node. In DFLATE we use symbolic cryptography but do not treat keys as values in the language. We thus assume that an attacker has access to some set of signing and encryption keys based on trust relationships, but do not consider a Dolev-Yao-style attacker that can learn new keys from observations.

V. THE DFLATE SECURITY TYPE SYSTEM

DFLATE types (Figure 3) include unit, sums, products, functions, type functions, and type variables. (Functions and type functions have non-standard annotations that we describe below.) Delegation types are singleton types: each delegation type $(p \succcurlyeq q)$ is inhabited by a single value $\langle p \succcurlyeq q\rangle$. Monadic type ℓ `says` τ protects an expression of type τ at level ℓ; it is the type of values such as $\overline{\eta}_\ell\ v$ (where v has type τ).

Channels are not first-class values but do have types of the form $\texttt{chan}_{pl_1 \rightarrow pl_2}\ pc\ \tau$ and $\texttt{chan}_{pl_1 \leftarrow pl_2}\ pc\ \tau$. These types specify channels that connect places pl_1 and pl_2 (either nodes or TEEs) and may exchange values of type τ in contexts up to pc. Recall that channels are uni-directional. The former type specifies a *send channel*, (indicated by the subscript $pl_1 \rightharpoonup pl_2$) meaning pl_1 may use the channel to send values to pl_2, the latter specifies a *receive channel* (subscript $pl_1 \leftharpoonup pl_2$) meaning pl_1 may use the channel to receive values from pl_2.

Typing judgment $\Pi; \Gamma; \Theta; pl; pc \vdash e : \tau$ indicates that expression e has type τ. *Delegation context* Π contains a sequence of delegations that are valid just before executing e. It is a conservative approximation of the delegation sequence D that is present at run time. The delegation context is extended by `assume` and `where` terms. *Variable typing context* Γ maps variables to types. Channel variable scope is maintained using the *channel environment* Θ. Principal pl indicates the *place* the term is typed at, either a node n or computation

principal t. *Program counter level pc* is an upper bound (in the information-flow ordering $\sqsubseteq$) on the decision to execute e, and also a lower bound on observable side-effects of e.

The core DFLATE typing rules are presented in Figures 6 and 7 present some of the key DFLATE typing rules. Rules in Figure 6 are adapted from FLAC, and those in Figure 7 cover DFLATE's distributed computation and TEE extensions. Premises of these rules are either typing judgments or judgments that specify required relationships between principals, or between principals and types.

Acts-for judgments have the form $\Pi \Vdash p \succcurlyeq q$ and require that p has at least as much authority as q in delegation context Π. (Alternatively, that assuming the delegations in Π, q trusts p.) Recall that $\sqsubseteq$ is defined in terms of $\succcurlyeq$ so using the same rules we may also derive judgments of the form $\Pi \Vdash p \sqsubseteq q$. Intuitively, if $\Pi \Vdash p \sqsubseteq q$ information labeled with p can flow to information labeled q, since (given the delegations in Π) the confidentiality and integrity of q is at least as restrictive as that of p. Both of these judgments are simplified versions of the corresponding FLAM judgments [6], which we can use in DFLATE since delegations are reasoned about statically and thus do not provide an information channel. (FLAC could also benefit from this simplification.) More details are available in the technical report. The benefit of embedding DFLATE's acts-for judgment in the FLAM logic is that we can rely on FLAM's formal properties, which have been mechanically verified [7], in our proofs. In our technical report [25], we formalize this embedding rigorously, correcting some technical errors in the original FLAC [5] formalization.

Type protection judgments have the form $\Pi \vdash \ell \leq \tau$, indicating that type τ protects information labeled with ℓ. Intuitively, it means that the type system ensures that any information gained by using a value of type τ will have a security level at least as restrictive as ℓ. The rules for deriving type protection judgments are based on a subset of FLAC's rules. The primary rule is DP-LBL:

$$[\text{DP-LBL}] \qquad \frac{\Pi \Vdash \ell \sqsubseteq \ell'}{\Pi \vdash \ell \leq \ell' \texttt{ says } \tau}$$

This rule connects acts-for judgments to protected types. If ℓ flows to ℓ', then the type $\ell' \texttt{ says } \tau$ protects level ℓ. Singleton types like `unit` and $(p \succcurlyeq q)$ protect any level since the type itself encodes the value: observing the runtime value carries no information. However, the type $\tau_1 + \tau_2$ is not protected at any level since observing the value reveals the side of the sum the value is on, even if the sides have the same type. All protection rules are given the accompanying technical report [25].

DFLATE's type protection judgment is more restrictive than both FLAC's and the protection rules in the Dependency Core Calculus [2, 1] (which FLAC's are based on). The restrictiveness comes from the omission of three rules. One rule, DP-LBL1, permits a level to be protected by the inner type of a `says` type if the outer type does not protect it.

$$[\text{~~DP-LBL1~~}] \qquad \frac{\Pi \vdash \ell \leq \tau}{\Pi \vdash \ell \leq \ell' \texttt{ says } \tau}$$

This rule is not compatible with the cryptographic mechanisms DFLATE seeks to model: it makes nested `says` types commutative in the sense that $p \texttt{ says } q \texttt{ says } \tau$ protects the same levels as $q \texttt{ says } p \texttt{ says } \tau$. Commutativity undermines the expressiveness of integrity policies since a value of type τ signed by q then p (and thus unmodified by p) cannot be statically distinguished from a value signed by p then q (and thus unmodified by q). It also complicates reasoning about confidentiality since encryption order is not reflected statically.

The other two rules we omit assume that information can be gained from abstractions only by applying them. In a distributed setting, however, functions can be sent over channels to potentially malicious hosts, who can directly examine the encoding of an abstraction and potentially learn information.

Every DFLATE typing rule contains a *clearance premise* $\Pi \Vdash pl \succcurlyeq pc$ that requires place *pl* to act for *pc*. This ensures a place cannot observe or use data exceeding its authority, as discussed in Section II-B.

For function type $\tau_1 \xrightarrow{pc,\Theta,\mathcal{P},\Pi} \tau_2$, level *pc* is the *latent effect* of the function (i.e., a lower-bound on the observable side-effects when the function is invoked), Θ is the channel environment the function expects, Π is the delegation context the function expects, and $\mathcal{P}$ are the places at which the function make be invoked. Rule DT-LAM shows that the function body must be well-typed for the function's *pc* and channel environment, for every place $pl \in \mathcal{P}$. Function application (rule DT-APP) may occur only if the *pc* at call site flows to the latent effect of the function, the call-site place is in $\mathcal{P}$, and the channel environment and delegation context of the caller is compatible with the function's channel environment and delegation context. Note that any place can receive a lambda expression but only those within $\mathcal{P}$ are allowed to invoke it. Channels are not first class, and so the channel environment requirement ensures that the caller has the appropriate channels available and channel variables do not escape via closures. (Type abstraction and application is similar.)

Expression $\eta_\ell\ e$ will evaluate e and then protect the result at level ℓ (in implementation, by signing and encrypting it). It has type $\ell \texttt{ says } \tau$ (rule DT-UNITM) provided that e is well-typed and the program counter level *pc* flows to ℓ. Intuitively, this premise is required because program counter level *pc* is an upper bound on the decision to execute the statement and on the information available in this computational context (e.g., through variables). Thus, the result of e might be influenced by information at level *pc* and must be protected appropriately. Clearance ($\Pi \Vdash pl \succcurlyeq pc$) ensures that place *pl* has appropriate integrity to sign the value. Suppose Alice wants to protect a value at Bob's integrity by evaluating $\eta_{\texttt{Bob}}\ v$. To type check, it must be the case that $\Pi \Vdash pc \sqsubseteq \texttt{Bob}$. By clearance we have $\Pi \Vdash \texttt{Alice} \succcurlyeq pc$, and thus $\Pi \Vdash \texttt{Alice}^{\leftarrow} \succcurlyeq \texttt{Bob}^{\leftarrow}$, indicating Alice has access to Bob's signing key. Note that a principal can create protected values that are more confidential than its clearance, e.g., Alice can encrypt values using Bob's public encryption key without having access to Bob's decryption key.

Rule DT-SEALED permits protected values to be on nodes that would not have the authority to create them. For instance, even if Alice does not trust Bob, sealed value $\overline{\eta}_{\texttt{Alice}}\ v$ is

$$\text{[DT-Lam]}\quad \frac{\forall \mathit{pl}' \in \mathcal{P}.\ \Pi'; \Gamma, x{:}\tau_1; \Theta'; \mathit{pl}'; \mathit{pc}' \vdash e{:}\ \tau_2 \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash \lambda(x{:}\tau_1)[\mathit{pc}', \Theta', \mathcal{P}, \Pi'].\ e{:}\ \tau_1 \xrightarrow{\mathit{pc}', \Theta', \mathcal{P}, \Pi'} \tau_2}$$

$$\text{[DT-UnitM]}\quad \frac{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e{:}\ \tau \qquad \Pi \Vdash \mathit{pc} \sqsubseteq \ell \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash \eta_\ell\ e{:}\ \ell\ \texttt{says}\ \tau}$$

$$\text{[DT-Sealed]}\quad \frac{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash v{:}\ \tau \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash \overline{\eta}_\ell\ v{:}\ \ell\ \texttt{says}\ \tau}$$

$$\text{[DT-App]}\quad \frac{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e{:}\ \tau_1 \xrightarrow{\mathit{pc}', \Theta', \mathcal{P}, \Pi'} \tau_2 \qquad \Pi \Vdash \mathit{pc} \sqsubseteq \mathit{pc}' \qquad \mathit{pl} \in \mathcal{P} \qquad \forall (p \succcurlyeq q) \in \Pi'.\ \Pi \Vdash p \succcurlyeq q \qquad \Theta\,|_{\mathsf{dom}(\Theta')} = \Theta' \qquad \Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e'{:}\ \tau_1 \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e\ e'{:}\ \tau_2}$$

$$\text{[DT-BindM]}\quad \frac{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e{:}\ \ell\ \texttt{says}\ \tau_1 \qquad \Pi; \Gamma, x:\tau_1; \Theta; \mathit{pl}; \mathit{pc} \sqcup \ell \vdash e'{:}\ \tau_2 \qquad \Pi \vdash \mathit{pc} \sqcup \ell \leq \tau_2 \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash \texttt{bind}\ x = e\ \texttt{in}\ e'{:}\ \tau_2}$$

$$\text{[DT-Assume]}\quad \frac{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e{:}\ (q \succcurlyeq r) \qquad \Pi \Vdash \mathit{pc} \succcurlyeq \nabla(r) \qquad \Pi \Vdash \nabla(q^{\rightarrow}) \succcurlyeq \nabla(r^{\rightarrow}) \qquad \Pi \cdot \langle q \succcurlyeq r \rangle; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash e'{:}\ \tau \qquad \Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}}{\Pi; \Gamma; \Theta; \mathit{pl}; \mathit{pc} \vdash \texttt{assume}\ e\ \texttt{in}\ e'{:}\ \tau}$$

Fig. 6: Core sequential typing rules

well-typed at Bob if v is well-typed. Sealed values reflect the security guarantees of cryptographic protection mechanisms: that attackers cannot distinguish ciphertexts or forge signatures.

In $\texttt{bind}\ x = e\ \texttt{in}\ e'$, expression e evaluates to a protected value $\overline{\eta}_\ell\ v$, and x is bound to v in e'. Rule DT-BindM requires that the type of e' must protect $\mathit{pc} \sqcup \ell$ and e' must type check at a more restrictive level $\mathit{pc} \sqcup \ell$. Clearance (for expression e') ensures that $\mathit{pc} \sqcup \ell$ does not exceed the place's authority: the place is trusted to compute on data protected at level ℓ. For example, if Bob evaluates $\texttt{bind}\ x = \overline{\eta}_{\texttt{Alice}}\ v\ \texttt{in}\ e'$, for e' to type check it must be the case that $\Pi \Vdash \texttt{Bob} \succcurlyeq \mathit{pc} \sqcup \texttt{Alice}$. It follows that $\Pi \Vdash \texttt{Bob}^{\rightarrow} \succcurlyeq \texttt{Alice}^{\rightarrow}$, indicating Bob has access to Alice's decryption key.

DT-Assume ensures that when the delegation context is extended, there is sufficient integrity to do so. Specifically, when r delegates to q, r's security concerns may be compromised, so we require that *pc* acts for $\nabla(r)$, the voice of r. Premise $\Pi \Vdash \nabla(q^{\rightarrow}) \succcurlyeq \nabla(r^{\rightarrow})$ ensures *robustness* of the delegation, a desirable property from FLAM [6] that we also enforce.

Figure 7 shows the distributed and TEE typing rules. Rule DT-Spawn limits the channel environment of newly spawned computations to the new channels created by the parent. Only place *pl* has access to the send endpoint of ch_1 and the receive endpoint of ch_2. Conversely, the newly created process on node n can use the receive endpoint of ch_1 and the receive endpoint of ch_2. Spawned processes e inherit the delegation context from the parent, but not the variable context. The program counter level of the spawned process e_n, pc', is at least as restrictive as the *pc* of the parent process. This ensures that e_n does not inadvertently reveal that it was spawned.

DT-Send requires that channel ch is the `send` endpoint and that the expression has the correct type. The channel program counter level pc_{ch} is an upper bound on the confidentiality and integrity of the decision to send the message. This is distinct from the policy used to restrict what information can be sent in messages, which is expressed via type τ (see Section II-A for discussion). After the message is sent, execution proceeds with e' which must type check at a program counter level that is at least as restrictive as pc_{ch}. This ensures that information revealed by successfully sending a message is protected appropriately. In addition to the usual clearance premise, DT-Send also has a channel clearance premise $\Pi \Vdash \mathit{pl} \succcurlyeq \mathit{pc}_{ch}$ that ensures p has sufficient authority to use the channel. Rule DT-Receive is similar to DT-Send . The type of channel messages τ and the channel program counter level pc_{ch} allow the sender and receive to co-ordinate on the security and contents of messages sent over the channel.

Expression $\texttt{TEE}^t\ e$ executes e in a TEE. Rule DT-TEE requires e to be closed (and so it cannot use variables to access data from the host node). The channel environment for the TEE is limited to endpoints for the TEE[6] with a channel pc that protects *pc*. This restriction ensures two properties. First, all messages into and out of a TEE pass through the TEE's host, which reflects the operation of current TEE implementations. Second, the restriction to only channels with suitable channel pc's ensures that any sends and receives the TEE perform also protect the *pc* that launched the TEE. Without this second property, hosts could use TEEs as covert channels to send messages from restrictive contexts to less restrictive ones.

Expression e executes with the integrity of t and confidentiality $\mathit{pc}^{\rightarrow}$. Rule DT-TEE differs from all other typing rules in that there is no relation between the integrity of the program counter where the TEE is executed ($\mathit{pc}^{\leftarrow}$) and the integrity of the program counter within the TEE ($t^{\leftarrow}$): this is a form of endorsement. Computation principal t is unique for a given expression e and the implementation of DFLATE can use remote attestation to ensure that the TEE is executing e, even if the host is untrusted. The typing rule reflects this guarantee by type checking e at an integrity level unique to that expression. This ensures that the code e is not altered (e.g., by malware) before the execution. Thus, principals that delegate trust to t will consider the TEE trusted, but the TEE gains no additional authority over principals that do not delegate to t. The confidentiality level of the information revealed by the TEE is at least that of the host and thus expression e is type checked with confidentiality $\mathit{pc}^{\rightarrow}$.

A. Examples revisited

Figure 8 presents DFLATE code for the three scenarios in Figure 2. Each program applies function `f` to a protected value from Alice (principal a) and outputs the result to Carol (c) using Bob (b) as an intermediate, protecting the output at level $a \sqcap c$, implying Alice and Carol can read it and both trust its

[6]An extended version of the `spawn` expression is used to establish channels between the host and TEE. Remote nodes can not have a channel directly to the TEE. See the technical report for details.

[DT-SPAWN]

$$\frac{\Pi;\emptyset;[ch_1 \mapsto \mathtt{chan}_{n \leftarrow pl}\ pc_1\ \tau_1, ch_2 \mapsto \mathtt{chan}_{n \rightarrow pl}\ pc_2\ \tau_2];n;pc' \vdash e_n : \mathtt{unit} \quad \Pi;\Gamma;\Theta[ch_1 \mapsto \mathtt{chan}_{pl \rightarrow n}\ pc_1\ \tau_1, ch_2 \mapsto \mathtt{chan}_{pl \leftarrow n}\ pc_2\ \tau_2];pl;pc \vdash e : \tau \quad \Pi \Vdash pl \succcurlyeq pc \quad \Pi \Vdash pc \sqsubseteq pc'}{\Pi;\Gamma;\Theta;pl;pc \vdash \mathtt{spawn}\ @n\ (ch_1[pc_1;\tau_1],\ ch_2[pc_2;\tau_2]).\ e_n\ \mathtt{then}\ e : \tau}$$

[DT-SEND]

$$\frac{\Pi;\Gamma;\emptyset;pl;pc \vdash e : \tau \quad \Pi;\Gamma;\Theta;pl;pc' \vdash e' : \tau' \quad \Pi;\Gamma;\Theta;pl;pc \vdash ch : \mathtt{chan}_{pl \rightarrow pl'}\ pc_{ch}\ \tau \quad \Pi \Vdash pc \sqsubseteq pc_{ch} \quad \Pi \Vdash pc_{ch} \sqsubseteq pc' \quad \Pi \vdash pc' \leq \tau' \quad \Pi \Vdash pl \succcurlyeq pc \quad \Pi \Vdash pl \succcurlyeq pc_{ch}}{\Pi;\Gamma;\Theta;pl;pc \vdash \mathtt{send}\ ch\ e\ \mathtt{then}\ e' : \tau'}$$

[DT-RECEIVE]

$$\frac{\Pi;\Gamma;\Theta;pl;pc \vdash ch : \mathtt{chan}_{pl \leftarrow q}\ pc_{ch}\ \tau \quad \Pi;\Gamma,x:\tau;\Theta;pl;pc' \vdash e : \tau' \quad \Pi \Vdash pc \sqsubseteq pc_{ch} \quad \Pi \Vdash pc_{ch} \sqsubseteq pc' \quad \Pi \vdash pc' \leq \tau' \quad \Pi \Vdash pl \succcurlyeq pc \quad \Pi \Vdash pl \succcurlyeq pc_{ch}}{\Pi;\Gamma;\Theta;pl;pc \vdash \mathtt{recv}\ ch\ \mathtt{as}\ x\ \mathtt{in}\ e : \tau'}$$

[DT-TEE]

$$\frac{\Pi;\emptyset;\Theta';t;t^{\leftarrow} \wedge pc^{\rightarrow} \vdash e : \tau \quad C = \{ch \mid \Theta(ch) = \mathtt{chan}_{t=pl}\ pc'\ \tau\ \wedge\ \Pi \Vdash pc \sqsubseteq pc'\} \quad \Theta\mid_C = \Theta' \quad \Pi \Vdash p \succcurlyeq pc}{\Pi;\Gamma;\Theta;pl;pc \vdash \mathtt{TEE}^t\ e : \mathtt{unit}}$$

Fig. 7: Core distributed and TEE typing rules

```
1 spawn @b (ch_b[a ∨ b ∨ c; int], ch'_b[pc'_b; τ̇]) {
2   spawn @c (ch_c[a ∨ b ∨ c; int], ch'_c[pc'_c; τ̇]) {
3     recv ch_c as x in [η_{a⊓c} (f x)]
4   }
5   recv ch_b as y in send ch_c y then ()
6 }
7 bind z = η_a v in send ch_b z then ()
```

(a) DFLATE code for Figure 2(a).

```
1 spawn @b (ch_b[a ∨ b ∨ c; a says int], ch'_b[pc'_b; τ̇]) {
2  spawn @c (ch_c[a ∨ b ∨ c; a says int], ch'_c[pc'_c; τ̇]) {
3    recv ch_c as y in
4      bind x = y in [η_{a⊓c} (f x)]
5  }
6  recv ch_b as z in send ch_c z then ()
7 }
8 send ch_b (η_a v) then ()
```

(b) DFLATE code for Figure 2(b).

```
1 spawn @b (ch_b[(a ∨ b)^←; a says int], ch'_b[pc'_b; unit]) {
2  spawn @c (ch_c[(a ∨ b ∨ c ∨ t)^←; a says int], ch'_c[pc'_c; unit]) {
3     recv ch_c as y in bind x = y in x
4  }
5  spawn @b (ch_t[(a ∨ b ∨ t)^←; a says int],
6            ch'_t[(a ∨ b ∨ c ∨ t)^←; a ⊓ c says int]) {
7     TEE^t {
8      assume b^← ≽ t^← in assume c^→ ≽ a^→ in
9       recv ch_t as y in
10       send ch'_t (bind x = y in [η_{a⊓c} (f x)]) then ()
11    }
12 }
13 recv ch_b as z in send ch_t z then
14    recv ch'_t as y in send ch_c y then ()
15 }
16 send ch_b (η_a v) then ()
```

(c) DFLATE code for Figure 2(c).

Fig. 8: DFLATE code

integrity. Despite similar functionality, each program requires different trust relationships between Alice, Bob, and Carol.

Figure 8(a) is an implementation of Figure 2(a), where no cryptographic mechanisms are used. For this program to type check under some delegation context Π, it must be the case that Alice trusts Bob and Carol completely. That is, $\Pi \Vdash b \succcurlyeq a$ and $\Pi \Vdash c \succcurlyeq a$. Furthermore, Carol must trust Alice and Bob with her integrity, $\Pi \Vdash a^{\leftarrow} \succcurlyeq c^{\leftarrow}$ and $\Pi \Vdash b^{\leftarrow} \succcurlyeq c^{\leftarrow}$. To see why, first consider the `send` in line 7. For this term to type check, it must be the case that $\Pi \Vdash a \sqsubseteq a \vee b \vee c$ since, by DT-BINDM , the *pc* at this point is at least as restrictive as the level a on the protected value $\eta_a\ v$, and by DT-SEND , this *pc* must flow to the channel *pc*, $a \vee b \vee c$. If $\Pi \Vdash a \sqsubseteq a \vee b \vee c$ holds, from the definitions of $\sqsubseteq$, $\sqcup$, and $\sqcap$ (Section III-A), it follows that $\Pi \Vdash b^{\rightarrow} \succcurlyeq a^{\rightarrow}$ and $\Pi \Vdash c^{\rightarrow} \succcurlyeq a^{\rightarrow}$. A similar argument for the `recv` at line 3 implies $\Pi \Vdash c^{\rightarrow} \succcurlyeq a^{\rightarrow}$.

Now consider line 3, where function `f` is applied and the result protected at $a \sqcap c$. DT-RECEIVE requires that the *pc* of the continuation is at least $a \vee b \vee c$, and DT-UNITM requires that this *pc* is protected by $a \sqcap c$, i.e., $\Pi \Vdash a \vee b \vee c \sqsubseteq a \sqcap c$. For this to hold, it must be the case that $\Pi \Vdash a \vee b \vee c \succcurlyeq a$ and $\Pi \Vdash a \vee b \vee c \succcurlyeq c$. These judgments imply all the following:

$$\Pi \Vdash b^{\leftarrow} \succcurlyeq a^{\leftarrow} \qquad \Pi \Vdash c^{\leftarrow} \succcurlyeq a^{\leftarrow}$$
$$\Pi \Vdash a^{\leftarrow} \succcurlyeq c^{\leftarrow} \qquad \Pi \Vdash b^{\leftarrow} \succcurlyeq c^{\leftarrow}$$

Figure 8(b) is an implementation of Figure 2(b). Recall that in Figure 2(b), Alice signs and encrypts her message to Carol, and Bob does not learn the contents of this message. In other words, Alice trusts Carol with her confidentiality and integrity but does not trust Bob with her confidentiality. However, she needs to trust Bob's integrity because of his power to suppress the message. The DFLATE program shown in Figure 8(b) reflects the same trust relations. Values are protected when sent over channels. Hence, the channel type for ch_b and ch_c is a `says int` rather than just `int` as in Figure 8(a). This program requires Alice to delegate her confidentiality and integrity to Carol, but does not require her to delegate her confidentiality to Bob. However, Alice and Carol must still trust Bob's integrity since he can influence the computation by suppressing Alice's message. To see why: the protected value $\eta_a\ v$ is never used in a `bind` on Bob's node, so there is no requirement that Alice trust Bob with her secrets. When Carol binds the value in order to apply function `f`, the *pc* at the box in line 4 is at $(a \vee b \vee c) \sqcup a$ which is equal to $a^{\rightarrow} \wedge (a \vee b \vee c)^{\leftarrow}$ by definition of $\sqcup$ and lattice absorption.[7] Therefore, DT-UNITM requires that $\Pi \Vdash a^{\rightarrow} \wedge (a \vee b \vee c)^{\leftarrow} \sqsubseteq a \sqcap c$, which implies that $\Pi \Vdash c^{\rightarrow} \succcurlyeq a^{\rightarrow}$ as well as the same integrity relationships

[7]The absorption laws $(a \vee b) \wedge a = a$ and $(a \wedge b) \vee a = a$ for all lattice elements a and b are algebraic properties of all lattices.

implied by Figure 8(a).

Figure 8(c) is an implementation of Figure 2(c). Recall that Bob can influence the incoming and outgoing messages to the enclave E but the recipients (of messages from enclave) can detect the modifications. Thus it suffices for Alice and Carol to trust the enclave E. The DFLATE program in Figure 8(c) uses a TEE to further reduce the need for mutual trust among Alice, Bob, and Carol. By running the computation in a TEE (identified as t) hosted on potentially untrusted Bob's node `b` (lines 5 and 6), Alice no longer needs to delegate integrity to Carol since Carol has no influence on the computation. Since the enclave protects the result (`f x`) at line 9 on behalf of Alice and Carol, they must each delegate integrity to $t^{\leftarrow}$ (Alice's integrity is also required for the `assume` at line 7).

$$\Pi \Vdash t^{\leftarrow} \succcurlyeq a^{\leftarrow} \qquad \Pi \Vdash t^{\leftarrow} \succcurlyeq c^{\leftarrow} \qquad \Pi \Vdash a^{\leftarrow} \succcurlyeq c^{\leftarrow}$$

After receiving `y` on channel ch_t, the *pc* at line 9 is $(a \vee b \vee t)^{\leftarrow}$, reflecting Bob's influence relaying messages. In order to bind `y` to `x`, the enclave must have clearance to read $a^{\rightarrow}$, so Alice must delegate her confidentiality to $t^{\leftarrow}$. Since Bob is unable to modify Alice's message, the enclave endorses Bob's influence by assuming $b^{\leftarrow} \succcurlyeq t^{\leftarrow}$, and allows the result of (`f x`) to flow to Carol by assuming $c^{\rightarrow} \succcurlyeq a^{\rightarrow}$. These assumptions allow the body of the `bind` to type check. The below judgments show the assumptions needed inside the TEE, (Π_{tee} is the TEE's delegation context at lines 8 and 9.)

$$\Pi_{tee} \Vdash b^{\leftarrow} \succcurlyeq t^{\leftarrow} \qquad \Pi_{tee} \Vdash c^{\rightarrow} \succcurlyeq a^{\rightarrow}$$

Introducing temporary delegations in a TEE using `assume` is preferable to the delegation contexts required by 8(a) or 8(b) since they are enabled only for the scope of the TEE, and the TEE guarantees that the code is executed as-is.

VI. Implementation Considerations

In this section we discuss how DFLATE can be realized using existing cryptographic techniques and TEE mechanisms. Specifically, we identify security principals with public keys, and rely on a public key infrastructure to distribute private keys to appropriately authorized nodes. Our TEE abstraction is carefully designed to be implementable using Intel's SGX and similar mechanisms; we describe how the remote attestation mechanism can be used by the DFLATE runtime to authenticate TEEs and provision the TEE with appropriate private keys.

A. Cryptography and Representation of Principals

We require that every primitive principal $n \in \mathcal{N}$ (which includes nodes and computation principals) is associated with a public/private key pair where the public key can be used for encryption and verifying signatures, and the private key can be used for decryption and signing.[8] There are many possible cryptographic schemes that can be used, and we do not require any specific one. We do, however, require infrastructure to store and distribute keys, which we discuss below.

[8]The public key will likely be a tuple of an encryption key and a verification key, and similarly for the private key.

A conjunction or disjunction of principals is represented by a distinct key pair. That is, the cryptographic scheme does not need to support group encryption, group signatures, etc. Instead, our key infrastructure will provide appropriate access control for private keys of conjuncts and disjuncts of principals such as `Alice` $\wedge$ `Bob` and `Alice` $\vee$ `Bob` $\vee$ `Carol`.

Computation principals $t \in \mathcal{T}$ are identified by a secure hash of (the bytecode representation of) the corresponding computation, and are associated with a key pair. We describe below how a TEE executing the code corresponding to computation principal t is provisioned with the key pair for t.

Value $\overline{\eta}_p\ v$ represents value v encrypted and signed by p.[9] The value $\overline{\eta}_p\ v$ is implemented by first encrypting v (using the appropriate key for $p^{\rightarrow}$), and then signing the result (using the appropriate key for $p^{\leftarrow}$). Conversely, evaluation of `bind` $x =$ $\overline{\eta}_p\ v$ `in` e verifies then decrypts. We ensure that places (i.e., nodes and TEEs) that need to perform decryption and signing have access to the appropriate keys; see below.

Delegations $\langle p \succcurlyeq q \rangle$ are run-time values, and are implemented as a statement "q delegates to p" that is signed appropriately (i.e., signed by the principal $\nabla(q)$).

We assume that node-to-node communication is secure and unobservable, which can be achieved using tools such as Tor [15] or Riffle [26].

B. TEE Implementation

Intel's SGX is the most widely deployed TEE mechanism, although other TEE implementations exist (e.g., Sanctum [14]). Modulo security vulnerabilities[10] and the need to trust Intel, SGX is suitable for implementing DFLATE's TEE abstraction.

To start executing `TEE`^{t} e, first an SGX enclave is created with the DFLATE runtime. SGX's remote attestation mechanism can be used to prove that the enclave is running the DFLATE runtime. Once a remote party knows it is communicating with an instantiation of the DFLATE runtime, the DFLATE runtime can state that it is executing computation e whose hash is t.

C. DFLATE Runtime

The DFLATE runtime system is responsible for executing DFLATE code, establishing communication channels between nodes, dynamically type-checking values (especially closures) that are received over channels, interacting with the SGX mechanisms and our key management infrastructure, and other tasks required to support execution of DFLATE programs.

The DFLATE runtime needs to be able to execute inside an SGX enclave. Current SGX SDK support is limited to C and C++, so the DFLATE runtime would be most easily implemented in C or C++. However, DFLATE code is represented as bytecode that is executed by the DFLATE runtime. This is necessitated by the ability to send closures over channels, but also simplifies our use of SGX remote attestation protocols.

[9]By contrast, term $\eta_p\ e$ will evaluate e and then encrypt and sign the result.

[10]Recent security vulnerabilities discovered in SGX [38] appear to be implementation issues rather than fundamental concerns.

D. Key Distribution

Computation at a place will need to encrypt, decrypt, sign, and verify data. While encryption and signature verification use the public part of a key pair, decryption and signing require that the place possess appropriate private keys. Fortunately, the clearance premises in the typing rules ensure that a well-typed program at place *pl* will need to perform decryptions and signatures only for principals p such that $\Pi_{init} \Vdash pl \succcurlyeq \nabla(p)$ (where Π_{init} is the initial delegation context and $\nabla(p)$ is the authority required to act on behalf of p).

When an enclave is created for computation principal t, the enclave does not initially have the private key for t, nor for any other principals p such that $\Pi_{init} \Vdash t \succcurlyeq \nabla(p)$. Thus the enclave must be provisioned with appropriate keys at run time.

To address this, we require a global *key master* component that can store key pairs and allow nodes and enclaves to acquire the private keys that they are authorized to have. Moreover, the key master creates key pairs as needed for conjuncts and disjuncts of principals. The key master can be implemented as a distributed service to reduce trust in any single entity.

Node n can request the private key for principal p from the key master by proving that it is n (i.e., that it possesses the private key for n), whereupon the key master will check that $\Pi_{init} \Vdash n \succcurlyeq \nabla(p)$, and, if so, securely send n the private key for p. Since we can conservatively approximate the principals occurring in a computation, node n could be provisioned with the appropriate private keys before execution, or the implementation could allow n to request private keys lazily during execution.[11]

However, for a computation principal t, the provisioning of private keys is slightly different, and must be performed at run time. First, the key master and the SGX enclave engage in a remote attestation protocol. Once the key master has proof that the enclave is running the DFLATE runtime, it establishes a secure channel with the enclave. The DFLATE runtime then informs the key master it is executing the code corresponding to computation principal t, and requests private keys. Notably, this is the only place that the SGX remote attestation protocol is needed in our proposed DFLATE implementation: secure communication between a node and a TEE can be established using keys for DFLATE principals. Each enclave needs to run the remote attestation protocol only once, with the key master, in order to acquire keys for DFLATE principals.

VII. Security Guarantees

DFLATE's type system enforces information-flow policies expressed using the FLAM principal algebra, and thus enjoys noninterference-based security guarantees. DFLATE permits weakening, or *downgrading*, of policies.[12] Downgrading occurs by adding delegations (via `assume` terms) and by TEE execution (via endorsement of the TEE's program counter level).

However, downgrading in DFLATE is carefully controlled and restricted: well-typed `assume` terms can only execute in contexts with sufficient integrity, and endorsement of TEEs reflect measurement and verification of code executing in a TEE. We thus expect that well-typed DFLATE programs satisfy a variety of expressive noninterference-based security guarantees, based on controlled downgrading (e.g., [12, 24, 8, 9, 13, 3]), suitably adapted to be consistent with our threat model IV.

To demonstrate that DFLATE does indeed enjoy noninterference-based properties, we state and prove two variants of noninterference. The first (Theorem 1) uses a "batch-job" model, and holds for confidentiality and integrity. In a batch-job model, inputs are provided at the beginning of execution, and outputs are provided if and when the program terminates [30]. For our purposes, we regard the input as being data on one node (thus modeling that node possessing confidential information, or that node containing untrustworthy data) and the output as the final result on a specific node. Even though the execution of a DFLATE program involves nodes interacting with each other over channels, the batch model ensures that ultimate result of the program is appropriately secure.

The second (Theorem 2) uses a stronger observational model where an attacker observes the internal state of a compromised node, but holds only for confidentiality. It does not hold for integrity, due to asymmetry in security guarantees inherent in distributed decentralized applications that use TEEs.

A. Batch-Job Noninterference

We state noninterference with respect to a security level H. Intuitively, for confidentiality, inputs labeled H (or a more restrictive security level) are regarded as confidential inputs, and we are concerned with ensuring that no information about them is revealed in outputs observable by an attacker, i.e., an entity that can observe outputs at level ℓ where it is not the case that H can flow to ℓ. For integrity, inputs labeled H are regarded as low-integrity and we want to ensure they do not influence high-integrity outputs (i.e., outputs at level ℓ).

Since we are stating noninterference, we are concerned only with executions where there is no downgrading from H (or above) to ℓ. However, we do not want to rule out all downgrading, as delegations and TEEs are central to DFLATE's expressiveness. Instead, we assume that for a given process $\langle n_k, e_k \rangle$ in a well-typed distributed configuration $\mathcal{D}$, we have a *delegation approximation* $\hat{\Pi}_k^{\mathcal{D}}$ that over-approximates delegations that the process may make during execution.[13] See accompanying technical report [25] for a formal definition.

Suppose all processes in $\mathcal{D} = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle$ are well-typed, and the ith process takes an input value protected by H (i.e., it has a free variable of type H `says` τ) and the jth process produces a value of type ℓ `says bool`. Moreover, suppose the delegation approximations for the processes ensure that they never downgrade from H (i.e., $\hat{\Pi}_k^{\mathcal{D}}$ permits the same flows from H as Π_{init}). If we have two executions of $\mathcal{D}$ where the input to the ith process is replaced with different values, then the result of the jth process will be the same. We

[11] Care must be taken to ensure that the decision to communicate with the key master does not reveal confidential information.

[12] Weakening confidentiality is called *declassification* [35]; weakening integrity is called *endorsement* [11].

[13] A straightforward static analysis can be used to compute delegation approximations, but any over-approximation suffices for the security condition.

state this formally. (Proofs are in the accompanying technical report [25].)

Theorem 1 (Batch-Job Noninterference)**.** *Let H and ℓ be security levels such that $\Pi_{init} \not\Vdash H \sqsubseteq \ell$. Let $\mathcal{D} = \langle n_1, e_1\rangle \parallel \cdots \parallel \langle n_m, e_m\rangle$ such that for all $k \in 1..m$ we have*

$$\Pi_{init}; \Gamma_k; \Theta_k; n_k; pc_k \vdash e_k : \tau_k,$$

where $\tau_j = \ell$ `says bool` and $x : H$ `says` $\tau \in \Gamma_i$.

Assume that no process downgrades from H, i.e., $\forall k \in 1 \ldots m.\ \forall \ell'.\ \Pi_{init} \Vdash H \sqsubseteq \ell' \Leftrightarrow \hat{\Pi}_k^{\mathcal{D}} \Vdash H \sqsubseteq \ell'$. For all v_1 and v_2, and all $z \in \{1, 2\}$ such that

$$\Pi; \Gamma_i; \Theta_i; n_i; \mathbf{pc}_i \vdash v_z : H \texttt{ says } \tau,$$

let $\mathcal{D}_z = \langle n_1, e_1\rangle \parallel \cdots \parallel \langle n_i, e_i\{v_z/x\}]\rangle \parallel \cdots \parallel \langle n_m, e_m\rangle$.

If $\mathcal{D}_1 \Longrightarrow^* \langle n_1, e_1'\rangle \parallel \cdots \parallel \langle n_j, v_j'\rangle \parallel \cdots \parallel \langle n_{m'}, e_{m'}'\rangle$ and $\mathcal{D}_2 \Longrightarrow^* \langle n_1, e_1''\rangle \parallel \cdots \parallel \langle n_j, v_j''\rangle \parallel \cdots \parallel \langle n_{m''}, e_{m''}''\rangle$ then $v_j' = v_j''$.

B. Noninterference for Stronger Observational Model

We also prove a stronger confidentiality noninterference result for an attacker that is able to observe the execution of a process at a compromised node. Intuitively, the attacker sees the sequence of expressions (with stuttering removed, since we ignore timing channels) but cannot see the contents of protected values or TEEs for which the node does not have the decryption key. Recall (from Section VI) that node n has access to keys for all principals p such that $\Pi_{init} \Vdash n \succcurlyeq \nabla(p)$. Thus, the attacker cannot observe the contents of protected value $\overline{\eta}_\ell\ v$ if $\Pi_{init} \not\Vdash n \succcurlyeq \nabla(\ell^{\rightarrow})$, nor see the contents of a TEE for computation principal t if $\Pi_{init} \not\Vdash n \succcurlyeq \nabla(t^{\rightarrow})$.

The formal definitions of *the process trace of node n* and *equivalence of process traces* are given in the accompanying technical report. We say that two process traces are equivalent to node n if n is unable to distinguish them.

Our stronger noninterference result is similar to the result of Theorem 1, but holds only for confidentiality.

Theorem 2 (Compromised-node Noninterference)**.** *Let $H^{\rightarrow}$ and n_j be security levels such that $\Pi_{init} \not\Vdash n_j \succcurlyeq H^{\rightarrow}$. Let $\mathcal{D} = \langle n_1, e_1\rangle \parallel \cdots \parallel \langle n_m, e_m\rangle$ such that for all $k \in 1..m$ we have*

$$\Pi_{init}; \Gamma_k; \Theta_k; n_k; pc_k \vdash e_k : \tau_k,$$

where $x : H^{\rightarrow}$ `says` $\tau \in \Gamma_i$.

Assume that no process downgrades from $H^{\rightarrow}$, i.e., $\forall k \in 1 \ldots m.\ \forall \ell.\ \Pi_{init} \Vdash H^{\rightarrow} \sqsubseteq \ell \Leftrightarrow \hat{\Pi}_k^{\mathcal{D}} \Vdash H^{\rightarrow} \sqsubseteq \ell$. For all v_1 and v_2, and all $z \in \{1, 2\}$ such that

$$\Pi; \Gamma_i; \Theta_i; n_i; \mathbf{pc}_i \vdash v_z : H^{\rightarrow} \texttt{ says } \tau,$$

let $\mathcal{D}_z = \langle n_1, e_1\rangle \parallel \cdots \parallel \langle n_i, e_i\{v_z/x\}]\rangle \parallel \cdots \parallel \langle n_m, e_m\rangle$.

Then for all executions $\mathcal{D}_1 \Longrightarrow^* \mathcal{D}_1'$ and $\mathcal{D}_2 \Longrightarrow^* \mathcal{D}_2'$ the process traces of node n are equivalent.

An equivalent of Theorem 2 does not hold for integrity. This asymmetry is due to the message suppression ability of the attacker. Consider the following program on nodes n_1 and n_2.

$$\left\langle n_1,\ \begin{array}{l}\texttt{bind u = x in}\\ \texttt{case u of}\\ \quad \texttt{inj}_1\texttt{(z). send } ch\ \texttt{() then ()}\\ \quad \texttt{inj}_2\texttt{(z). recv } ch'\texttt{ as y in y}\end{array} \right\rangle \Bigg\| \left\langle n_2,\ \begin{array}{l}\texttt{recv } ch \texttt{ as } y \texttt{ in}\\ \quad \texttt{()}\end{array} \right\rangle$$

On the left, the attacker node binds low-integrity input `x` (of type $H^{\leftarrow}$ `says bool`) to `u` and branches on the value, sending on channel ch in one branch and receiving on channel ch' in the other. On the right, a high-integrity node is engaged in communication with n_1 and terminates after receiving a message. For inputs $\texttt{x} = \overline{\eta}_{H^{\leftarrow}}\ \texttt{inj}_1\ ()$, process n_2 terminates, but for $\texttt{x} = \overline{\eta}_{H^{\leftarrow}}\ \texttt{inj}_2\ ()$ it blocks, thus distinguishing the executions.

The weaker integrity result validates our goal of faithfully expressing the power of attackers to suppress messages without eclipsing the guarantees provided by the cryptographic mechanisms and TEEs. DFLATE cannot protect against the suppression of high-integrity messages, but for all programs that result in high-integrity messages, Theorem 1 guarantees their contents have not been influenced by an attacker.

VIII. Related work

A. Enclaves and Information Flow

Gollamudi and Chong [24] use enclaves to enforce information flow policies against low-level attackers that can inject arbitrary code into non-enclave parts of a program. DFLATE uses enclaves to enforce confidentiality and integrity against low-level attackers in a distributed setting. Our current noninterference results model passive attackers; we leave more powerful attacker models for future work.

In CFLOW, Fournet et al. [19] compile a sequential imperative program into a distributed program, preserving its security properties using cryptographic techniques. A straightforward security type system enforces noninterference. Fournet and Planul [17] extend CFLOW to use Trusted Platform Modules (TPM) and remote attestation to minimize the TCB while preserving noninterference. DFLATE programs are explicitly distributed at the source level via `spawn`, `send`, and `recv` terms. CFLOW's communication channels are always public and untrusted whereas DFLATE channels specify separate policies for the presence of a message and its contents. CFLOW's TPM is trusted and has a fixed integrity level, but TEEs in DFLATE have distinct integrity and confidentiality levels, allowing TEEs to be trusted than their host.

Subramanyan et al. [37] provide a formal foundation for the remote execution of enclaves and use it to prove that two remote enclave executions emit observationally equivalent traces if the attacker provides the same inputs in both executions. DFLATE uses the high-level guarantees of TEEs and proves end-to-end semantic guarantees (noninterference) of distributed applications using enclaves.

B. Communication Channels and Cryptography

Zdancewic et al. [40] securely partition a program into subprograms that communicate to simulate the original program. The resulting distributed program prevents *read channels*, which leak information when a remote read request occurs

for a secret reason. DFLATE's channel pc annotations protect against similar leaks.

Fournet and Rezk [18] use a security type system to enforce the correct use of cryptographic primitives for controlled downgrading. Compiling DFLATE to this language would ensure DFLATE's monadic abstractions are implemented securely.

Wysteria [34] is a language for writing secure multiparty computation protocols. *Wires* in Wysteria express the idea of data ownership and are comparable to the monadic unit types in DFLATE. However, because Wysteria models communication implicitly through variable binding, it does not detect insecure flows that arise due to explicit communication.

Gazeau et al. [20] enforce confidentiality (but not integrity) of the client data in the cloud. Like our assumptions regarding access to cryptographic keys, their security guarantee relies on honest nodes denying access to attacker nodes.

Fabric [28] and DStar [41] use static and dynamic mechanisms to enforce IFC for distributed programs. They use cryptographic protocols to establish secure channels between nodes, but unlike DFLATE, do not allow high-integrity or secret data to flow through untrusted hosts.

Our channel design is similar to Rafnsson et al. [33], who also distinguish the presence of a message from the contents of a message. DFLATE channel policies are decentralized in that the security of a channel is relative to each principal rather than a centralized security lattice.

IX. Conclusion

DFLATE offers high-level security abstractions for decentralized, distributed applications that use cryptography and trusted execution environments. These abstractions accurately reflect the strengths and limitations of these mechanisms without exposing low-level implementation details. DFLATE is suitable for formal analysis of decentralized distributed applications and as a core programming model for a general-purpose secure distributed programming language. We have formalized DFLATE's semantics and shown that the type system enforces two variants of noninterference: the stronger variant holds only for confidentiality, reflecting the asymmetry in the security guarantees of the underlying mechanisms.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1565387. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martín Abadi. Access control in a core calculus of dependency. In *11th ACM SIGPLAN Int'l Conf. on Functional Programming*, New York, NY, USA, 2006. ACM.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99. ACM, 1999.

[3] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1. ACM, 2013. doi: 10.1145/2487726.2488368. URL http://doi.acm.org/10.1145/2487726.2488368.

[5] Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *29th IEEE Symp. on Computer Security Foundations (CSF)*, pages 135–147, June 2016. URL http://www.cs.cornell.edu/andru/papers/flac.

[6] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *28th IEEE Symp. on Computer Security Foundations (CSF)*, pages 569–583, July 2015. URL http://www.cs.cornell.edu/andru/papers/flam.

[7] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization: Technical report. Technical Report 1813–40138, Cornell University Computing and Information Science, May 2015. URL http://hdl.handle.net/1813/40138.

[8] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.

[9] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[10] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[11] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *Proceedings of the Sixth International Conference on Information Systems Security*, 2010.

[12] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 1875–1891, October 2017.

[13] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 242–253, July 2006. URL http://www.cs.cornell.edu/andru/papers/robdlm.pdf.

[14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, pages 857–874, 2016.

[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings*

of the 13th Conference on USENIX Security Symposium - Volume 13. USENIX Association, 2004.

[16] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190. ACM, 1988.

[17] Cédric Fournet and Jérémy Planul. Compiling information-flow security to minimal trusted computing bases. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08. ACM, 2008.

[19] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09. ACM, 2009.

[20] Ivan Gazeau, Tom Chothia, and Dominic Duggan. Types for location and data security in cloud environments. In *IEEE Symp. on Computer Security Foundations (CSF)*, August 2017.

[21] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[22] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[23] Shafi Goldwasser and Mihir Bellare. *Lecture Notes on Cryptography*, chapter 10. 2001.

[24] Anitha Gollamudi and Stephen Chong. Automatic enforcement of expressive security policies using enclaves. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2016. ACM.

[25] Anitha Gollamudi, Stephen Chong, and Owen Arden. Technical Report: Information Flow Control for Distributed Trusted Execution Environments. Technical Report TR-01-19, Harvard University, May 2019.

[26] Young Hyun Kwon. Riffle : an efficient communication system with strong anonymity. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2016.

[27] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *IEEE Symp. on Security and Privacy*, pages 359–376. IEEE, 2015.

[28] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *J. Computer Security*, 25(4–5):319–321, May 2017. doi: 10.3233/JCS-0559. URL http://www.cs.cornell.edu/andru/papers/jfabric.

[29] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1. ACM, 2013. doi: 10.1145/2487726.2488368. URL http://doi.acm.org/10.1145/2487726.2488368.

[30] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, June 2006.

[31] Rafael Pass and Abhi Shelat. *A Course in Cryptography*, chapter 7. 3rd edition, 2010.

[32] W. Rafnsson and A. Sabelfeld. Compositional information-flow security for interactive systems. In *2014 IEEE 27th Computer Security Foundations Symposium*, July 2014.

[33] Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing interactive programs. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, 2012.

[34] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symp. on Security and Privacy*, pages 655–670, May 2014.

[35] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 255–269, June 2005.

[36] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011. URL http://doi.acm.org/10.1145/2096148.2034688.

[37] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, 2017. ACM.

[38] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[39] Lucas Waye, Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong. Cryptographically secure information flow control on key-value stores. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*, New York, NY, USA, November 2017. ACM Press.

[40] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3), August 2002.

[41] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008. URL http://dl.acm.org/citation.cfm?id=1387610.

[42] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, June 2012. URL http://www.cs.cornell.edu/andru/papers/pltiming.html.