Abstract—Decentralized applications cannot assume that their security policies will be enforced on untrusted remote nodes. Trusted execution environments (TEEs) combined with cryptographic mechanisms provide a way to securely execute trustworthy code on an untrusted host and exchange confidential and authenticated messages with it. TEEs do not, however, establish the trustworthiness of a TEE application. Consequently, developing secure TEE applications requires specialized expertise and careful auditing.

This paper presents DFLATE, a core security calculus for decentralized, 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 our formal results. DFLATE enforces a strong form of noninterference for confidentiality, but only a weak form for integrity. This asymmetry, not present in other IFC models, 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, remote attestation

I. INTRODUCTION

Many applications rely on security checks in compilers and runtime systems to enforce security policies. In decentralized settings, the effectiveness of such checks is limited: local security checks cannot ensure that a remote host will protect confidential information it receives. Encrypting data can ensure that an untrusted host cannot reveal secret information, but it also prevents the host from performing general computation on it.\(^1\) Lack of trust between entities involved in a distributed application may require data to be hosted separately from the computations that depend on it, hurting performance. Worse, it is possible that no entity is sufficiently trusted to both access the data and compute the result, limiting the kinds of tasks a decentralized application can perform.

Trusted Execution Environments (TEEs) like SGX [22, 4] and Sanctum [10] address some of these limitations with application enclaves. An enclave is a protected user-level process that is strongly isolated from both the OS and other applications by trusted hardware. Remote nodes can verify the integrity of the 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 a malicious host. 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 [21, 31, 28], 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 and distributed applications. DFLATE extends the Flow-Limited Authorization Calculus (FLAC) [5] with distributed execution, communication channels, concurrency, and TEEs. DFLATE’s type system ensures that well-typed programs enforce 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 the example of a simple distributed application, illustrated in Figure 1. In the figure, “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, who 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 (message 4 and 5). Later, she can use these keys to exchange encrypted and signed inputs and outputs with the enclave (messages 6-9).

\(^1\)Fully homomorphic encryption [17] can permit such computation, but at great cost to performance.
without need to repeat the remote attestation protocol.

Omitting or improperly executing any of the above steps can undermine Alice’s security. If the remote attestation is omitted, Alice has no guarantee that the enclave is running in a TEE that protects it from Bob, or that Bob is running the enclave code and not some malicious version of it. Furthermore, Bob might be able to obtain the provided keys from the supposedly-secure channel.

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 to the enclave. Similarly, if the enclave fails to properly encrypt and sign the outputs, Bob may be able to read or modify them.

Fortunately, the security and correctness of the first three messages in Figure 1 is largely independent of the application. This means a relatively simple (but trustworthy) library API or language extension could provide remote attestation capabilities to applications and eliminate programmer errors.

Unfortunately, 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 \( \text{msg} \) from within an enclave over the channel \( \text{ch} \) if \( h \) is true.

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

Because of the TEE and the cryptographic mechanisms, Bob cannot directly access \( h \) or \( \text{msg} \), but he can infer the value of \( h \) based on whether a message is sent. The above code can also be problematic from an integrity perspective. If Bob can influence the value of \( h \), he can suppress the message. Similar code might permit Bob to re-send stale messages or permute the message order.

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

There are two primary challenges to enforcing high-level IFC in a decentralized, distributed setting that employs encryption, digital 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 like Fabric [21] or DStar [31] are insufficiently precise for our purposes. Encryption and digital signatures allow secret or high-integrity messages to be send over untrusted channels. Fabric and DStar, however, cannot express that, for a message sent over a channel controlled by Bob, Alice needs to trust the contents of the message, but Bob can influence whether the message is sent. Instead, they can only express that both Alice and Bob can influence the sent message. In other words, they are too coarse-grained to distinguish the attacker’s influence on control flow from its influence on data flow. This limitation means their enforcement mechanisms cannot be used to determine if code respects the programmer’s intended policy.

Since the primary utility of a TEE is to run computation on a potentially malicious node, this scenario arises in any non-trivial distributed TEE application. Reasoning about protected data flowing through untrusted nodes is fundamental to TEE applications, so properly enforcing their requires finer-grained policy specification and enforcement than these models permit.

The second challenge to enforcing security in our setting is to design high-level abstractions that accurately reflect the guarantees of the TEE in a decentralized setting. As discussed above, TEEs do not inherently protect against leaks. Also, entities may differ on which enclaves they trust, and an enclave may itself be malicious. For example, if the enclave in Figure 1 were malicious, it might leak secrets to Bob or permit him to influence sensitive data. Alice is, at least in theory, able to anticipate such behavior by examining the enclave’s code since she verified the enclave’s measurement. However, detecting malicious and vulnerable code by hand is difficult and time-consuming.

Currently, developers integrate TEEs into their applications using low-level library APIs. Using these libraries correctly may require a different skillset from that needed for the rest of the application. A better approach would be to design high-level programming abstractions that capture the essential parameters and attributes of programming with TEEs without bogging the developer down in 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 due both to the limitations of TEEs, as well as the relative trust entities may place in them. The TEE mechanism, combined with remote attestation, guarantees that specific code is running in a secure address space, but doesn’t hide the existence of messages going into or coming out of the TEE, nor does it guarantee the delivery of these messages. Therefore a secure TEE abstraction must reflect these limitations on communication. Furthermore, since there may be no universally-trusted TEEs or entities, the security abstraction must also allow entities to express their trust in specific TEEs and entities individually.

DFLATE addresses these challenges. DFLATE has suffi-
ciently 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 distributed 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 trusted execution environments. We formalize the security of DFLATE with a proof semantics that establishes noninterference guarantees for well-typed DFLATE programs. Confidentiality noninterference [18] guarantees that an attacker cannot infer secret information from the public outputs of a program. Integrity noninterference guarantees that an attacker cannot influence high-integrity outputs by modifying low-integrity inputs. Integrity is often considered a dual of confidentiality [8], thus most systems that protect confidentiality noninterference also protect integrity noninterference.

Unlike most IFC models, DFLATE provides asymmetric guarantees for confidentiality and integrity. This asymmetry reflects the inherent limitations of the guarantees that TEEs provide. 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 that hold only when messages are not suppressed.

II. MOTIVATING THE DFLATE DESIGN

The DFLATE model 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 that does not assume the availability of cryptographic mechanisms. In this section we motivate three design features of DFLATE that are informed by cryptography and TEEs.

A. Fine-grained policies for secure communication

Consider a scenario where Carol receives a message from Alice 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 within an application, similar to the model used by Fabric and DStar. Sending the 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 with a digital signature, and encrypts the message with Carol’s public key. In this case, Bob can neither learn the contents of A1, nor modify its contents. However, Bob does learn of the existence of the message A1. Furthermore, while Bob cannot modify A1, he does have the power to replace A1 with a previously signed message A2, or could choose to send no message at all.

Most existing IFC abstractions do not distinguish between the two scenarios in Figure 2 and enforce policies conservatively using checks similar to Figure 2(a). This lack of precision has significant consequences for real applications since it effectively ignores the guarantees offered by cryptographic mechanisms. The primary use-case for cryptographic mechanisms is to transmit or receive protected information over an untrusted channel. In order to take full advantage of these mechanisms, we need abstractions for distributed communication security that are capable of distinguishing the ability to observe traffic over a channel and influence or suppress the message sequences from the ability to disclose or modify the messages themselves.

To distinguish these abilities 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 the context in which the channel may be used. A node may receive a message that it is unable to read or modify—this is enforceable 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 is untrusted).

B. Decentralized and distributed trust management

The security abstractions of most IFC models are based on the capabilities of the compiler and runtime to protect secrets from users. Some distributed IFC systems also assume the ability to establish secure channels between nodes, which are implemented with cryptographic protocols by the runtime. DFLATE goes further than previous models by basing its abstractions more directly on the capabilities of cryptographic mechanisms. Doing so enables stronger assumptions about what flows are possible. For the scenario illustrated in Figure 2(a), once Alice has sent message A1 to Bob, she has no control what he does with it. Bob may modify it, send it to other (potentially unauthorized) nodes, or perform computation that depends on it. For the scenario in Figure 2(b) where A1 is encrypted and signed, Alice knows that Bob can only read, propagate, or perform computations on A1 if he possesses the decryption key, and can only (convincingly) modify A1 if he has access to the appropriate signing key. Therefore, even though Alice may not control what actions occur on Bob’s node, she does control some of the information those actions may process based on which keys she gives to Bob.

DFLATE incorporates these observations into its design in a couple of ways. The DFLATE type system enforces a clearance bound which restricts what data may be used in computations on a node. The clearance bound models the ability of a node to digitally sign a value or decrypt an encrypted value. 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 label using a DFLATE program, modeling Bob’s inability to access Alice’s signing key.

DFLATE also permits principals to express trust in code running in a TEE. For each source-level DFLATE computation e, DFLATE defines a unique computation principal t. Code
running in a TEE is subject to clearance bounds on the computation principal rather than the node executing the TEE. Therefore Alice can express trust in an enclave running on Bob’s node, allowing the enclave 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 does not trust Alice or the enclave, Alice cannot use the enclave to leak Bob’s secrets or influence his data.

C. Observability of TEE interactions

Trusted execution environments introduce additional subtlety into information flow control design. TEEs provide guarantees similar to those of a trusted third party, but running an application in a TEE on an untrusted node is not equivalent to running the application on a trusted node.

Consider our previous examples, but where Bob executes the application code in an enclave E as illustrated in Figure 2(c). Although the code executes within an enclave, Alice and Carol must still consider Bob’s ability to observe and manipulate incoming and outgoing messages, as they did in Figure 2(b). The difference here is that the general computation may occur in E whereas in Figure 2(b) Bob could only serve as a conduit.

Most distributed IFC approaches (e.g., [21, 31]) 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 [11]. With TEEs, however, ignoring this ability is not as reasonable: in Figure 2(c), Bob is the only available conduit to E, whereas Alice and Carol might find some other way of communicating that Bob cannot observe. Communicating over an untrusted channel, then, is fundamental to the TEE abstraction. DFLATE ensures that programs capture the ability of a host to mediate communication with its enclave, 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. Interacting with a TEE, however, requires communicating through the host of the TEE. Thus the host observes all remote interaction with the TEE, and a malicious host could fail to relay the messages or manipulate their order.

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 system are represented by FLAM principals. For any principal p, we refer to its confidentiality authority, the authority necessary to learn p’s secrets, as $p^{-}$. The integrity authority of p, or the authority necessary to influence information trusted by p, is written $p^{+}$. More complex principals can be constructed from the set of all primitive principals $N$ using conjunction (∧) and disjunction (∨) operations. Given any two principals p and q, principal $p \land q$ represents the authority of both p and q and $p \lor q$ represents the authority of either p or q.

Given a set of primitive principals $N$ (e.g., Alice, Bob, etc.), and two distinguished principals $\top$ and $\bot$ (respectively the universally trusted and universally untrusted principals), the set of all FLAM principals $P$ is the closure of $N \cup \{\top, \bot\}$ under the operations $\wedge$ and $\vee$, and the projections $\leftarrow$ and $\rightarrow$. $P$ forms a distributive lattice ordered by increasing trust, the “acts for” relation $\triangleright$, with $\top$ and $\bot$ as most and least trusted elements, and with $\wedge$ and $\vee$ as join and meet operations.

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

$$p \subseteq q \iff p^{-} \wedge q^{-} \triangleright q^{-} \wedge p^{+}$$

$$p \sqcap q \iff (p^{-} \wedge q^{-}) \wedge (p^{+} \lor q^{+})$$

$$p \sqcup q \iff (p^{-} \lor q^{-}) \wedge (p^{+} \land q^{+})$$

B. DFLATE syntax and local semantics

The DFLATE language is inspired by the Flow-Limited Authorization Calculus (FLAC) [5]. Like FLAC, DFLATE

\footnote{One way to remember what each arrow means is to think of confidentiality as secrets “coming from” $p$, and integrity as information “accepted by” $p$.}

\footnote{More precisely, the lattice is on the equivalence classes of $\triangleright$.}
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 syntax of DFLATE. The principals (referred to with metavariables $p$, $q$, $\ell$, and $pc$) are similar to those in FLAC, but extended with computation principals from $T$. Metavariable $pl$ refers to primitive principals extended with computation principals and nodes are just places executing the program. Principals are used both to specify the information flow policies on data, but also to represent the authority of the entities in the distributed system. The authority of each DFLATE program is represented either by the principal (from $N$) of the node executing the code, or the computation principal of the enclave (from $T$) if it is executed by a TEE.

Metavariables $v$ and $e$ range over internal values and expressions. DFLATE includes standard syntax for variables $x$, tuples $(e,e)$ and their projections $\text{proj}_j e$, tagged unions $\text{inj}_j e$, and case expressions $\text{case } v \text{ of } \text{inj}_j(x).e | \text{inj}_j(x).e$. Lambda expressions are annotated with a program counter label and a channel environment that specify the information flow context that the function may be applied in and what channel variables must be in scope. The underlined syntax that appears at the end of these syntactic categories are not part of DFLATE’s source syntax, but are used by our formalization to represent intermediate values during evaluation.

Figure 4 presents a selection of DFLATE’s sequential evaluation rules. The complete set of sequential evaluation rules is found in Appendix B. Sequential evaluation judgments have the form $p, D \vdash e \rightarrow e',$ denoting that expression $e$ steps to $e'$ at node $p$ in the delegation context $D$. We abuse the notation a little to simplify the presentation and use metavariable $p$ (instead of $pl$) to represent a node.

The monadic unit term $\eta_{\ell} e$ protects $e$ at the security level $\ell$. This syntax is similar to that used by DCC [3] and FLAC [5], but the DFLATE type system treats monadic terms slightly differently to better model cryptographic protection mechanisms. For example, $\eta_{\text{Alice}} \cdot \text{Alice} \Rightarrow \text{Bob} \cdot v$ can represent a value signed by Alice and encrypted with Bob’s key. Once a value is protected, it may flow to places that would be insecure for the unprotected value to go. Our formalization distinguishes the source-level expression $\eta_{\ell} e$ which must take as input the unprotected value and implies access to cryptographic signing keys, from the runtime value $\eta_{\ell} v$ that has previously been protected. Hence, rule DE-UNITM in Figure 4 evaluates the term $\eta_{\ell} v$ to $\eta_{\ell} v$. Using a protected value is only possible via the monadic bind term. Rule DE-BINDM binds the protected value $v$ to variable $x$ in expression $e$. Binding $v$ from $\eta_{\ell} v$ is analogous to decrypting and authenticating a value protected by the encryption and signature keys represented by label $\ell$.

As in FLAC, delegations are first-class values. The expression assume $(p \succ q)$ introduces a new trust relationship $p \succ q$ that may be assumed when type-checking $e$, but is not present outside the body of the assume. Our formalization tracks delegations that are used during evaluation, so the evaluation rule DE-ASSUME evaluates an assume term to a where term that records the delegation used to authorize the computation, and the rule DE-WHERE maintains a context $D$
of enclosing delegations when evaluating the body of a `where`. This context of delegations, not present in FLAC, is required in DFLATE so enclosing delegations can be properly exported when values are sent over channels. Therefore, prior to sending a value \( v \) over channel \( \nu \), the rule \textbf{DE-Send} exports the context \( D \) by wrapping \( v \) in a `where` term for each delegation in \( D \). These delegations are only used for formal bookkeeping and would be unnecessary in a DFLATE implementation.

Expression \( \text{TEE}^t \ s \) represents an enclave that runs the computation \( s \). The category \( s \) (omitted in Figure 3) is identical to that of \( e \), but does not contain \text{TEE} or \text{spawn} terms. This syntactically prohibits nested or forking \text{TEE} code and reflects similar restrictions in existing TEE architectures. The expression \( s \) is uniquely identified by principal \( t \). The identifier \( t \) is analogous to using a hash of the code \( s \) computed by the TEE to securely identify the code running in an enclave. Assuming that \( t \) uniquely identifies \( s \) is compatible with the trust assumptions of most TEE designs: the code is securely measured and the hash is unique up to collisions, which occur with negligible probability. Rule \textbf{DE-TEE} evaluates the source-level TEE term to the intermediate value \( \text{runTEE}^t \ s \). Note that the \( t \) in \( \text{runTEE}^t \ s \) is only related to the source-level expression \( s \). Additional steps evaluate the expression \( s \), but \( t \) remains fixed.

C. Distributed semantics

A sequential configuration, or a process, \( \langle p, e \rangle \), denotes the expression \( e \) running on node \( p \). A distributed configuration, \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \), is the parallel composition of the processes \( \langle p_i, e_i \rangle \) at each node \( p_i \).

The evaluation rules for distributed configurations are presented in Figure 5. These rules have the form

\[
\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \Rightarrow \langle p_1, e'_1 \rangle \parallel \cdots \parallel \langle p_n, e'_n \rangle
\]

Note that \( m \) and \( n \) can be different if the evaluation step spawns a new process. The premises of the rules in Figure 5 make use of evaluation contexts [12] for sequential evaluation. The term \( E[e] \) denotes an expression containing a subexpression \( e \), which is reducible according to an evaluation context grammar presented in Appendix B. We omit this grammar here since it is relatively standard.

Rule \textbf{PAR-Step} connects the distributed rules with the sequential ones. It states that the distributed configuration takes a step whenever one of the processes takes a step. There is no change in the number of processes, and only one process may take a sequential step for each application of \textbf{PAR-Step}. The premise \( p_i, \emptyset \vdash e_i \rightarrow e'_i \) says that the process assumes no top-level delegations. This premise might seem restrictive at first glance, but it merely requires that delegation contexts are recorded explicitly by nested `where` terms. At the top level, the delegation context is empty. The other evaluation rules in Figure 5 state their premises in terms of evaluation contexts (e.g., \( E[e] \)) allowing the expression being evaluated to occur in a non-empty delegation context.

Processes use channels to communicate. DFLATE channels are not first-class, and the same channel endpoint cannot be used by multiple processes. This restriction allows us to specify the endpoints of a channel statically using the channel type (metavariabale \( c \)). Channel endpoints are unidirectional: a send endpoint allowing \( p \) to send values of type \( \tau \) to \( q \) is represented with the type \text{chan}_{pq} \ p \rightarrow \ q \ \tau. \) An endpoint allowing \( p \) to receive values from \( q \) has type \text{chan}_{qp} \ p \rightarrow \ q \ \tau. \) The \( \tau \) annotation specifies the information flow context in which channels may send or receive messages.

The \textbf{spawn} term creates a new process, connecting it to the parent with a send and receive channels. Rule \textbf{PAR-Spawn} evaluates spawn terms. For example: the following term, evaluated by Alice, creates a process \( \langle \text{bob}, e \rangle \) and two channels. Alice uses \( ch_1 \) to send messages of type \( \tau_1 \) to Bob, and Bob uses \( ch_2 \) to send messages of type \( \tau_2 \) to Alice.

\[
\text{spawn} (\text{bob}, ch_1[pc_1;\tau_1], \ ch_2[pc_2;\tau_2]). \ e \ e_\kappa
\]

All instances of channel variables \( ch_1 \) and \( ch_2 \) in \( e \) and \( e_\kappa \) are substituted with new channel values \( v_1 \) and \( v_2 \). Evaluation then proceeds in parallel on Bob’s node with \( e \), and on Alice’s node with \( e_\kappa \). In addition to remote node, spawn takes a \( pl \) parameter which is used to specify the other endpoint of the channel if both parent and child process are executing on the same node. This is useful when spawning a TEE, \( pl \) can be the TEE identifier.

The term \textbf{send} \( ch \ e \) then \( e_\kappa \) sends the result of evaluating \( e \) over the channel \( ch \), then proceeds with the continuation \( e_\kappa \). The term \textbf{recv} \( ch \ as \ x \ in \ e_\kappa \) receives a message over channel \( ch \) and maps it to variable \( x \) in the continuation \( e_\kappa \). DFLATE channels are synchronous: sends block until they are ready to be received and receives block until a message is available. Thus, the evaluation rule \textbf{PAR-Send-Recv} only applies when both a send and a receive for the same channel are ready to be reduced.

Suppose node \( p_i \) is sending a value \( v \) over the channel \( \nu \) and node \( p_j \) is ready to receive it. Observe that the result of stepping \textbf{send} \( ch \ v \) then \( e \) with rule \textbf{PAR-Send-Recv} results in the expression \( e \Box \) and \textbf{recv} \( ch \ as \ x \ in \ e' \) becomes \( e' \langle x \mapsto v \rangle \Box \). The \( \Box \) term is used to denote the end of the process: \( e \Box \) reduces to an output value \( v' \Box \). While \( v' \Box \) is a valid expression, output values are not in the syntactic category of internal values \( v \) in Figure 3. Therefore, terms like \( (\lambda(x:\tau))[pc, \Theta], e \langle x \mapsto v \rangle \Box \) cannot be reduced with \textbf{DE-App} since the rule only applies when the argument evaluates to an internal value in the category \( v \). Since no evaluation rule applies, the program is stuck. Our type system ensures this never occurs for well-typed programs. This semantic device merely ensures that the continuation of sending or receiving process is a subterm of the send or receive. It does not affect the expressiveness of the language, only the structure. For example,

\[
(\lambda(x:\tau)[pc, \Theta], e) \ e' \langle y \mapsto v \rangle
\]

under an alternate semantics that evaluated this term to

\[
(\lambda(x:\tau)[pc, \Theta], e) \ e'[y \mapsto v]
\]
can be rewritten in DFLATE as
\[
\text{recv } ch \text{ as } y \text{ in } (\lambda(x: \tau)[pc, \Theta], e) e' \]
which evaluates to the same expression. This design helps simplify type system constraints that track the influence of sent and received messages on control flow.

The DFLATE sequential semantics is deterministic, but it is possible for multiple distributed rules to be applicable in a given configuration. For example, multiple processes may have send/recv pairs ready to be reduced, and the semantics does not specify which to reduce first. However, because of the carefully managed channel scopes, there can never be two sends or receives on the same channel since each endpoint is in scope for only one process. This invariant eliminates races and ensures all sequences of rule applications eventually result in the same final configuration.

IV. THE DFLATE SECURITY TYPE SYSTEM

The DFLATE type system statically enforces information flow control policies on the data processed by DFLATE programs. The grammar presented in Figure 3 in Section III-B splits types into two kinds: output types, which have the form \( \tau \odot \), and internal types of the form \( \tau \). Metavariable \( \tau \) is used when either an output or an internal type is permitted. The type system ensures that expressions with output types cannot be used as internal values.

Well-typed programs do not always reduce to values, however. They may also become stuck waiting for communication: either a ready send with no corresponding recv or vice versa. To distinguish these stuck programs from those that the type system is designed to prevent, we refer to well-typed programs that become stuck waiting for communication as blocked programs.

DFLATE supports System-F-style types like products \( \tau_1 \times \tau_2 \), sums \( \tau_1 + \tau_2 \), functions \( \tau \rightarrow \pi \rightarrow \tau \), and type functions \( \forall X[pc, \Theta], \pi \). Delegation types \( (p \rightarrow q) \) are singleton types: each delegation type \( (p \rightarrow q) \) is inhabited by a single value \( (p \rightarrow q) \), similar to how \( () \) is the single value of type unit.

The type \( \ell \) says \( \tau \) protects an expression of type \( \tau \) at level \( \ell \). The mechanism used to protect the value is abstract, but...
authority as $q$ in delegation context $\Pi$. Recall that $\sqsubseteq$ is defined in terms of $\triangleright$ so we may also derive judgments of the form $\Pi \vDash p \sqsubseteq q$ using the same rules.

Both of these forms are actually syntactic sugar for $\Pi; \perp \land \top; \perp \land \top \vdash p \triangleright q$ in the more general judgment form $\Pi; pc; \ell \vDash p \triangleright q$. The rules for deriving acts-for judgments, presented in Appendix B, are based on a fragment of the Flow-Limited Authorization Model logic [6]. Each delegation in the FLAM delegation context is labeled with a confidentiality and integrity policy. The logic establishes rules for how those delegations may be used to derive authorization judgments. The query label, $pc$, protects the distributed authorization query by limiting communication to nodes that protect the confidentiality of $pc$. The result label, $\ell$, protects the confidentiality and integrity of the delegation context by limiting the delegations used to derive the judgment to those that flow to $\ell$. For example, the rule $R$-ASSUME below specifies when $(p \triangleright q | \ell')$ may be used to derive $\Pi; pc; \ell \vDash p \triangleright q$. The voice operator $\nabla(q)$ defines the integrity necessary to control the delegations of, or speaks for, principal $q$.

\[
\begin{align*}
(p \triangleright q | \ell') & \in \Pi \\
\Pi; pc \sqsubseteq \ell'; \ell' & \vDash \ell \land \nabla(q) \\
\Pi; pc & \vDash \nabla(p \\
\Pi; pc; \ell & \vDash \nabla(p) \triangleright \nabla(q) \\
\Pi; pc & \vDash p \triangleright q \\
\end{align*}
\]

This rule connects acts-for judgments to protected types. If label $\ell$ flows to $\ell'$, then the type $\ell' \text{ says } \tau$ protects label $\ell$. Singleton types like $\text{unit}$ and $(p \triangleright q)$ protect any label since the type itself encodes the value: observing the runtime value carries no information. However, the type $\tau_1 \triangleright \tau_2$ 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. The complete set of protection rules is shown in Figure 2 in Appendix B.

DFLATE's type protection judgment is more restrictive than that of FLAC's, as well as the protection rules in the Dependency Core Calculus [3, 2], which FLAC's are based on. The restrictiveness comes from the omission of three rules. One rule, $\text{DP-LBL1}$, permits a label to be protected by the inner type of a $\text{says}$ type if the outer type does not protect it.

\[
\begin{align*}
\Pi & \vdash \ell \sqsubseteq \tau \\
\Pi & \vdash \ell \leq \tau
\end{align*}
\]

This rule is not compatible with the cryptographic mechanisms DFLATE seeks to model. The problem is that it makes nested $\text{says}$ types commutative in the sense that $p \text{ says } q \text{ says } \tau$ protects the same labels as $q \text{ says } p \text{ says } \tau$. Commutativity undermines the expressiveness of integrity policies since a value of type $\tau$ 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 the order of encryption is not reflected statically. A value of type $p \text{ says } q \text{ says } \tau$, may have been encrypted first with $q$'s public key, then $p$'s, or first with $p$'s then $q$'s. Successfully decrypting the value requires determining the correct order.

The other rules omitted by DFLATE are $\text{DP-FUN}$ and $\text{DP-TF}$UN. These rules say that function types protect a label if the $pc$ and return type do.

\[
\begin{align*}
\Pi & \vdash \ell \sqsubseteq pc' \\
\Pi & \vdash \ell \leq \tau_2 \\
\Pi & \vdash \ell \leq \tau
\end{align*}
\]

These rules are reasonable in a local setting where an attacker cannot observe raw lambda values. Thus, only the information that is observable from applying these functions is relevant. In a distributed setting, however, lambdas are sent over channels to potentially malicious hosts, making it unreasonable to assume they can't learn anything from the encoding of the lambda term.

Figure 6 presents the core typing rules for sequential terms adapted from FLAC. Every DFLATE typing rule contains a
clearance premise $\Pi \vdash p \geq p$ that requires the place $p$ act for the $pc$ at all times. This premise ensures a place cannot observe or use data exceeding its authority.

Rule DT-LAM says a lambda expression is well-typed if the body of the lambda is well-typed under the given $pc'$ and $\Theta'$. Function and type application rules DT-APP allow a function to be applied only if the $pc$ at the application flows to the $pc'$ of the function. The premise $\Theta \vdash_{\text{dom}(\Theta')} \Theta'$ ensures that the functions do not capture channels that are not in the scope of the caller. This prevents channel variable escaping through closures during communication.

Rule DT-UNITM requires that a well-typed expression $e$ created at in a context with program counter label $pc$ is protected at level $\ell$ such that $pc$ flows to $\ell$. This premise, along with the clearance premise $\Pi \vdash p \geq p$, prevents a node from protecting a value with an integrity that exceeds its authority. Suppose Alice wants to protect a value at Bob’s integrity with $\eta_B v$. By clearance, $\Pi \vdash Alice \geq p$, so to type check by DT-UNITM, it must be the case that $\Pi \vdash pc \subseteq Bob$. It follows that $\Pi \vdash Alice^\rightarrow \geq Bob^\rightarrow$, indicating she has access to Bob’s signing key.

Rule DT-SEALED permits protected values to flow through hosts that would not have had the authority to create them. For instance, even if Alice does not trust Bob, the sealed value $\eta_{Alice} v$ is well-typed at Bob as long as $v$ is well-typed. Sealed values reflect the security guarantees of cryptographic protection mechanisms: that attackers cannot distinguish ciphertexts or forge signatures.

Rule DT-BINDM says that if an expression $e$ protected at level $\ell$ is bound to $x$ and used in $e'$, then the type of $e'$ must protect $pc \sqcup \ell$ and $e'$ must type check at a more restrictive label $pc \sqcup \ell$. This raised $pc$ label, along with the clearance premises, prevent a node from reading protected values that exceed its authority. Suppose Bob wants to bind a value $\eta_{Alice} v$ to $x$ in an expression $e'$. For $e'$ to type check at label $pc \sqcup \ell$, it must be the case (by a simple induction on the typing rules) that $\Pi \vdash Bob \geq pc \sqcup Alice$. It follows that $\Pi \vdash Bob^{-\rightarrow} \geq Alice^{-\rightarrow}$, indicating Bob has access to Alice’s decryption key.

Rule DT-ASSUME ensures extensions of the delegation context for the body $e'$ are secure. Notice the similarity between the premises of DT-ASSUME and R-ASSUME. These premises guarantee all delegations added to the context are usable via the R-ASSUME rule.

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 $p$ has access to the send endpoint of $ch_1$ and the receive endpoint of $ch_2$, node $p'$ is other endpoint. If $p'$ isn’t same as the current node, then $p'$ is same as node $q$. This is useful for specifying the endpoints when enclaves are spawned. Processes spawned on remote nodes can only have $q$ as valid endpoint. Spawned processes inherit the delegation context from the spawning process. The premise $\Pi \vdash pc' \leq \tau'$ ensures the output of the process protects the output of the spawning process. Thus if $pc'$ is confidential (or untrusted) the output can only be used in confidential (or untrusted) contexts.

Rule DT-SEND requires that channel $ch$ is the send endpoint and that the expression has the correct type. The premise $\Pi \vdash pc \sqsubseteq pc_{ch}$ requires that the current $pc$ flows to the channel $pc$, ensuring $pc_{ch}$ is an upper bound on the confidentiality and integrity of the control flow up to the send. After the send is evaluated, execution proceeds with the continuation $e'$ which must type check at a $pc'$ at least as restrictive as the channel $pc$. This ensures that information revealed by successfully sending a message is protected by $pc'$. In addition to the usual clearance premise, DT-SEND also has a channel clearance premise $\Pi \vdash p \geq pc_{ch}$ that ensures $p$ has sufficient authority to use the channel. Finally, notice that whether or not $e'$ has an output type $\check{\square}$, the send term has an output type, ensuring that the continuation-passing discipline is maintained.

Rule DT-RECEIVE is very similar DT-SEND. Here, the channel direction is reversed, and the received value is mapped to variable $x$ in the continuation.

Rule DT-TEE says that the closed expression $e$ executes within enclave $t$ with the integrity of $t$ and confidentiality $pc^\rightarrow$. Only channels connecting $p$ and $t$ with a channel $pc$ that protects $pc$ are in scope for the expression $e$. This restriction ensures two properties. First, all messages into and out of a TEE pass through the TEE’s host. Second, the IFC context at $p$ at the other end of the channel not only protects the TEE’s $pc$.
using Bob as an intermediate, protecting the output with label $\tau'$. Now consider line 3, where the function $f$ is applied and the result protected at $a \cap c$. DT-RECEIVE requires that the $pc$ of the continuation is at least $a \cap b \cup c$, and DT-UNITM requires that this $pc$ is protected by the label: $\Pi \vdash_a b \cup c \subseteq a \cap c$.

A. Examples revisited

Figure 8 presents DFLC code for the scenarios illustrated in Figure 2. Each of these programs applies a function $f$ to a protected value from Alice ($a$) and outputs the result to Carol ($c$) using Bob as an intermediate, protecting the output with label $a \cap c$, implying both Alice and Carol can read it and both trust its integrity. Despite their functional similarity, however, each program requires very different trust relationships between Alice, Bob, and Carol.

Figure 8(a) presents an implementation of Figure 2(a), where no cryptographic mechanisms are used. For this program to type check under some delegation context $\Pi$, it must be the case that Alice trusts Bob and Carol completely. In other words, $\Pi \vdash b^\triangleright \equiv a^\triangleright \wedge \Pi \vdash c^\triangleright \equiv a^\triangleright$ for $\pi = \triangleright$ and $\pi = \leftarrow$. Furthermore, Carol must trust Alice and Bob with her integrity, $\Pi \vdash a^\triangleright \equiv c^\triangleright$ and $\Pi \vdash b^\triangleright \equiv c^\triangleright$.

To see why, first consider the send line 7. For this term to type check, it must be the case that $\Pi \vdash a \subseteq a \cap b \cup c$ since, by DT-BINDM, the $pc$ at this point is at least as restrictive as the label $a$ on the protected value $\eta_a v$, and by DT-SEND, this $pc$ must flow to the channel $pc$, $a \cap b \cup c$. If $\Pi \vdash a \subseteq a \cap b \cup c$ holds, recalling the definitions of $\subseteq$, $\cap$, and $\neg$ from Section III-A, it follows that $\Pi \vdash b^\triangleright \equiv a^\triangleright$ and $\Pi \vdash c^\triangleright \equiv a^\triangleright$. Note this would still be the case if a different channel $pc$ for $ch_b$ were chosen. If the channel $pc$ was $a^\triangleright \wedge (a \vee b)^\triangleright$ instead of $a \vee b \wedge c$, the clearance requirement for the recv at line 5 would require $\Pi \vdash b \triangleright \equiv a^\triangleright \wedge (a \vee b)^\triangleright$, which implies $\Pi \vdash b^\triangleright \equiv a^\triangleright$. A similar argument for the recv at line 3 would imply $\Pi \vdash c^\triangleright \equiv a^\triangleright$.

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 in order to send the value to Carol. When Carol binds the value in order to apply the function $f$, the $pc$ at the box in line 4 is at $(a \vee b \wedge c) \wedge a$ which is equal to $a^\triangleright \wedge (a \vee b \wedge c)^\triangleright$ by the definition of $\sqcup$ and lattic absorption. Therefore, DT-UNITM requires that $\Pi \vdash a^\triangleright \wedge (a \vee b \wedge c)^\triangleright \subseteq a \cap c$, which implies that $\Pi \vdash c^\triangleright \equiv a^\triangleright$, as well as the same integrity relationships as implied by Figure 8(a).

The 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 the TEE, Alice no longer needs to delegate integrity to Carol since Carol has no influence.

3 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.
delegate integrity to on the computation. Since the enclave protects the result of the computation. Since the enclave protects the result

\[ \Pi \vdash t^\rightarrow \models a^\rightarrow \quad \Pi \vdash t^\rightarrow \models c^\rightarrow \quad \Pi \models a^\rightarrow \models c^\rightarrow \]

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

\[ \Pi_{\text{tee}} \models b^\rightarrow \models t^\rightarrow \quad \Pi_{\text{tee}} \models c^\rightarrow \models a^\rightarrow \]

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.

V. NONINTERFERENCE

Noninterference requires that secret inputs cannot affect public outputs, and that untrusted inputs cannot affect trusted outputs. Not all DFLATE programs are noninterfering. This is by design: many programs release information but are still considered secure. Password checkers, for instance, release some information to an attacker about the password even when the attacker’s guess is wrong. Nevertheless, specifying the static requirements for a noninterfering DFLATE program and proving noninterference is enforced is a useful evaluation of DFLATE’s security guarantees and limitations.

In DFLATE each process has its own set of beliefs about delegations; these are reflected as delegation contexts. Since two process may have different delegation contexts, enforcing noninterference is challenging. First of all, not all nodes necessarily agree on which flows are permitted. Second, since trust assumptions may change during evaluation via assume terms, nodes that once agreed on a set of trust relationships may diverge. Note that diverging trust assumptions are not necessarily a security problem—authorizing the declassification or endorsement of information is a common operation in secure distributed applications—but this aspect of DFLATE means that our definitions of noninterference require some care.

We have proven theorems for three variants of noninterference in DFLATE. The first two variants are presented as a single statement, Theorem 1, with two cases: one for confidentiality and one for integrity. This theorem characterizes noninterference in terms of the inputs and outputs of DFLATE programs. Our third and stronger variant, Theorem 2 found in Appendix A due to space limitations, states noninterference in terms of the program traces observable by the attacker, but only holds for confidentiality.

Common to each theorem is a label \( H \) representing restricted information. For confidentiality, inputs labeled \( H^\rightarrow \) should not flow to outputs visible to an attacker. For integrity, an attacker input labeled \( H^\rightarrow \) should not flow to high-integrity outputs. To define noninterference, we require that all nodes initially agree on what labels \( H^\rightarrow \) may flow to. For example, we can instantiate \( H \) as Bob to state that Bob’s secrets cannot flow to an attacker when running programs on Alice, Bob, and Carol. For our theorems to apply, Alice, Bob, and Carol merely need to agree on what labels \( Bob^\rightarrow \) may flow to. If they disagree, the program is not necessarily insecure—Bob may have authorized Alice to release some secrets—but since such programs may have flows that violate noninterference, our theorems do not apply. Generalizations of noninterference (e.g., robust declassification [29] and nonmalleable information flow [9]) define security for interfering programs; exploring
these generalizations in DFLATE is left for future work.

We also require that nodes maintain the above agreement on $H^\pi$ flows by not delegating $H^\pi$'s authority via $\text{assume}$ terms. Here again, such delegations are not necessarily insecure, but are beyond the scope of noninterfering programs. A draconian way of eliminating delegations of $H^\pi$'s authority would be to forbid occurrences of $\text{assume}$ altogether. This is unnecessarily conservative. Instead, we specify the absence of such downgrades in a process using a static approximation, called a delegation approximation, of all delegations used by that process. We leave abstract the approximation algorithm and instead define the properties of a valid delegation approximation in Definition A.1 in Appendix A. Informally, a delegation approximation $\Pi_k$ is a valid approximation of process $\langle p_k, e_k \rangle$ if for any delegation $\langle p \Rightarrow q \rangle$ that occurs on $p_k$ during the evaluation of $e_k$ (including those received in where terms from other nodes), we have $\langle p \Rightarrow q \rangle \in \Pi_k$. A trivial analysis producing a valid delegation approximate is to include the delegations from all assume terms in the system. We sketch a more precise analysis in our technical report [1].

Theorem 1 states our confidentiality and integrity noninterference results on inputs and outputs of programs. For well-typed processes $\langle p_k, e_k \rangle$ (for $k \in \{1, \ldots, m\}$), Theorem 1 distinguishes two nodes: $p_i$ and $p_j$. For noninterference to hold, the output on node $p_j$ must be same for two runs where different values are mapped to $x$ on node $p_i$.

**Theorem 1 (DFLATE Input/Output Noninterference).** Let $e_k$ for $k \in \{1, \ldots, m\}$ be DFLATE programs such that $\Pi_k; \Gamma_k; \Theta_k; p_k; c_k$ is $\text{assume}$ $e_k$ if $\forall x \in \mathbb{H}$, a flow from $H^\pi$ to $\ell'$ implies that the program demonstrates why a stronger integrity result is impossible. The program meets Conditions 1-3 of Theorem 1, but accepts inputs whose runs have unequal final configurations.

Condition 3 defines the authority of the attacker and is split into two cases, one for confidentiality ($\pi=\rightarrow$) and one for integrity ($\pi=\leftarrow$). For confidentiality, the output node $p_j$ is considered to be the attacker; it may read information at $\ell^\pi$, but not at $H^\pi$. For integrity, the output node is not the attacker. Instead, $p_j$ receives a high-integrity output labeled $\ell^\pi$ that should not have been influenced by the attacker’s input labeled $H^\pi$.

For confidentiality ($\pi=\rightarrow$), evaluating the distributed program with any two secret inputs of the proper type produces a final configuration where $e'_j = e'_{j2}$. Note that these outputs may be output values or expressions since the final configuration of a program could contained blocked send or recv expressions. Whether blocked or not, the expressions $\eta_{j1}$ and $\eta_{j2}$ must be equal. This result implies that a secret input on $p_i$ cannot affect whether the process on $p_j$ remains blocked. If it could, then one run would result in final configuration that terminates with a value $v$, while the other would remain blocked, contradicting the theorem since the final configuration contains different terms at $p_j$.

For integrity ($\pi=\leftarrow$), the result is not as general. Here, evaluating the distributed program with any two untrusted inputs results in equal outputs only if both runs terminate with values. Unlike the confidentiality case, final configurations in the integrity case may contain unequal expressions on $p_j$ if one or more of the expressions are blocked. The asymmetry in these results derives from the power of the integrity attacker to suppress or reorder high-integrity messages it processes. A confidentiality attacker does not posses a similar ability to infer information about the secret messages it processes.

To further illustrate this asymmetry, the following well-typed program demonstrates why a stronger integrity result is impossible. The program meets Conditions 1-3 of Theorem 1, but accepts inputs whose runs have unequal final configurations.

\[
\begin{align*}
\text{bind} & \; u = \eta_{y \leftarrow y} \quad \text{in} \\
\langle p_i, \text{in}_{j_1}(x), \text{send} \; ch \; v \; \text{then} \; () \text{ in}_{j_2}(x), \text{recv} \; ch' \; \text{as} \; y \; \text{in} \; y \rangle & = (q, \text{recv} \; ch \; \text{as} \; x \; \text{in} \; x) \\
\end{align*}
\]

On the left, the attacker node binds low-integrity input $i$ 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 engaged in a communication with the attacker node. It invokes a function only after it receives some input from the attacker node. For inputs $i = \text{in}_{j_1}(x)$ the expression on $q$ evaluates to $v$, but for $i = \text{in}_{j_2}(x)$ becomes blocked. 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 the 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.

VI. RELATED WORK

A. Enclaves and Information Flow

Gollamudi and Chong [20] use enclaves to enforce infor-
mation flow policies against a spectrum of attackers in a non-distributed setting. Specifically, they focus on enforcing confidentiality using enclaves against a powerful low-level active attacker that can inject arbitrary code into non-enclave parts of the system. 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.

Fournet et al. [15] compile a high-level sequential program into a distributed program, preserving its security properties using cryptographic techniques. The target program running on different threads communicates using shared variables. Fournet and Planul [13] extend the compiler to use Trusted Platform Modules (TPM) and remote attestation to minimize the TCB. The target program is at least as secure as the source program.

Subramanyan et al. [27] provide a formal foundation for the remote execution of enclaves by proving the security of the remote execution on Trusted Abstract Platform (TAP), an idealized abstraction of enclave platforms parameterized by an adversary model. Specifically they prove that two remote enclave executions emit traces that are observationally equivalent as long as the attacker provides same inputs in both the executions. By contrast, DFLATE proves end-to-end semantic guarantees (noninterference) of the distributed applications using enclaves.

B. Communication Channels and Cryptography

Zdancewic et al. [30] use language-based techniques to securely partition a program into subprograms that communicate to simulate the original program. The resulting distributed program prevents read channels, which occur when a host leaks information by making a remote read request for a secret reason. DFLATE’s channel pc annotations protect against similar leaks. If a message is sent or received on a channel, our typing rules ensure neither host learns unauthorized information.

Fournet and Rezk [14] present a security type system to enforce the correct usage of cryptographic primitives for controlled declassification and endorsement.

Wysteria [26] is a functional 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. It uses two modes of operation: secure and parallel. It is permitted to access data belonging to other principals in secure mode but is restricted in parallel mode. This is analogous to the bind operation in DFLATE which requires sufficient authority to use data protected at a security level. However, because Wysteria models communication implicitly through variable binding, it does not detect insecure flows that may arise due to explicit communication.

Gazeau et al. [16] allow creation and sharing of principals in a controlled way without requiring any global PKI infrastructure. Their security guarantee relies on the correct enforcement of access control, namely, honest nodes do not provide access to attacker nodes. However, they only enforce confidentiality (but not integrity) of the client data running in a cloud server.

Fabric [21] and DStar [31] use static and dynamic mechanisms to enforce IFC for distributed programs. They use cryptographic protocols to establish secure channels between nodes, but do not allow high-integrity or secret data to flow through untrusted hosts.

CLIO [28] uses cryptography to enforce security in IFC programs that use a remote key-value store. Runtime values are encrypted as they leave the language runtime and stored on an untrusted store. CLIO does not model general communication between nodes, but it does model the attacker’s ability to suppress entries in the key-value store by overwriting them. Dependencies on messages that are suppressible by an attacker are tracked using availability policies, which specify the availability of data, in addition to confidentiality and integrity policies. Extending DFLATE with availability would further improve DFLATE’s precision in distinguishing the ability to modify a value from the ability to suppress or deny it.

Our channel design is similar in principle to those in Rafnsson, Hedin, and Sabelfeld’s work on interactive programs [25], which also uses two policies to distinguish the presence of a message from the contents of a message by defining distinguished channel operations for each (centralized) security level. DFLATE channel policies are decentralized in that the security of a channel is relative to each principal rather than a centralized security lattice. Furthermore, the continuation-passing-style send and recv commands allow for a somewhat simpler type system.

SX10 [23] extends X10 concurrent programming language and enforces noninterference using a combination of static analysis and security type system that reasons about information flows due to internal timing channels. It does not use any cryptographic protection mechanisms or enclaves. SX10 enforces observational determinism even in the presence of nondeterminism of events arising due to scheduling. We could extend DFLATE with similar techniques to enforce observational determinism.

VII. Conclusion

DFLATE offers high-level security abstractions for decentralized, distributed applications that employ cryptographic mechanisms and trusted execution environments. These abstractions accurately reflect the strengths and limitations of these mechanisms without exposing low-level implementation details at the source level. DFLATE provides a suitable basis for formal analysis of decentralized distributed applications, and as a core programming model for a general-purpose secure distributed programming language. We have formalized a semantics for DFLATE and demonstrated that the DFLATE type system enforces confidentiality noninterference on program traces, but enforces integrity noninterference only for programs that terminate with a value. This mismatch in guarantees validates our design: attackers are able to suppress or reorder messages, so high-integrity messages relayed by low-integrity hosts may allow attackers to influence program behavior. However, we demonstrate that when messages are delivered, attackers cannot influence the contents of high-integrity messages.

REFERENCES


[27] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas
That is, \( p \oveq_{\hat{\Pi}} q \) if they are protected at a level higher than \( p \). We now strengthen the attacker’s observation model to \( \Pi \). Using the observation function, we define the trace of a process that is observed by the attacker.

**Definition A.2 (Trace of a Process).** Let

\[
S = \langle p_1, e_1^1 \rangle \parallel \cdots \parallel \langle p_i, e_i^i \rangle \parallel \cdots \parallel \langle p_m, e_m^i \rangle
\]

such that \( i = v \) where \( \langle p_1, q_1 \rangle \ldots \langle p_k, q_k \rangle \). \( p_j \) is the delegation approximation of \( \Pi_j \), then

\[
\forall \ell, \ell', \hat{\Pi}_{S(j)} \not\subseteq \ell \implies \Pi_j, \langle p_1, q_1 \rangle, \ldots, \langle p_k, q_k \rangle \not\subseteq \ell' \implies \Pi_j, \langle p_1, q_1 \rangle, \ldots, \langle p_k, q_k \rangle \not\subseteq \ell'
\]

A. Noninterference for Stronger Confidentiality Attacker

We now strengthen the attacker’s observation model to include the entire trace of the execution on a node. We formally define the trace observed by the attacker using an observation function, \( O(e, \Pi, p) \), that emits parts of the expression \( e \) that can be observed by a principal \( p \). It erases sealed terms if they are protected at a level higher than \( p \). It also erases the contents of a \( \text{TEE} \) if it is more confidential and has less integrity than \( p \). Intuitively, this models the inability of an attacker in distinguishing ciphertexts.

We show \( O(e, \Pi, p) \) for interesting cases; for the remaining terms, the function is homomorphically applied to the subexpressions of \( e \).

\[
\begin{array}{c|c}
\text{e} & \text{O(e, \Pi, p)} \\
\hline
\eta e' & \eta O(e', \Pi, p) \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\eta e' & \eta \bullet \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\eta e' & \eta O(e', \Pi, p) \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\eta e' & \eta \bullet \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\text{TEE} \cdot e' & \text{TEE} \cdot O(e', \Pi, p) \quad \text{if } \Pi \not\subseteq t \subseteq p \\
\text{TEE} \cdot e' & \text{TEE} \cdot \bullet \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\text{runTEE} \cdot e' & \text{runTEE} \cdot O(e', \Pi, p) \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\text{runTEE} \cdot e' & \text{runTEE} \cdot \bullet \quad \text{if } \Pi \not\subseteq \ell \subseteq p \\
\end{array}
\]

We lift the observation function to a process executing on node \( q \) (with respect to a principal \( p \)) as below.

\[
O((q, e), \Pi, p) = (q, O(e, \Pi, p))
\]

Using the observation function, we define the trace of a process that is observed by the attacker.

**Definition A.3 (Trace Equivalence)**

Two traces, \( T_{r_1} \) and \( T_{r_2} \), are equivalent, denoted as \( T_{r_1} \approx T_{r_2} \), if they are equal after removing the duplicates. That is

\[
T_{r_1} \approx T_{r_2} \iff U(T_{r_1}) = U(T_{r_2})
\]

Using these formal definitions we state the noninterference against this stronger attacker in Theorem 2.

**Theorem 2 (Noninterference for Stronger Confidentiality Attacker).** Assume the conditions stated in Theorem 1. For all \( v_i \) and \( v_j \) such that, \( \Pi; \Gamma; \Theta; p; \mathcal{P} \vdash_d v_i; H \) says \( \tau \),
Theorem 1. If $p_j$ observes traces $s$ of the process executing on the node $v_j$ then the traces of the process executing on the node $v_j$ are equivalent. That is, $\tau(\hat{p_j}(s)) = \tau(p_j(s))$.

Fig. 9: Missing DFLATE syntax

$E ::= \epsilon \mid E \in E \mid (E,e) \mid \langle E,E \rangle \mid \eta E \mid \text{bind } x = E \in e \mid E \tau \\
\text{case } e \text{ of } \eta_j(x), s \mid \text{tag } x = s \mid \text{assume } s \in E \\
\text{send } ch \; s \; \text{then } s \; \text{recv } ch \; \text{then } s \; \text{where } s \; \text{e;}$

Fig. 10: Evaluation Contexts (E). Define $W[e]$ as substituting hole (in $W$) with expression $e$.

$\Pi;\Gamma;\Theta; p_j; pc_i \vdash_d v_r : H^- \text{ says } r \text{ and } \Pi_j \not\vdash_d p_j : H^-$.

Fig. 11: FLAC Robust Assumption

$\Pi;\Gamma;\Theta; p_j; pc_i \vdash_d e : \tau \quad \Pi; pc \vdash pc \quad \Pi; p \vdash pc$

Fig. 12: Type protection levels

$\Pi;\Gamma;\Theta; p_j; pc_i \vdash_d e : \tau \quad \Pi; pc \vdash pc \quad \Pi; p \vdash pc$

The statement of Theorem 2 is similar to that of Theorem 1 but says that the traces observed by the attacker node $p_j$ for two executions that only differ in their secret inputs are equal up to stuttering. Theorem 2 implies the confidentiality result of Theorem 1.

APPENDIX B
Fig. 14: DFLATE typing rules for standard terms

Fig. 15: DFLATE evaluation step for a single process
Fig. 16: DFLATE evaluation step for case terms

Fig. 17: Propagation of where terms

Fig. 18: DFLATE typing rules for spawn and communication primitives