Information Flow Control for Distributed Trusted Execution Environments

The Harvard community has made this article openly available. **Please share** how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:40614411">http://nrs.harvard.edu/urn-3:HUL.InstRepos:40614411</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Information Flow Control for Distributed Trusted Execution Environments

Anitha Gollamudi
Stephen Chong
and
Owen Arden

TR-01-19

Computer Science Group
Harvard University
Cambridge, Massachusetts
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 distrustful, 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 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.

Trusted Execution Environments (TEEs) such as SGX [30, 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 [29, 43, 41], 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 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 as from within an enclave over channel if \( h \) is true.

\[
\text{if } h \text{ then send ch (enc (sign msg)) } \text{ else ()}
\]

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 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 [29] and DStar [43]) 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 [23] 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 [38] 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 e. 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 e. 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., [29, 43]) 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 [16]. 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

2Using LIO-style clearance bounds as a proxy for access to cryptographic keys was first introduced in CLIO [41].
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 $p$'s secrets, and integrity authority, the authority necessary to influence information trusted by $p$. Authority projections of the form $p^\pi$ where $\pi \in \{\to, \leftarrow\}$ allow us to represent the partial authority of a principal. For example, the principal $p^\to$ denotes a principal with only the confidentiality authority of $p$, and $p^\leftarrow$ denotes a principal with only the integrity authority of $p$. The combined authority of principals $p$ and $q$ is represented by the conjunction $p \land q$, and the selective authority of principals $p$ and $q$ (i.e., the individual authority of either $p$ or $q$) is represented by the disjunction $p \lor q$. 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 $\land$, $\lor$, $\leftarrow$, and $\to$ over the set of primitive principals $\mathcal{N}$, extended with $\top$ and $\bot$. Principals in this set are related by a preorder $\succ$, the “acts for” relation, which orders principals by increasing trust. The equivalence classes of $\succ$ form a distributive lattice with $\top$ and $\bot$ as most and least trusted elements, and with $\land$ and $\lor$ as join and meet operations.

The trust ordering $\succ$ 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 flows-to relation also forms a distributive lattice with $\bot \to \top$ (public and trusted) as the least restrictive element, and $\top \to \bot$ (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:

$$
p \subseteq q \iff p^\to \succ q^\to \text{ and } q^\to \succ p^\leftarrow
$$

$$
p \land q \iff (p^\to \land q^\to) \land (p^\leftarrow \lor q^\leftarrow)
$$

$$
p \lor q \iff (p^\to \lor q^\to) \land (p^\leftarrow \land q^\leftarrow)
$$

Every principal $p$ is equivalent (under the trust ordering) to a principal in normal form, $q^\to \land r^\leftarrow$, i.e., the conjunction of a confidentiality authority and an integrity authority. The voice

$$
n \in \mathcal{N} \text{ (primitive principals)} \quad x \in \mathcal{V} \text{ (variables)}
$$

$$
ch \in \mathcal{V}_c \text{ (channel variables)} \quad \nu \in \mathcal{C} \text{ (channel values)}
$$

$$
p, \ell, pc ::= n \mid \bot \mid p^- \mid p^+ \mid p \land p \mid p \lor p
$$

$$
v ::= (x) \mid \langle v, v \rangle \mid (p \succ p) \mid \text{inj}_i v \mid \lambda(x: \tau)[pc, \Theta, P, \Pi].e
$$

$$
e ::= \tau \mid v \mid e e \mid \langle e, e \rangle \mid \eta_i e \mid e \hat{\tau} \mid \text{proj}_i e
$$

$$
e ::= \text{inj}_i e \mid \text{case of inj}_i x(e) \mid \text{inj}_j x(e)
$$

$$
e ::= \text{bind } x = e \text{ in } e \mid \text{assume } e \text{ in } e
$$

$$
e ::= \text{send } ch e \text{ then } e \mid \text{recv } ch \text{ as } x \text{ in } e
$$

$$
e ::= \text{TEE}^\ell s \mid \text{spawn } @ (pl.ch[pc; \hat{\tau}], ch[pc; \hat{\tau}]).e \text{ then } e
$$

$$
\tau ::= \hat{\tau} \mid \hat{\top} \mid \hat{\bot} \mid \hat{\bot} \mid \hat{\top} \mid \hat{\top} \land \hat{\bot} \mid \hat{\top} \lor \hat{\bot}
$$

$$
\tau ::= \hat{\top} \mid \hat{\bot} \mid \hat{\top} \land \hat{\bot} \mid \hat{\top} \lor \hat{\bot}
$$

$$
\tau ::= \hat{\top} \mid \hat{\bot} \mid \hat{\top} \land \hat{\bot} \mid \hat{\top} \lor \hat{\bot}
$$

$$
e ::= \text{chan}_{p \leftarrow q} pc \hat{\tau} \mid \text{chan}_{p \to q} pc \hat{\tau}
$$

Fig. 3: DFLATE syntax

of a principal $p$, $\nabla(p)$, is the integrity authority necessary to act on the behalf of the principal. Formally, if $q^\to \land r^\leftarrow$ is the normal form of $p$, then $\nabla(p) = \nabla(q^\to \land r^\leftarrow) = q^\to \land r^\leftarrow$.

**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 $p$, $q$, $\ell$, 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
where and decision to add it have sufficiently high integrity. We use term to add this delegation, i.e., that the delegation to add and the

del
}\left[\text{DE-BINDM}\right] \quad pl, D \vdash \text{bind } x = \eta_{\ell} v \text{ in } e \rightarrow e\{v/x\}

\left[\text{DE-ASSUME}\right] \quad pl, D \vdash \text{assume } v \text{ in } e \rightarrow e \text{ where } v

\left[\text{DE-TEE}\right] \quad pl, D \vdash \text{TEE}^f s \rightarrow \text{runTEE}^f s

\left[\text{DE-WHERE}\right] \quad \frac{pl, D \cdot v \vdash e \rightarrow e'}{pl, D \vdash e \text{ where } v \rightarrow e' \text{ where } v}

\left[\text{DE-SEND}\right] \quad \frac{pl, D \vdash \text{send } \nu v \text{ then } e \rightarrow \text{send } \nu v' \text{ then } e}{v' = \text{export}_\text{del}(D, v) \quad \text{noTEE}(v)}

The operational semantics for DFLATE uses two judgments: one for sequential semantics and one for distributed semantics (see Section III-C). Sequential semantic judgment \( pl, D \vdash e \rightarrow e' \) 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 \( \langle p \triangleright q \rangle \), indicating that principal \( q \) has delegated its authority to principal \( p \). We assume that there is a well-known initial delegation sequence \( D_{\text{init}} \). Expression \( \text{assume } \langle p \triangleright q \rangle \text{ in } e \) adds delegation \( \langle p \triangleright q \rangle \) 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 \( e \text{ where } v \) to record that delegation \( v \) holds for evaluation of \( e \). Rules \text{DE-ASSUME} and \text{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 \( \ell \). This syntax is similar to that used by DCC [2] and FLAC [5], but the

5 All inference rules are given in Appendix A.

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 [32] and existentially unforgeable signature schemes [24]. Intuitively \( \eta_{\ell} e \) evaluates \( e \) to a value, and then encrypts and signs the value with keys appropriate for \( \ell \) to protect it. Protected value \( \eta_{\ell} v \) represents the encrypted and signed value \( v \) (see rule \text{DE-UNITM}). For example, \( \eta_{\text{Alice}}\tau - \text{ABob}^\tau v \) represents value \( v \) signed by \text{Alice} and encrypted with \text{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 \( \text{bind } x = \eta_{\ell} v \text{ in } e \), which binds \( v \) to variable \( x \) in \( e \) (rule \text{DE-BINDM}). This is analogous to decrypting and verifying the signature of protected value \( \eta_{\ell} v \).

Expression \( \text{TEE}^f s \) represents a TEE that will execute computation \( s \). Syntactically 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 \text{DE-TEE} evaluates the source-level TEE term to the intermediate value \( \text{runTEE}^f s \). Note that the \( t \) in \( \text{runTEE}^f s \) 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 \( D \) to range over distributed configurations.

Rules for distributed configurations are presented in Figure 5 and have the form \( D \Rightarrow D' \). Some of the rules use evaluation contexts [17] for sequential evaluation: \( E[e] \) is an expression

Fig. 5: DFLATE distributed semantics
with subexpression $e$ that can be reduced. Evaluation contexts are standard and defined in Appendix A.

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 delegation 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 block 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 $v$ with a process that is receiving on $v$. 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.

Suppose node $n_i$ is sending a value $v$ over the channel $v$ and node $n_j$ is ready to receive it. Observe that the result of stepping $send \ ch \ v$ then $e$ with rule PAR-SEND-RECV results in the expression $e_1[x \mapsto v] ; \square$. The $; \square$ term is used to denote the end of the process: $e[x \mapsto v] \square$ reduces to an output value $v' \square$. While $v' \square$ is a valid expression, output values are not in the syntactic category of internal values $v$ in Figure 3. Our type system ensures that well-typed programs never output values as internal values. This restriction 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.

This design helps simplify type constraints that track the influence of sent and received messages on control flow.

Term spawn @ (pl, ch1[pc1; \dot{\tau}_1], ch2[pc2; \dot{\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 $pc_1$, $\dot{\tau}_1$, $pc_2$, and $\dot{\tau}_2$ restrict how channels may be used. Place pl indicates the intended endpoint of the newly created channel: if the newly spawned process is on the same host as the parent, then pl may indicate a computation principal $t$ that is the intended endpoint. This enables a pattern where a parent process can spawn a child process that executes a TEE, and the parent process can communicate with the TEE.

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 $A$ 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 $A$ is the malicious principal and $p$ is a "good principal" (i.e., a principal that doesn’t trust $A$), then $A$ 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 $\eta_\ell \ v$ and $\eta_\ell \ v'$ (which represent values $v$ and $v'$ encrypted and signed by principal $\ell$) if the attacker does not have the authority to decrypt $\eta_\ell \ v$ and $\eta_\ell \ v'$. 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., [44, 34, 28]) 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 [15] 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 $(p \gg q)$ is inhabited by a single value $(p \gg q)$. Monadic
type \( \ell \) says \( \tau \) protects an expression of type \( \tau \) at level \( \ell \); it is the type of values such as \( \eta_v \) \( v \) (where \( v \) has type \( \tau \)).

The grammar presented in Figure 3 splits types into two kinds: output types, which have the form \( \tau \rhd \Box \), and internal types of the form \( \tau \). Output types represent expressions that cannot be consumed. For example, they cannot be passed as arguments to functions. 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. This distinction in types simplifies parts of the proof but does not limit the expressiveness of the language, just the structure of it.

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

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.

Typing judgment \( \Gamma; \Theta; pl; pc \vdash e: \tau \) indicates that expression \( e \) has type \( \tau \). Delegation context \( \Gamma \) 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 \( \Gamma \) maps variables to types. Variables may have only internal types, so all mappings in \( \Gamma \) have the form \( x : \tau \). Channel variable scope is maintained using the channel environment \( \Theta \). 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 \( \Box \)) 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 \( \Pi \vdash p \stackrel{\geq}{\rightarrow} q \) and require that \( p \) has at least as much authority as \( q \) in delegation context \( \Pi \). (Alternatively, that assuming the delegations in \( \Pi \), \( q \) trusts \( p \)). Recall that \( \equiv \) is defined in terms of \( \geq \) so using the same rules we may also derive judgments of the form \( \Pi \vdash p \equiv q \).

Intuitively, if \( \Pi \vdash p \equiv q \) information labeled with \( p \) can flow to information labeled \( q \), since (given the delegations in \( \Pi \)) 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 FLAC 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.) Both of these forms are actually syntactic sugar for \( \Pi; \downarrow^\rightarrow \land T^\rightarrow ; \downarrow^\rightarrow \land T^\rightarrow \vdash p \geq q \) in the more general FLAC judgment form \( \Pi; pc; \ell \vdash p \geq q \). The rules for deriving acts-for judgments, presented in Appendix A, are based on a fragment of the Flow-Limited Authorization Model logic [6]. Each delegation in the FLAM delegation context \( \Pi \) 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 \( \ell \).

In DFLATE, authority is verified statically by the DFLATE compiler independently of any sensitive data. Therefore the query label, \( pc \), which protects the information in the acts-for query \( p \geq q \), is considered public and trusted, or \( \downarrow^\rightarrow \land T^\rightarrow \). Similarly, the result of that query is based only on trustworthy information the compiler itself has provided and processed, so the result is considered public and trusted too. 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 this technical report, 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 \( \tau \) protects information labeled with \( \ell \). Intuitively, it means that the type system ensures that any information gained by using a value of type \( \tau \) will have a security level at least as restrictive as \( \ell \). The rules for deriving type protection judgments are based on a subset of FLAC’s rules. The primary rule is DP-LBL1:

\[
\boxed{\text{[DP-LBL1]}}
\begin{align*}
\Pi & \vdash \ell \equiv \ell' \\
\Pi & \vdash \ell \leq \ell' \text{ says } \tau
\end{align*}
\]

This rule connects acts-for judgments to protected types. If \( \ell \) flows to \( \ell' \), then the type \( \ell' \) says \( \tau \) protects level \( \ell \). Singleton types like \text{unit} and \( (p \geq q) \) protect any level since the type itself encodes the value: observing the runtime value carries no information. However, the type \( \tau_1 \leftrightarrow \tau_2 \) 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 in Figure 16 in Appendix A.

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.

\[
\boxed{\text{[DP-LBL1]}}
\begin{align*}
\Pi & \vdash \ell \leq \tau \\
\Pi & \vdash \ell \leq \ell' \text{ says } \tau
\end{align*}
\]
This rule is not compatible with the cryptographic mechanisms DFLATE seeks to model: it makes nested \( \text{says} \) types commutative in the sense that \( p \text{ says } q \text{ says } \tau \) protects the same levels 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 encryption order 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 distinguishing the correct order.

The other two rules we omit are DP-FUN and DP-TFUN, which state that abstractions protect a level if the \( pc \) and return type do.

\[
\begin{align*}
\text{[DP-FUN]} & \\
\Pi \vdash \ell & \subseteq pc' & \Pi \vdash \ell & \leq \tau_2 & \Pi \vdash \ell & \leq \tau \\
\Pi \vdash \ell & \leq \tau_1 & \frac{pc', \Theta, \Pi}{\tau_2}
\end{align*}
\]

These rules are reasonable in a setting where an attacker cannot observe raw lambda values, and information can be gained from a function value only by applying it. 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 \( \Pi \vdash pl \gg pc \) 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 \( \tau_1 \xrightarrow{pc, \Theta, \Pi} \tau_2 \), level \( pc \) is the \emph{latent effect} of the function (i.e., a lower-bound on the observable side-effects when the function is invoked), \( \Theta \) is the channel environment the function expects, \( \Pi \) is the delegation context the function expects, and \( P \) are the places at which the function may 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 \( pl \in P \). 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 \( P \), 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 \( P \) 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.

(The type abstraction and application is similar.)

Expression \( \eta \in e \) will evaluate \( e \) and then protect the result at level \( \ell \) (in implementation, by signing and encrypting it). It has type \( \ell \text{ says } \bar{\tau} \) (rule DT-UNITM) provided that \( e \) is well-typed and the program counter level \( pc \) flows to \( \ell \). 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 (\( \Pi \vdash pl \gg pc \)) ensures that place \( pl \) has appropriate integrity to sign the value. Suppose Alice wants to protect a value at Bob’s integrity by evaluating \( \eta_{\text{Bob}} v \). To type check, it must be the case that \( \Pi \vdash pc \subseteq \text{Bob} \). By clearance we have \( \Pi \vdash \text{Alice} \gg pc \), and thus \( \Pi \vdash \text{Alice}^{\rightarrow} \gg \text{Bob}^{\rightarrow} \), 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-SEATED 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 \( \eta_{\text{Alice}} v \) is 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 \( \text{bind } x = e \text{ in } e' \), expression \( e \) evaluates to a protected value \( \eta_{\text{Alice}} v \), and \( x \) is bound to \( v \) in \( e' \). Rule DT-BINDM requires that the type of \( e' \) must protect \( pc \sqcup \ell \) and \( e' \) must type check at a more restrictive level \( pc \sqcup \ell \). Clearance (for expression \( e' \)) ensures that \( pc \sqcup \ell \) does not exceed the place’s authority: the place is trusted to compute on data protected at level \( \ell \). For example, if Bob evaluates \( \text{bind } x = \eta_{\text{Alice}} v \text{ in } e' \), for \( e' \) to type check it must be the case that \( \Pi \vdash \text{Bob} \gg pc \sqcup \text{Alice} \).

It follows that \( \Pi \vdash \text{Bob}^{\rightarrow} \gg \text{Alice}^{\rightarrow} \), 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 \( \Pi \vdash \nabla(q^{\rightarrow}) \gg \nabla(r^{\rightarrow}) \) ensures \emph{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. Premise \( \Pi \vdash pc' \leq \ell' \) 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.

DT-SEND requires that channel \( ch \) is the \emph{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 \( \tau \) (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 \( \Pi \vdash pl \gg pc_{ch} \) that...
ensures p has sufficient authority to use the channel. Finally, notice that whether or not e has an output type τ, the send term has an output type, ensuring that the continuation-passing discipline is maintained. Rule DT-TEE is similar to DT-SEND. Here, the channel direction is reversed, and the received value is mapped to variable x in the continuation. The type of channel messages τ and the channel program counter level pc.ch allow the sender and receiver to coordinate on the security and contents of messages sent over the channel.

Expression \( \text{TEE}^f e \) 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 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 pc^+\( t \). Rule DT-TEE differs from all other typing rules in that there is no relation between the integrity of 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 pc^+\( t \).

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 ∩ c, implying Alice and Carol can read it and both trust its 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 \( \Pi \), it must be the case that Alice trusts Bob and Carol completely. That is, \( \Pi \vdash b \gg a \) and \( a \vdash c \gg a \). Furthermore, Carol must trust Alice and Bob with her integrity, \( \Pi \vdash a^c = e^c \) and \( a \gg b^c \gg e^c \). To see why, first consider the send in line 7. For this term to type check, it must be the case that \( \Pi \vdash a \subseteq a \gg b \gg c \). Since, by DT-BINDM, the pc at this point is at least as restrictive as the level a on the protected value \( \eta_a \), and by DT-SEND, this pc must flow to the channel pc, \( a \gg b \gg c \). If \( \Pi \vdash a \subseteq a \gg b \gg c \)
elements reflects the same trust relations. Values are protected when the message. The DFLATE program shown in Figure 8(b)
wishes with her confidentiality and integrity in Figure 2(b), Alice signs and encrypts her message to Carol, but does not require her to delegate her
trust Bob's integrity since he can influence the computation by assuming Bob's influence by assuming $b^\rightarrow \succ t^\rightarrow$.

Now consider line 3, where function $f$ is applied and the result protected at $a \cap c$. DT-RECEIVE requires that the $pc$ of the computation is at least $a \lor b \lor c$, and DT-UNITM requires that this $pc$ be protected by $a \cap c$, i.e., $\Pi \models a \lor b \lor c \subseteq a \cap c$.

For this to hold, it must be the case that $\Pi \models a \lor b \lor c \succ a$ and $\Pi \models a \lor b \lor c \succ c$. These judgments imply all the following:

\[
\begin{align*}
\Pi \models b^\rightarrow & \succ a^\rightarrow \\
\Pi \models c^\rightarrow & \succ a^\rightarrow \\
\Pi \models a^\rightarrow & \succ c^\rightarrow \\
\Pi \models b^\rightarrow & \succ c^\rightarrow 
\end{align*}
\]

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 $(a \lor b \lor c) \cap a$ which is equal to $a^\rightarrow \land (a \lor b \lor c)^{\uparrow}$ by definition of $\sqcup$ and lattice absorption. Therefore, DT-UNITM requires that $\Pi \models a^\rightarrow \land (a \lor b \lor c)^{\uparrow} \subseteq a \cap c$, which implies

\[
\Pi \models a^\rightarrow \land (a \lor b \lor c)^{\uparrow} \subseteq a \cap c
\]

\[\text{Note this would still be the case if a different channel } pc \text{ for } ch_b \text{ were chosen. If the channel } pc \text{ was } a^\rightarrow \land (a \lor b)^{\uparrow} \text{ instead of } a \lor b \lor c, \text{ the clearance requirement for the } \text{recv } \text{ at line 5 would require } \Pi \models b^\rightarrow \succ a^\rightarrow \land (a \lor b)^{\uparrow}, \text{ which implies } \Pi \models b^\rightarrow \succ a^\rightarrow.\]

\[\text{The absorption laws } (a \lor b) \land a = a \text{ and } (a \land b) \lor a = a \text{ for all lattice elements } a \text{ and } b \text{ are algebraic properties of all lattices.}\]

that $\Pi \models c^\rightarrow \succ a^\rightarrow$ as well as the same integrity relationships 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 $t^\rightarrow$ (Alice’s integrity is also required for the assume at line 7).

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

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

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

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 \( n \in N \) (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.

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 \( Alice \land Bob \) and \( Alice \lor Bob \lor Carol \).

Computation principals \( t \in T \) 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 \( \pi_p v \) represents value \( v \) encrypted and signed by \( p \).\(^9\) The value \( \pi_p v \) is implemented by first encrypting \( v \) (using the appropriate key for \( p^- \)), and then signing the result (using the appropriate key for \( p^+ \)). Conversely, evaluation of \( \text{bind } x = \pi_p v \) 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 \( (p \gg q) \) 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 [16] or Riffle [27].

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\(^e\) \( e \), 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.

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 \( \Pi_{\text{init}} \vdash pl \gg \nabla(p) \) (where \( \Pi_{\text{init}} \) 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 \( \Pi_{\text{init}} \vdash t \gg \nabla(p) \). 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 \( \Pi_{\text{init}} \vdash n \gg \nabla(p) \), 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,
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.\(^\text{12}\) 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, 25, 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 [31]. 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 \(\ell\) where it is not the case that \(H\) can flow to \(\ell\). 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 \(\ell\)).

Since we are stating noninterference, we are concerned only with executions where there is no downgrading from \(H\) (or above) to \(\ell\). 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 \(D\), we have a delegation approximation \(\Pi_k\) that over-approximates delegations that the process may make during execution.\(^\text{13}\) See Appendix B for a formal definition.

Suppose all processes in \(D = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle\) are well-typed, and the \(i\)th process takes an input value protected by \(H\) (i.e., it has a free variable of type \(H\) says \(\tau\)) and the \(j\)th process produces a value of type \(\ell\) says \(\text{bool} \square\). Moreover, suppose the delegation approximations for the processes ensure that they never downgrade from \(H\) (i.e., \(\Pi_k\) permits the same flows from \(H\) as \(\Pi_{n_k}\)). If we have two executions of \(D\) where the input to the \(i\)th process is replaced with different values, then the result of the \(j\)th process will be the same. We state this formally. Proofs of theorems are in Appendix C.

**Theorem 1** (Batch-Job Noninterference). Let \(H\) and \(\ell\) be security levels such that \(\Pi_{n_k} \nleq H \subseteq \ell\). Let \(D = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle\) such that for all \(k \in 1..m\) we have

\[
\Pi_{n_k}; \Gamma_k; \Theta_k; n_k; pc_k \vdash e_k : \tau_k,
\]

where \(\tau_k = H\) says \(\text{bool} \square\) and \(x : H\) says \(\tau \in \Gamma_i\).

Assume that no process downgrades from \(H\), i.e., \(\forall k \in 1..m. \forall \ell'. \Pi_{n_k} \models H \subseteq \ell' \iff \Pi_k \models H \subseteq \ell'\). For all \(v_1\) and \(v_2\), and all \(z \in \{1, 2\}\) such that

\[
\Pi_{\Gamma_i}; \Theta_i; n_i; pc_i \vdash v_z : H\] says \(\tau_z\),

let \(D_2 = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_z, e_z(v_z/x) \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle\). If \(D_1 \Rightarrow^* \langle n_1, e'_1 \rangle \parallel \cdots \parallel \langle n_z, v'_z \rangle \parallel \cdots \parallel \langle n_m, e'_m \rangle\) and \(D_2 \Rightarrow^* \langle n_1, e'_1 \rangle \parallel \cdots \parallel \langle n_z, v''_z \rangle \parallel \cdots \parallel \langle n_m, e''_m \rangle\) then \(v'_z = v''_z\).

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 \(\Pi_{n_k} \models n \nleq \nabla(p)\). Thus, the attacker cannot observe the contents of protected value \(\pi_{\ell'} v\) if \(\Pi_{n_k} \nleq n \nleq \nabla(\ell')\), nor see the contents of a TEE for computation principal \(\ell\) if \(\Pi_{n_k} \nleq n \nleq \nabla(\ell')\).

The formal definitions of the process trace of node \(n\) and equivalence of process traces are provided in Appendix B. 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 \(\Pi_{n_k} \nleq n_j \nleq H\rightarrow\). Let

\(^{12}\)Weakening confidentiality is called declassification [37]; 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.
\[ \mathcal{D} = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle \] 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 \( x : H^- \) says \( \tau \in \Gamma_i \).

Assume that no process downgrades from \( H^- \), i.e., \( \forall k \in 1..m \), \( \forall \ell. \Pi_{init} \vdash H^- \subset \ell \equiv \Pi^p_{init} \vdash H^- \subset \ell \). For all \( v_1 \) and \( v_2 \), and all \( z \in \{1, 2\} \) such that
\[
\Pi; \Gamma_i; \Theta_i; n_i; pc_i \vdash v_z : H^- \text{ says } \tau,
\]
let \( D_z = \langle n_1, e_1 \rangle \parallel \cdots \parallel \langle n_i, e_i(v_z/x) \rangle \parallel \cdots \parallel \langle n_m, e_m \rangle \). Then for all executions \( D_1 \Longrightarrow^* D'_1 \) and \( D_2 \Longrightarrow^* D'_2 \) the process traces of node \( n \) are equivalent.

An equivalent of Theorem 2 does not hold for integrity. To see why, consider the following well-typed program that is running on node \( n_j \) and outputs \( \eta \text{true} \) for all inputs \( x \).
\[
\lambda_{\text{true}} \left( \text{bind } u \equiv x \in (\lambda_{\text{unit}} u) \right)
\]
where
\[
\lambda_{\text{true}} = \lambda_{(\_; ()):}^{(}\ell, \emptyset, P, \Pi_{\lambda}, \eta \text{true} \]
and
\[
\lambda_{\text{unit}} = \lambda_{(\_; \text{bool}):}^{(}\ell, \emptyset, P, \Pi_{\lambda}, ()}
\]
Let the input \( x \) be controlled by an adversary (say \( H^- \)). Then a high integrity observer (say \( \ell^- \)) can observe the following traces when the given inputs \( H^- \text{ says true} \) and \( H^- \text{ says false} \).

<table>
<thead>
<tr>
<th>Trace on ( S_1 )</th>
<th>Trace on ( S_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( \langle n_j, \lambda_{\text{true}} \lambda_{\text{unit true}} \rangle )</td>
<td>( \langle n_j, \lambda_{\text{true}} \lambda_{\text{unit false}} \rangle )</td>
</tr>
<tr>
<td>2. ( \langle n_j, \lambda_{\text{true}} () \rangle )</td>
<td>( \langle n_j, \lambda_{\text{true}} () \rangle )</td>
</tr>
<tr>
<td>3. ( \langle n_j, \eta \text{true} \rangle )</td>
<td>( \langle n_j, \eta \text{true} \rangle )</td>
</tr>
</tbody>
</table>

Though the final output is same in both the executions, the traces differ in line 1. This asymmetry is due to the message suppression ability of the attacker. Consider the following program on nodes \( n_1 \) and \( n_2 \).

\[
\langle n_1 \parallel \langle \text{case } \text{u of } \text{inj}_1(z), \text{send } \ell \text{ then } () \rangle \parallel \langle \text{inj}_2(z), \text{recv } \ell \text{ as } y \text{ in } y \rangle \rangle \parallel \langle n_2, \text{recv } \ell \text{ as } y \text{ in } y \rangle
\]

On the left, the attacker node binds low-integrity input \( x \) (of type \( H^- \text{ says bool} \)) to \( u \) and branches on the value, sending on channel \( \ell \) in one branch and receiving on channel \( \ell \) 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 \( x = \eta \text{true inj}_1(z) \), process \( n_2 \) terminates, but for \( x = \eta \text{true inj}_2(z) \) 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.

\[ \text{VIII. Related Work} \]

\[ \text{A. Enclaves and Information Flow} \]

Gollamudi and Chong [25] 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. [20] 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 [18] 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 \texttt{spawn}, \texttt{send}, and \texttt{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. [39] 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.

\[ \text{B. Communication Channels and Cryptography} \]

Zdancewic et al. [42] securely partition a program into subprograms that communicate to simulate the original program. The resulting distributed program prevents \texttt{read channels}, which leak information when a remote read request occurs for a secret reason. DFLATE’s channel \texttt{pc} annotations protect against similar leaks.

Fournet and Rezk [19] 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 [36] is a language for writing secure multiparty computation protocols. \texttt{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. [21] 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 [29] and DStar [43] 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. [35], 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. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-16-1-0351. 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 or the United States Air Force.

REFERENCES


n ∈ N (primitive principals) t ∈ T (enclaves) x ∈ V (variables)
ch ∈ VC (channel variables)
ν ∈ C (channel values)
p, ℓ, pc ::= n | T | ⊥ | p* | p~/p | p ∧ p | p ∨ p | t
τ ::= ⊤ | ⊥
τ ::= p @ p | unit | τ + τ | τ × τ
| τ ⊢ ϕ, τ | X | ∀X [pc, Θ, Π]. τ
:e ::= () | (v, ν) | (p > p) | injv, e | λ(x: τ)[pc, Θ, Π]. e
| ΛX[pc, Θ, Π, E]. e | ηe v | v where e
:e ::= x | v | e e | (e, e) | ηe e | e e | injv e | case e = injv1(x). e | injv2(x). e
| bind x = e in e | assume x e in e
| send ch e then e | recv ch as x in e
| TEE e s | spawn (npl, ch[pc, τ], ch[pc, e]) then e
| runTEE e s | send ch v then e | e where e = e; e
:s ::= x | v | s s | inj v e | case e of injv1(x). s | injv2(x). e
| bind x = s in s | assume s in s
| send ch s then s | recv ch as x in s
| send ch v then s | s where s = e; e

Fig. 9: Complete DFLATE syntax

E ::= [ ] | E e | v E | (E, e) | (v, E) | projv, E | injv, E
| ηe E | bind x = E in e | E τ
| assume E in e | send ν E then e
| case E of injv1(x). e | injv2(x). e
| where e
W ::= E | W e | v W | (W, e) | projv, W | injv, W
| ηe W | bind x = W in e | W τ
| assume W in e | send ν W then e
| case W of injv1(x). e | injv2(x). e
| where v | runTEE s E

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

APPENDIX A

We present the full syntax (Figure 9), evaluation contexts (Figure 10), evaluation rules (Figures 11, 12, 13, 14), robust assumptions (Figure 15), type protection rules (Figure 16), typing rules (Figures 17, 18, 19).

APPENDIX B

Definition B.1 (Delegation Approximation Contains All Delegations). Let S be the well-typed distributed system

Πinit; Γ1; Θ1; p1; pc1 ⊢ e1: τ1 | ⋯ | Πinit; Γm; Θm; pm; pm ⊢ em: τm | ⋯ |

The delegation approximation "\( \hat{\Pi}_n \)" of the process \( \{p_i, e_i\} \) is the union of the set of all delegations in the initial delegation context II and those that are assumed during the evaluation of the process \( \{p_i, e_i\} \).
Fig. 12: DFLATE evaluation step for case terms

\[
\begin{align*}
[\text{DE-CASE}'] & \quad \text{pl}, D \vdash e \to e' \\
[\text{DE-CASE1}] & \quad \text{pl}, D \vdash \text{(case (inj}_1 v) \text{ of inj}_1(x), e_1 \mid \text{inj}_2(x), e_2 \to \text{case e'} \text{ of inj}_1(x), e_1 \mid \text{inj}_2(x), e_2} \\
[\text{DE-CASE2}] & \quad \text{pl}, D \vdash \text{(case (inj}_1 v) \text{ of inj}_1(x), e_1 \mid \text{inj}_2(x), e_2 \to e_1[x/v]} \\
[\text{DW-APP}] & \quad \text{pl}, D \vdash (v \text{ where } (a \succ b)) v' \to (v v') \text{ where } (a \succ b) \\
[\text{DW-UNPAIR}] & \quad \text{pl}, D \vdash \text{proj}_1 ((v_1, v_2) \text{ where } (a \succ b)) \to \text{proj}_1 (v_1, v_2) \text{ where } (a \succ b) \\
[\text{DW-CASE}] & \quad \text{pl}, D \vdash \text{(case } v \text{ where } (a \succ b) \text{ of inj}_1(x), e_1 \mid \text{inj}_2(x), e_2 \to \text{case } v \text{ of inj}_1(x), e_1 \mid \text{inj}_2(x), e_2 \text{ where } (a \succ b)} \\
[\text{DW-UNITM}] & \quad \text{pl}, D \vdash \eta_v v \text{ where } (a \succ b) \to \eta_v v \text{ where } (a \succ b) \\
[\text{DW-BINDM}] & \quad \text{pl}, D \vdash \text{bind } x = (v \text{ where } (a \succ b)) \in e \to (\text{bind } x = v \text{ in } e) \text{ where } (a \succ b) \\
[\text{DW-ASSUME}] & \quad \text{pl}, D \vdash \text{assume } (v \text{ where } (a \succ b)) \in e \to (\text{assume } v \text{ in } e) \text{ where } (a \succ b) \\
[\text{DW-TEEG}] & \quad \text{pl}, D \vdash \text{runTEE}^i (v \text{ where } (a \succ b)) \to (\text{runTEE}^i v) \text{ where } (a \succ b) \\
[\text{DW-HALT}] & \quad \text{pl}, D \vdash \text{halt } (v \text{ where } (a \succ b)) ; \square \to \square (v \square \text{ where } (a \succ b))
\end{align*}
\]

Fig. 13: Propagation of where terms

\[
\begin{align*}
[\text{PAR-STEP}] & \quad \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_m, e_m \rangle \Rightarrow \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_m, e_m \rangle \\
& \quad \text{where } e_i = W [\text{spawn } @ \ell_1] [p_1 ; c_1 ; \tau_1], [c_2 [p_2 ; \tau_2]]. \text{ e then } e' \] \\
[\text{PAR-SPAWN}] & \quad \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_m, e_m \rangle \Rightarrow \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_m, e_m \rangle | q_i, m+1 \\
& \quad \text{where } q_i, m+1 = e' \text{ e then } e' \text{ e then } e' \text{ channel i = 1 to i = m} \\
[\text{PAR-SEND-RECV}] & \quad \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_j, e_j \rangle | \cdots | \langle p_m, e_m \rangle \Rightarrow \langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_j, e_j \rangle | \cdots | \langle p_m, e_m \rangle \\
\end{align*}
\]

Fig. 14: DFLATE distributed semantics

\[
\begin{align*}
[\text{R-STATIC}] & \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q \\
[\text{R-ASSUME}] & \quad \text{pl}, \ell \vdash p \quad \text{pl}, \ell \vdash p \quad \text{pl}, \ell \vdash p \\
& \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q_1 \quad \text{pl}, p \vdash q_2 \\
& \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q \quad \text{pl}, p \vdash q_1 \quad \text{pl}, p \vdash q_2 \\
& \quad \text{pl}, p \vdash q_1 \quad \text{pl}, p \vdash p \vdash q_1 \quad \text{pl}, p \vdash p \vdash q_2 \\
& \quad \text{pl}, p \vdash q_1 \quad \text{pl}, p \vdash q_2 \\
[\text{R-DISJ}] & \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \\
& \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \\
& \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \quad \text{pl}, p \vdash p \\
[\text{R-TRANS}] & \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \\
[\text{R-WEAKEN}] & \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \quad \text{pl}, p \vdash p \vdash r \\
\end{align*}
\]

Fig. 15: FLAC Robust Assumption

\[
\begin{align*}
[\text{DP-UNIT}] & \quad \Pi \vdash \ell \leq \text{unit} \\
[\text{DP-DEL}] & \quad \Pi \vdash \ell \leq (a \succ b) \\
[\text{DP-HALT}] & \quad \Pi \vdash \ell \leq \tau_1 \\
[\text{DP-PAR}] & \quad \Pi \vdash \ell \leq \tau_2 \\
[\text{DP-LAL}] & \quad \Pi \vdash \ell \leq \ell' \quad \text{ says } \tau \\
\end{align*}
\]

Fig. 16: Type protection levels

**Proposition 1.** Let \( S \) be the well-typed distributed system from Definition B.1. If \( \Pi^D \) is the delegation approximation of the process \( \langle p_i, e_i \rangle \), then the following hold.

1) If

\[
\langle p_1, e_1 \rangle | \cdots | \langle p_i, e_i \rangle | \cdots | \langle p_m, e_m \rangle \Rightarrow \tau_i
\]

2) \( \forall \ell, \ell'. \Pi \vdash \ell \leq \ell' \Rightarrow \Pi^D \vdash \ell \leq \ell' \)

3) \( \forall \ell, \ell'. \Pi^D \not\vdash \ell \leq \ell' \Rightarrow \not\Pi \vdash \ell \leq \ell' \)
that an attacker, with restricted access to cryptographic keys, cannot distinguish ciphertexts. We lift the observation function to a process executing on node $n$ as below.

$$O(n, e, \Pi, A) = \langle n, O(e, \Pi, A) \rangle$$

We define the trace of a process using the observation function.

**Definition B.3** (Process Trace of a Node). Let $S$ be the well-typed distributed system as follows.

$$\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_k, e_k \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \Rightarrow^*$$

Then process trace of node $p_j$ with respect to a delegation context $\Pi$ and an attacker at level $\ell$, denoted as $Tr(S, \Pi, p_j, \ell)$, is the sequence of the expressions reduced on that node as observed by the attacker. That is, $O((p_j, e_j), \Pi, \ell), \ldots, O((p_j, e_j^k), \Pi, \ell)$ where is the trace concatenation operator.

Note that a trace may contain consecutive duplicates because it remains the same in those steps when other processes are scheduled to evaluate. The definition of trace deduplication is provided in Appendix C. We assume that the attacker is associated with the node $p_j$. The trace is computed as follows:

$$Tr(S, \Pi, p_j, \ell) = O((p_j, e_j), \Pi, \ell), \ldots, O((p_j, e_j^k), \Pi, \ell)$$

where $\Pi_j^D$ is the delegation approximation of the process running on node $p_j$. Intuitively, this is an approximation of the actual trace observed by an attacker during the execution of a process on the node $p_j$. Note that this is safe because it assumes the largest set of delegations that an attacker can influence. Two traces are equivalent if they are equal after applying the observation function and removing the duplicates.

**Definition B.4** (Trace Equivalence). Two traces, $Tr_1$ and $Tr_2$, are equivalent, denoted as $Tr_1 \approx Tr_2$, if they are equal after removing the duplicates. That is

$$Tr_1 \approx Tr_2 \iff U(Tr_1) = U(Tr_2)$$

We use these formal definitions to instantiate a stronger confidentiality attacker. Our noninterference result against this attacker generalizes the results stated in Theorem 1. Informally, it states that the attacker with limited confidentiality authority cannot distinguish between traces from executions differing in secret inputs. Theorem 2 formally states the noninterference against this attacker.

**Definition B.2** (Observation Function). We show $O(\epsilon, \Pi, A)$ for interesting cases; for the remaining terms, the function is homomorphically applied to the subexpressions of $\epsilon$.

$$e \quad O(\epsilon, \Pi, A) \quad Condition$$

$$\eta \quad O(\eta, v, \Pi, A) = \eta \quad \Pi \vdash A \Rightarrow \ell$$

$$\eta \quad O(\eta, v, \Pi, A) = \bullet \quad \Pi \not\vdash A \Rightarrow \ell$$

$$\epsilon \quad O(\epsilon, \Pi, A) = TEE^t \quad O(\epsilon', \Pi, A) \quad \Pi \vdash A \Rightarrow \ell^t$$

$$\epsilon \quad O(\epsilon, \Pi, A) = TEE^t \quad \Pi \not\vdash A \Rightarrow \ell^t$$

$$\epsilon \quad O(\epsilon, \Pi, A) = \text{runTEE}^t \quad O(\epsilon', \Pi, A) \quad \Pi \vdash A \Rightarrow \ell^t$$

$$\epsilon \quad O(\epsilon, \Pi, A) = \text{runTEE}^t \quad \Pi \not\vdash A \Rightarrow \ell^t$$

The observation function erases all terms protected at levels that do not flow to the level of the attacker. Similarly, it also erases the terms running inside a TEE that has less integrity and more confidentiality than the attacker. This precisely models
Fig. 18: DFLATE typing rules for secure operations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DT-UNIT</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pc \subseteq \ell \quad \Pi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-SEALLED</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pc \subseteq \ell$</td>
</tr>
<tr>
<td><strong>DT-BINDM</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \ell \quad \Pi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-ASSUME</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-WHERE</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-HALT</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-HALT2</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
</tbody>
</table>

Fig. 19: DFLATE typing rules for spawn and communication primitives

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DT-SPAWN</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pc \subseteq \ell \quad \Pi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-SEND</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-SENDDEL</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-RECEIVE</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-TEE</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-TEEQ</strong></td>
<td>$\Pi; \varnothing; \Theta; \phi; pc \vdash e: \tau \quad \Phi \vdash pl &gt; pl$</td>
</tr>
<tr>
<td><strong>DT-SYS</strong></td>
<td>$\forall i \in 1...n \quad \Pi; \varnothing; \Theta; \phi; pc \vdash e_i: \tau_i \quad \Phi \vdash pl &gt; pl$</td>
</tr>
</tbody>
</table>
\[\text{[BD-STEP]}\quad \text{pl, } D \vdash e_i \rightarrow e'_i \quad e_j = e_j \quad \{i, j\} = \{1, 2\}\]
\[\text{pl, } D \vdash (e_1 | e_2) \rightarrow (e'_1 | e'_2)\]

\[\text{[BD-APP]}\quad \text{pl, } D \vdash (\lambda(x : \tau_1)[pc_1, \Theta_1, P_1], e_1 | \lambda(x : \tau_2)[pc_2, \Theta_2, P_2], e_2) \quad v \rightarrow (e_1[x \mapsto [v]_1] | e_2[x \mapsto [v]_2])\]

\[\text{[BD-TAPP]}\quad \text{pl, } D \vdash (\Lambda X[pc_1, \Theta_1, P_1], e_1 | \Lambda X[pc_2, \Theta_2, P_2], e_2) \quad \tau \rightarrow (e_1[X \mapsto \tau] | e_2[X \mapsto \tau])\]

\[\text{[BD-UNPAIR]}\quad \text{pl, } D \vdash \text{proj}_j ([v_{11}, v_{12}] | [v_{11}, v_{22}]) \rightarrow ([v_{11} | v_{21})\]

\[\text{[BD-BINDM1]}\quad \text{pl, } D \vdash \text{bind } x = (\tau_{x_1} v_1 | \tau_{x_2} v_2) \quad \text{in } e \rightarrow ([e]_1[x \mapsto v_1] | [e]_2[x \mapsto v_2])\]

\[\text{[BD-BINDM3]}\quad \text{pl, } D \vdash \text{bind } x = (\tau_{x_1} v_1 | \tau_{x_2} v_2) \quad \text{in } e \rightarrow ([e]_1[x \mapsto v_1] | [e]_2[x \mapsto v_2])\]

\[\text{[BD-CASE]}\quad \text{pl, } D \vdash \text{case } ([\text{inj}_1 v_1 | \text{inj}_2 v_2] \circ \text{inj}_1(x), e_1 | \text{inj}_2(x), e_2) \rightarrow ([e]_1[x \mapsto v_1] | [e]_2[x \mapsto v_2])\]

Fig. 20: DFLATE Evaluation extensions

APPENDIX C

DFLATE NONINTERFERENCE PROOFS

We use Pottier and Simonet [33] technique to prove noninterference. At a high-level, the techniques involve extendin the calculus to have bracketed expressions and values that represent two different executions. Proving noninterference for DFLATE program entails proving that the individual projections of the final bracketed values are equal for inputs that are confidential and have less integrity.

Figure 20 shows evaluation extensions. Nesting of bracketed expressions isn’t allowed. Figure 22 defines projection. Typing extensions shown in Figure 23.

\[
\hat{v} ::= \ldots | (\hat{v} | \hat{e})
\]
\[
e ::= \ldots | (e | e)
\]

A. Adequacy

We prove that the bracketed semantics are adequate – sound and complete. By soundness, we mean that given a step in the bracketed execution, then at least one of the left or right projections take a step such that they are in relation with the bracketed execution. By completeness, we mean that given a left and right execution, we can construct a corresponding bracketed execution. We now prove the soundness of the bracketed semantics.

Lemma 1 (Soundness for Single Thread). If \( p, D \vdash e \rightarrow e' \) then \( p, D \vdash [e]_k \rightarrow [e']_k \) for \( k = \{1, 2\} \).

Proof. Inducting on the step relation. Interesting cases are described below.

**Case BD-STEP:** Since \( p, D \vdash (e_1 | e_2) \rightarrow (e'_1 | e'_2) \), we have that \( p, D \vdash e_k \rightarrow e'_k \). Since \( e_k = [e]_k \) and \( e'_k = [e']_k \), we have that \( p, D \vdash [e]_k \rightarrow [e']_k \).

**Case BD-APP:** Given \( p, D \vdash (\lambda(x : \tau_1)[pc_1, \Theta_1, P_1], e_1 | \lambda(x : \tau_2)[pc_2, \Theta_2, P_2], e_2) \quad v \rightarrow (e_1[x \mapsto [v]_1] | e_2[x \mapsto [v]_2]) \). For \( k = \{1, 2\} \), we have to prove that \( p, D \vdash [\lambda(x : \tau_k)[pc_k, \Theta_k, P_k], e_k] \quad [v]_k \rightarrow e_k[x \mapsto [v]_k] \)

Straightforward from DE-APP.

**Case BD-TAPP:** Similar to above case. Straightforward from DE-TAPP.

**Case BD-UNPAIR:** Straightforward from DE-UnPair.

**Case BD-BINDM1:** Straightforward from DE-BINDM.

**Case BD-BINDM3:** Same as above.

**Case BD-CASE:** Straightforward from DE-CASE1 and DE-CASE2.

Lemma 2 (Soundness for System). If \( (p_1, e_1) \parallel \cdots \parallel (p_i, e_i) \parallel \cdots \parallel (p_n, e_m) \Rightarrow (p_1, e'_1) \parallel \cdots \parallel (p_i, e'_i) \parallel \cdots \parallel (p_n, e'_m) \) then \( (p_1, [e_1]_k) \parallel \cdots \parallel (p_i, [e_i]_k) \parallel \cdots \parallel (p_n, [e_m]_k) \Rightarrow (p_1, [e_1]_k) \parallel \cdots \parallel (p_i, [e_i]_k) \parallel \cdots \parallel (p_n, [e_m]_k) \) for \( k = \{1, 2\} \).

Proof. We induct on the system step relation.
\( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, e_m \rangle \Longrightarrow \langle p_1, e'_1 \rangle \parallel \cdots \parallel \langle p_n, e'_n \rangle \)

\( i \in \{1, \ldots, m\} \quad e'_i = (e'_i \mid | e_{i,j2} \rangle \quad j = \begin{cases} \scriptstyle n & \text{if } n \neq m \\
 \{1, \ldots, m\} & \text{otherwise} \end{cases} \)

\[ e'_j = \begin{cases} e'_j & \text{if } n = m \text{ and } j = i \\ e_j & \text{if } n \neq m \\ \langle e_{n1} \mid e_{j2} \rangle & \text{if } n = m, j \neq i \text{ and } e_j = \langle e_{j1} \mid e_{j2} \rangle \\
W'[e_{n1} | e] & \text{if } n = m, j \neq i \text{ and } e_j = W[e] \end{cases} \]

\[ e'_j = \begin{cases} e'_j & \text{if } n = m \text{ and } j = i \\ e_j & \text{if } n \neq m \\ \langle e_{j1} \mid e_{j2} \rangle & \text{if } n = m, j \neq i \text{ and } e_j = \langle e_{j1} \mid e_{j2} \rangle \\
W'[e \mid e_{n2}] & \text{if } n = m, j \neq i \text{ and } e_j = W[e] \end{cases} \]

\[ e_i = W[(\text{spawn } q (p_i, \text{ch}_1, [p_{c_1}; \tau_1], \text{ch}_2, [p_{c_2}; \tau_2]), e_q \text{ then } e_{n1} | e)] \]

\[ e'_i = W[(e_{n1} \mid | \text{ch}_1, \text{ch}_2 | \mid e)] \]

\[ e_i = W[(\text{recv } x \mid e_{n1} | e)] \quad e'_i = W[(\text{send } v \mid e_{n1} | e)] \]

\[ e'_j = \begin{cases} W[j[(e'_{i} \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{recv } v \mid e_{n1} | e)] \\
W_{j}[(e'_{i} \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{recv } v \mid e_{n1} | e)] \end{cases} \]

\[ e_i = W[(e_{n1} \mid \text{send } v \mid e_{n1} | e)] \quad e'_i = W[(e_{n1} \mid \text{recv } v \mid e_{n1} | e)] \]

\[ e'_j = \begin{cases} W_{j}[(\text{recv } v \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{send } v \mid e_{n1} | e)] \\
W_{j}[(e_{n1} \mid \text{send } v \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{send } v \mid e_{n1} | e)] \end{cases} \]

\[ e_i = W[(e_{n1} \mid \text{recv } v \mid e_{n1} | e)] \quad e'_i = W[(e_{n1} \mid \text{send } v \mid e_{n1} | e)] \]

\[ e'_j = \begin{cases} W_{j}[(e'_{i} \mid \text{send } v \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{send } v \mid e_{n1} | e)] \\
W_{j}[(e'_{i} \mid e_{n1} | e)] & \text{if } e_j = W_{j}[(\text{send } v \mid e_{n1} | e)] \end{cases} \]

Fig. 21: DFLATE Evaluation extensions for system
Before proving completeness, we prove if the bracketed term is stuck then either left or right execution are stuck.

Fig. 22: Projection for bracketed expressions. \(\bullet\) denotes empty expression.

![Diagram](image-url)

Fig. 23: DFLATE Typing Extensions

**Case PAR-STEP:** From the premises of PAR-STEP, we have that \(p_i, D_{init} \vdash e_i \rightarrow e'_i\). Invoking soundness for single process (Lemma 1), we have that \(p_i, D_{init} \vdash |e_i| \rightarrow |e'_i|\) for some \(i \in \{1, \ldots, m\}\) and for all \(k \in \{1, 2\}\). Thus, we have \((p_1, [e_1]_k) \parallel \cdots \parallel (p_i, [e_i]_k) \parallel \cdots \parallel (p_m, [e_m]_k) \vdash (p_1, [e_1]_k) \parallel \cdots \parallel (p_i, [e_i]_k) \parallel \cdots \parallel (p_m, [e_m]_k)\).

**Case PAR-SPAWN:** Straightforward from the projection of premises.

**Case BD-PAR-STEP-L:** If \(n = m\) and \(j = i\), the case is a straightforward application of Lemma 1 (soundness of single process).

If \(n = m\) and \(j \neq i\), then from the premise, we have that \((p_i, [e_i]_1) \parallel \cdots \parallel (p_j, [e_j]_1) \Rightarrow (p_i, [e'_i]_1) \parallel \cdots \parallel (p_j, [e'_j]_1)\) where \([e_i]_1 = e_i\). Similarly, we have that \((p_i, [e_i]_2) \parallel \cdots \parallel (p_j, [e_j]_2) \Rightarrow (p_i, [e'_i]_2) \parallel \cdots \parallel (p_j, [e'_j]_2)\) where \([e_i]_2 = e_i\). Remaining terms are unchanged.

If \(n \neq m\) then remaining terms unchanged, we have that \((p_i, [e_i]_1) \Rightarrow (p_i, [e'_i]_1) \parallel \cdots \parallel (p_i, [e_i]_1) \parallel \cdots \parallel (p_i, [e_i]_2) \Rightarrow (p_i, [e'_i]_1) \parallel \cdots \parallel (p_i, [e_i]_1) \parallel (p_i, [e_i]_2) \Rightarrow (p_i, [e'_i]_2) \parallel \cdots \parallel (p_i, [e_i]_2)\). Hence proved.

**Case BD-PAR-STEP-R:** Proof similar to above case.

**Case BD-PAR-SPAWN-L:** Proof similar to the last case \((n \neq m)\) of BD-PAR-STEP-L.

**Case BD-PAR-SPAWN-R:** Proof similar to the last case \((n \neq m)\) of BD-PAR-STEP-L except that we now reason for taking the step on the right execution.

**Case BD-PAR-SEND-RECV-L:** Remaining terms unchanged, we have that \((p_i, W_i[e_{send} \nu v \text{ then } e_{\nu}] \parallel (p_j, W_i[e_{recv} \nu \text{ as } x \text{ in } e_{\nu}] \Rightarrow (p_i, W_i[e_{\nu}] \parallel (p_j, W_i[e_{\nu}]) \parallel (p_j, W_i[e_{\nu}[x \mapsto v]])\). The right projection of the process running on \(p_i\) and \(p_j\) remains unchanged. Hence proved.

**Case BD-PAR-SEND-RECV-R:** Similar to above case but the right projection takes a step and left projection remains unchanged.

**Case BD-PAR-RECV-SEND-L:** Similar to the case BD-PAR-SEND-RECV-L.

**Case BD-PAR-RECV-SEND-R:** Similar to the case BD-PAR-SEND-RECV-L.
Lemma 3 (Stuck Expressions for Single Thread). If \( \langle p, e \rangle \) gets stuck then \( \langle p, [e]_i \rangle \) is stuck for some \( i \in \{1, 2\} \).

Proof. We prove by induction on the structure of \( e \) that if \( \langle p, e \rangle \) gets stuck then \( \langle p, [e]_i \rangle \) is stuck for some \( i \in \{1, 2\} \).

Case \( \hat{e} \) : By assumption, \( e \) is not a value.

Case \( x \) : No reduction for a variable. Hence \( [x]_i \) is stuck as well.

Case \( e : e' \) : Since BD-APP and DE-APP are not applicable, it follows that \( e \) is not of the form \( \langle \lambda(x : \tau) v, \Theta', \mathcal{P}, \Pi \rangle, e_1 \rangle \parallel \lambda(x : \tau) v, \Theta', \mathcal{P}, \Pi \rangle, e_2 \rangle \) or \( \langle \lambda(x : \tau) v, \Theta', \mathcal{P}, \Pi \rangle, e \rangle \). Thus \( [e]_i \) is stuck as well.

Case \( \eta e \) : Since \( e \) is stuck, applying induction hypothesis it follows that \( [e]_i \) is also stuck and so \( [\eta e]_i \) is also stuck.

Case \( \text{pro}_{j} \), \( \text{inj}_{j} e \) : Similar to the above case.

Case \( e \tau \) : Since BD-TAPP and DE-TAPP are not applicable, it follows that \( e \) is not of the form \( (v_1 \mid v_2) \) or \( \Lambda X[p, \Theta, \mathcal{P}, \Pi]. e \).

Thus \( [e]_i \) is stuck as well.

Case \( \text{case } e \text{ of } \text{inj}_{j}(x) \) : Since BD-CASE and DE-CASE are not applicable, it follows that \( e \) is not of the form \( \langle \text{inj}_{j} v_1 \mid \text{inj}_{j} v_2 \rangle \) or \( \text{inj}_{j} v \). It follows that \( [\text{case } e \text{ of } \text{inj}_{j}(x), e_1 \mid \text{inj}_{j} v_2 \rangle, e_2 \rangle \) is also stuck.

Case \( \text{bind } x = v \text{ in } e \) : Similar to the above case.

Case \( \text{assume } e \text{ in } e' \) : Similar to the above case.

Case \( \text{TEE}^i e \) : Cannot get stuck.

Case \( \text{runTEE}^i e \) : Applying induction to \( e \), we have that \( \text{runTEE}^i e \) is stuck for some \( i \in \{1, 2\} \). \( \text{runTEE}^i e \) \( \text{runTEE}^i \) cannot get stuck.

Case Bracket : \( e = (e_1 \mid e_2) \). The only way this gets stuck is if both \( e_1 \) and \( e_2 \) get stuck. This is a contradiction.

Case Other : Remaining expressions do not take a step (single thread step relation \( \rightarrow \)).

Lemma 4 (Stuck Expressions for System). If \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) gets stuck then the configuration \( \langle p_1, [e_1]_k \rangle \parallel \cdots \parallel \langle p_n, [e_n]_k \rangle \) for some \( k \in \{1, 2\} \).

Proof. Induction on the structure of the system configuration.

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}'[\text{recv } v \text{ as } x \text{ in } e] \) on any other process, otherwise it would have taken a step using \( \text{PAR-SEND-RECV} \). Thus it is stuck for all \( k \in \{1, 2\} \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}'[\text{recv } v \text{ as } x \text{ in } e] \). On any other process, otherwise it would have taken a step using \( \text{BD-PAR-SEND-RECV-L} \). Thus it is stuck for \( k = 1 \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{recv } v \text{ as } x \text{ in } e] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : Similar to above case.

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ v then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : Similar to above case.

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ v then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : Similar to above case but \( k = 2 \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ v then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}'[\text{recv } v \text{ as } x \text{ in } e] \) or \( \text{W}[\text{recv } v \text{ as } x \text{ in } e] \), otherwise it would have taken a step using \( \text{BD-PAR-SEND-RECV-L} \). Thus \( k = 1 \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{send } v \text{ v then } e_{sn}] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}'[\text{recv } v \text{ as } x \text{ in } e] \) or \( \text{W}[\text{send } v \text{ v then } e_{sn}] \), otherwise it would have taken a step using \( \text{BD-PAR-SEND-RECV-R} \). Thus \( k = 2 \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{recv } v \text{ as } x \text{ in } e] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}[\text{recv } v \text{ as } x \text{ in } e] \) or \( \text{W}[\text{send } v \text{ v then } e_{sn}] \), otherwise it would have taken a step using \( \text{BD-PAR-SEND-RECV-R} \). Thus \( k = 1 \).

Case \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, \text{W}[\text{recv } v \text{ as } x \text{ in } e] \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \) : There is no matching \( \text{W}[\text{recv } v \text{ as } x \text{ in } e] \) or \( \text{W}[\text{send } v \text{ v then } e_{sn}] \), otherwise it would have taken a step using \( \text{BD-PAR-SEND-RECV-R} \). Thus \( k = 2 \).

Case Spawn \{Spawn always takes a step and cannot be stuck.

Case Other \{We have that all processes are stuck. Invoking Lemma 3 on each process, we have that each process is stuck for some \( k \in \{1, 2\} \).

We now prove the completeness of the bracketed semantics.

Lemma 5 (Completeness). If \( \langle p_1, [e_1]_k \rangle \parallel \cdots \parallel \langle p_n, [e_n]_k \rangle \) for all \( k \in \{1, 2\} \) then there exists a \( \hat{v}_i \) such that \( \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, e_n \rangle \Longrightarrow^* \langle p_1, e'_1 \parallel \cdots \parallel \langle p_n, e'_n \rangle \) for all \( k \in \{1, 2\} \).

Proof. From the rules in Figure 20, the brackets are always propagated outside, so they can only be applied finite number of times. If the system is stuck, then from Lemma 4, we have that either left or right executions are stuck. This is contradicting our assumption. Thus the system will not get stuck. Hence it must terminate to some value.
B. Proofs for Non-Interference

Before proving Theorem 1 and Theorem 2, we prove few supporting lemmas required for proving the type preservation.

**Lemma 6 (Closed Values in Send).** Let $\Pi_1; \emptyset; \Theta_1; p_1; p c_1 \vdash e_1: \tau_1 \parallel \cdots \parallel \Pi_1; \emptyset; \Theta_1; p_1; p c_1 \vdash e_k: \tau_k \parallel \cdots \parallel \Pi_1; \emptyset; \Theta_1; p_1; p c_1 \vdash e_m: \tau_m$. If $\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_n, e_i \rangle \parallel \cdots \parallel \langle p_n, e_m \rangle \Rightarrow^* \langle p_1, e_1' \rangle \parallel \cdots \parallel \langle p_n, e_n' \rangle$ then $\emptyset; \emptyset; p c_1' \vdash v: \tau$.

**Proof.** The only case where unbound variables appear is in closures. However, the top level program is closed and so all variables are $\beta$-substituted before getting reduced to $v$.

**Lemma 7 (Clearance).** Let $\Pi; \Gamma; \Theta; p; p c \vdash e: \tau$. Then $\Pi \vdash p \triangleright p c$

**Proof.** Proof is by straightforward induction on the typing judgment.

**Lemma 8 (Robust Assumption).** If $\Pi \vdash p c \triangleright \nabla(b)$, then $\Pi \cdot \langle a \triangleright b \rangle \vdash p c \triangleright \nabla(b)$ for any $a, b \in \mathcal{L}$.

**Lemma 9 (Robust Protection).** If $\Pi \vdash p c \leq \tau$, then $\Pi \cdot \langle a \triangleright b \rangle \vdash p c \leq \tau$ for any $a, b \in \mathcal{L}$.

**Lemma 10 (\Pi Extension).** If $\Pi; \Gamma; \Theta; p; p c \vdash e: \tau$ then $\Pi; \Pi; \Gamma; \Theta; p; p c \vdash e: \tau$ for any $p, q \in \mathcal{L}$.

**Proof.** Proof is by induction on the derivation of the typing judgment.

**Case DT-VAR:** Straightforward from the corresponding typing judgment.

**Case DT-UNIT:** Straightforward from the corresponding typing judgment.

**Case DT-DEL:** Straightforward from the corresponding typing judgment.

**Case DT-LAM:** Straightforward from the corresponding typing judgment.

**Case DT-APP:** Straightforward from the corresponding typing judgment.

**Case DT-TAPP:** Straightforward from the corresponding typing judgment.

**Case DT-PAIR:** Straightforward from the corresponding typing judgment.

**Case DT-UNPAIR:** Straightforward from the corresponding typing judgment.

**Case DT-INJ:** Straightforward from the corresponding typing judgment.

**Case DT-CASE:** Straightforward from the corresponding typing judgment.

**Case DT-UNITM:** Straightforward from the corresponding typing judgment.

**Case DT-BINDM:** Straightforward from the corresponding typing judgment.

**Case DT-ASSUME:** Given $\Pi; \Gamma; \Theta; p; p c \vdash \text{assume } e \text{ in } e': \tau$. From the premises,

$$\Pi; \Gamma; \Theta; p; p c \vdash e: \langle a \triangleright b \rangle$$  \hspace{1cm} (1)

$$\Pi \vdash p c \triangleright \nabla(b)$$  \hspace{1cm} (2)

$$\Pi \vdash \nabla(a') \triangleright \nabla(b')$$  \hspace{1cm} (3)

$$\Pi \cdot \langle a \triangleright b \rangle; \Gamma; \Theta; p; p c \vdash e': \tau$$  \hspace{1cm} (4)

Applying inductive step to (1) and (4), we have $\Pi \cdot \langle a \triangleright b \rangle, \langle p \triangleright q \rangle; \Gamma; \Theta; p; p c \vdash e': \tau$ and $\Pi \cdot \langle p \triangleright q \rangle; \Gamma; \Theta; p; p c \vdash e: \langle a \triangleright b \rangle$. Applying Lemma 8 to premises (C-B) and (C-B) we have $\Pi \cdot \langle p \triangleright q \rangle \vdash p c \triangleright \nabla(b)$ and $\Pi \cdot \langle p \triangleright q \rangle \vdash \nabla(a') \triangleright \nabla(b')$. Hence proved.

**Case DT-WHERE:** Similar to the proof of case DT-ASSUME.

**Case DT-BRACKET:** Applying induction to the premises of DT-BRACKET gives us that $\Pi \cdot \langle a \triangleright b \rangle; \Gamma; \Theta; p; p c \vdash \langle e_1 | e_2 \rangle: \tau$.

**Case DT-Bracket-VALUES:** Applying induction to premises, we have the conclusion.

**Case DT-SPAWN:** Applying induction to premises, we have the conclusion.

**Case DT-SEND:** Applying induction to premises, we have the conclusion.

**Case DT-SENDDEL:** Applying induction to premises, we have the conclusion.

**Case DT-RECEIVE:** Applying induction to premises, we have the conclusion.

**Case DT-TEE:** Applying induction to premises, we have the conclusion.

**Case DT-TEEG:** Applying induction to premises, we have the conclusion.

**Case DT-HALT:** Applying induction to premises, we have the conclusion.

**Case DT-HALT2:** Applying induction to premises, we have the conclusion.

The following lemma is required to prove the type preservation. It says that a expression $e$ well-typed at $p c$ is still well-typed at a reduced $p c'$ that satisfies the clearance.

**Lemma 11 (PC Reduction).** Let $\Pi; \Gamma; \Theta; p; p c \vdash e: \tau$. For all $p c, p c'$, such that $\Pi \vdash p c' \leq p c$ and $\Pi \vdash p \triangleright p c'$, then $\Pi; \Gamma; \Theta; p; p c' \vdash e: \tau$ holds.
Proof. Proof is by induction on the derivation of the typing judgment.

Case DT-VAR: Straightforward from the corresponding typing judgment.

Case DT-APP: Given $\Pi; \Gamma; \Theta; p; pc \vdash e \vdash e': \tau$. From DT-APP, we have

\[
\Pi; \Gamma; \Theta; p; pc \vdash e \vdash e : \tau_1 \\
\Pi; \Gamma; \Theta; p; pc \vdash e' : \tau_1 \\
\Pi \vdash pc \sqsubseteq pc'' \\
\Pi \vdash p \not\prec pc \\
\Theta \downarrow_{\text{dom}(\Theta')} = \Theta' \\
\forall (p \not\prec q) \in \Pi'. \Pi \vdash p \not\prec q
\]

Since $\Pi \vdash p \not\prec pc'$, Applying induction to the premises we have $\Pi; \Gamma; \Theta; p; pc' \vdash e : \tau_1$. From R-TRANS, we have $\Pi \vdash pc' \subseteq pc''$. Hence we have all the premises.

Case DT-TLAM: Straightforward from the corresponding typing judgment.

Case DT-PAIR: Similar to DT-App case.

Case DT-UNPAIR: Straightforward from the corresponding typing judgment.

Case DT-INJ: Straightforward from the corresponding typing judgment.

Case DT-METHOD: Straightforward from the corresponding typing judgment.

Case DT-BINDM: Before applying induction to the premises, we have to prove that $\Pi \vdash p \not\prec pc' \sqcup \ell$. Applying clearance (Lemma 7) to the premise $\Pi; \Gamma, x : \tau; \Theta; p; pc \sqcup \ell \vdash e : \tau$, we have that $\Pi \vdash p \not\prec pc \sqcup \ell$ and this implies $\Pi \vdash pc' \sqcup \ell \subseteq pc \sqcup \ell$. Hence we can apply induction to the premise and the rest of the proof is straightforward.

Case DT-ASSUME: Given $\Pi; \Gamma; \Theta; p; pc \vdash \text{assume } e \in e' : \tau$. From DT-ASSUME, we have

\[
\Pi; \Gamma; \Theta; p; pc \vdash e \vdash (p \not\prec q) \\
\Pi \vdash pc \not\prec \nabla(q) \\
\Pi \vdash \nabla(p^{-}) \not\prec \nabla(q^{-}) \\
\Pi \cdot (p \not\prec q); \Gamma; \Theta; p; pc \vdash e' : \tau \\
\Pi \vdash p \not\prec pc
\]

Applying inductive hypothesis to (11) and (14) we have $\Pi; \Gamma; \Theta; p; pc' \vdash e : (p \not\prec q)$. Applying inductive hypothesis to (14), we have $\Pi \cdot (p \not\prec q); \Gamma; \Theta; p; pc' \vdash e' : \tau$. Since $\Pi \vdash pc' \subseteq pc$, we have $\Pi \vdash pc' \not\prec \nabla(q)$. Combining, we have all the premises for DT-ASSUME and thus

\[
\Pi; \Gamma; \Theta; p; pc' \vdash \text{assume } e \in e' : \tau
\]

Case DT-WHERE: Given $\Pi; \Gamma; \Theta; p; pc \vdash e$ where $v : \tau$. From DT-WHERE, we have

\[
\Pi; \Gamma; \Theta; p; pc \vdash v \vdash (p \not\prec q) \\
\Pi \vdash pc \not\prec \nabla(q) \\
\Pi \vdash \nabla(p^{-}) \not\prec \nabla(q^{-}) \\
\Pi \cdot (p \not\prec q); \Gamma; \Theta; p; pc \vdash e : \tau
\]

Applying inductive hypothesis to (16) and (19) (and weakening delegation context (Lemma 10)), we have $\Pi; \Gamma; \Theta; p; pc' \vdash v \vdash (p \not\prec q)$ and $\Pi \cdot (p \not\prec q); \Gamma; \Theta; p; pc' \vdash e : \tau$. Using R-TRANS on (17), we have $\Pi \vdash pc' \not\prec \nabla(q)$ and so $\Pi \vdash pc' \not\prec \nabla(q)$. Combining, we have all the premises for T-WHERE and thus

\[
\Pi; \Gamma; \Theta; p; pc' \vdash e \text{ where } v : \tau
\]

Case DT-TEE: Given $\Pi; \Gamma; \Theta; p; pc \vdash \text{TEE}^t e : \tau$. From the premise, we have

\[
\Pi; \Theta; t ; t^{-} \wedge pc^{-} \vdash e : \tau
\]
Before applying induction to the premises, we have to prove that \( \Pi \mid t \gg t' \land pc'' \). From clearance (Lemma 7), we have that \( \Pi \mid t \gg t' \land pc' \). Combined with \( \Pi \mid pc' \sqsubseteq pc \), we have that

\[
\Pi \mid t \gg t' \land pc''
\]

We now use straight forward induction.

**Case DT-TEEG:** Similar to above case.

**Case DT-HALT:** Applying induction to premises, we have the conclusion.

**Case DT-HALT2:** Applying induction to premises, we have the conclusion.

**Case DT-RECEIVE:** Given \( \Pi; \Gamma; \Theta; pl; pc \vdash \text{recv ch as } x \text{ in } e': \tau \). We therefore have the premises

\[
\Pi; \Gamma; \Theta; pl; pc \vdash ch : \text{chan}_{p \rightarrow q} pc_{ch} \tau
\]

(20)

\[
\Pi; \Gamma, x : \tau; \Theta; p; pc'' \vdash e : \tau'
\]

(21)

\[
\Pi \mid pc \sqsubseteq pc''
\]

(22)

\[
\Pi \mid pc_{ch} \sqsubseteq pc''
\]

(23)

\[
\Pi \mid pc \sqsubseteq pc_{ch}
\]

(24)

\[
\Pi \mid p \gg pc
\]

(25)

\[
\Pi \mid p \gg pc_{ch}
\]

(26)

From DT-CHAN, we have

\[
\Pi; \Gamma; \Theta; p; pc' \vdash ch : \text{chan}_{p \rightarrow q} pc'' \tau
\]

Using R-TRANS, from (22) and (24), we have

\[
\Pi \vdash pc' \sqsubseteq pc''
\]

and

\[
\Pi \vdash pc' \sqsubseteq pc_{ch}
\]

Thus \( \Pi; \Gamma; \Theta; p; pc' \vdash \text{recv ch as } x \text{ in } e': \tau \)

**Case DT-SEND:** Proof similar to above case

**Case DT-SENDDEL:** Proof similar to above case except that values can be well-typed under empty delegation set.

**Case DT-SPAWN:** Given \( \Pi; \Gamma; \Theta; p; pc \vdash \text{spawn } @ q (pl, ch_1[p_1; \tau_1], ch_2[p_2; \tau_2]) \). \( e_1 \) then \( e_2 : \tau \). From the premises, we have

\[
\Pi; \Gamma; \Theta[ch_1 \mapsto \text{chan}_{p \rightarrow q} pc_1 \tau_1, ch_2 \mapsto \text{chan}_{p \rightarrow q} pc_2 \tau_2]; p; pc \vdash e_2 : \tau
\]

(27)

\[
\Pi; \Gamma[ch_1 \mapsto \text{chan}_{q \rightarrow p} pc_1 \tau_1, ch_2 \mapsto \text{chan}_{q \rightarrow p} pc_2 \tau_2]; q; pc \vdash e_1 : \text{unit} \Pi \vdash pc \sqsubseteq pc
\]

(28)

\[
\Pi \vdash p \gg pc
\]

(29)

\[
\Pi \vdash q \gg pc
\]

(30)

Straightforward application of induction hypothesis to the premises.

**Case DT-BRACKET:** The premise \( \Pi \mid H \sqcup pc \sqsubseteq pc'' \) implies \( \Pi \mid H \sqcup pc' \sqsubseteq pc'' \).

**Case DT-BRACKET-VALUES:** Applying induction to the premises gives the required conclusion.

\[\square\]

**Lemma 12** (Weakening Channel Environment for Values). If \( \Pi; \Gamma; \Theta; p; pc \vdash \hat{v} : \tau \) then \( \Pi; \Gamma; \Theta'; p; pc \vdash \hat{v} : \tau \) for any \( \Theta' \).

**Proof.** Proof is by induction on the structure of the values.

**Case Var:** Trivial

**Case Unit:** Trivial

**Case Delegation:** Trivial

**Case \( \eta \ell v \):** Apply induction to the premise of DT-UNITMSEALED.

**Case inj \( i v \):** Apply induction to the premise of DT-INJ.

**Case Lambda:** Trivial since functions carry their own channel environment.

**Case TLambda:** Same as above.

**Case Where:** Apply induction to the premise of DT-INJ.

**Case \( v; \Box \):** Apply induction to the premise of DT-HALT and DT-HALT2.
Case $(\hat{v}_1 \mid \hat{v}_2)$: From DT-BRACKET-VALUES, we have that
\[
\Pi \vdash H^\tau \leq \tau
\]
(31)
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_1 : \tau
\]
(32)
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_1 : \tau
\]
(33)
Applying induction to the premises, we have that
\[
\Pi \vdash H^\tau \leq \tau
\]
(34)
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_1 : \tau
\]
(35)
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_1 : \tau
\]
(36)
Thus $\Pi; \Gamma; \Theta'; p; \text{pc} \vdash (\hat{v}_1 \mid \hat{v}_2) : \tau$.

\[\square\]

Lemma 13 (Non-Bracket Values PC). Let $\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}_i \vdash e_i : \tau_i \mid \cdots | \Pi_m; \Gamma_m; \Theta_m; p_m; \text{pc}_m \vdash e_m : \tau_m$. If $\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}_i \vdash \hat{v} : \tau$ for some $i = \{1 \cdots m\}$ such that $\hat{v}$ is not a bracket then $\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}' \vdash \hat{v} : \tau$ for any $\text{pc}'$ such that $\Pi_i \not\vdash p_i \gg \text{pc}'$.

Proof. Recall that $\text{pc} = \prod_i \text{pc}_i$. Proof is by induction on the values. Also the condition $\Pi_i \not\vdash p_i \gg \text{pc}'$ ensures that the proof is straight forward for all values except functions and where terms. We prove important cases.

Case Function: Given $\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}_i \vdash \lambda(x : \hat{\tau})[\text{pc}''', \Theta'', \Xi, \Pi\lambda], e : \hat{\tau} \xrightarrow{\text{pc}', \Theta', \Pi\lambda} \tau$. We have to prove that:
\[
\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}' \vdash \lambda(x : \hat{\tau})[\text{pc}''', \Theta'', \Xi, \Pi\lambda], e : \hat{\tau} \xrightarrow{\text{pc}', \Theta', \Pi\lambda} \tau
\]

Case Where: Given $\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}_i \vdash e$ where $e = e_1 \mid e_2$. We have to prove that:
\[
\Pi_i; \Gamma_i; \Theta_i; p_i; \text{pc}' \vdash e \mid e_1 \mid e_2 : \tau
\]

From DT-WHERE, we have that $\Pi_i \vdash \text{pc} \gg \nabla(q)$. Note that $\text{pc}$ is constant for a given distributed configuration and does not change with the $\text{pc}'$. We are already given that $\Pi_i \not\vdash p_i \gg \text{pc}'$. Thus we have all premises for DT-WHERE.

\[\square\]

Lemma 14 (Variable Substitution). If $\Pi; \Gamma; \tau' : \Theta; p; \text{pc} \vdash e : \tau$ and $\Pi; \Gamma; \Theta; p; \text{pc} \vdash e : \tau'$, then $\Pi; \Gamma; \Theta; p; \text{pc} \vdash e[x \mapsto \hat{v}] : \tau$.

Proof. Proof is by induction on the structure of the expression $e$. Interesting case is bracket. Given $\Pi; \Gamma; \tau' : \Theta; p; \text{pc} \vdash (e_1 \mid e_2) : \tau$. We have to prove that
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash (e_1[x \mapsto \hat{v}_1] \mid e_2[x \mapsto \hat{v}_2]) : \tau
\]

We first describe the case DT-BRACKET. The proof for the case DT-BRACKET-VALUES is analogous. From DT-BRACKET, we have that
\[
\Pi; \Gamma, x : \tau' ; \Theta; p; \text{pc}' \vdash e_1 : \tau
\]
(37)
\[
\Pi; \Gamma, x : \tau' ; \Theta; p; \text{pc}' \vdash e_2 : \tau
\]
(38)
\[
\Pi \vdash (H^\tau \sqcup \text{pc}) \subseteq \text{pc}'
\]
(39)
\[
\Pi \vdash H^\tau \leq \tau
\]
(40)
Applying clearance (Lemma 7), we have that $\Pi \vdash p \gg \text{pc}'$. Depending on whether $\hat{v}$ is a bracket value, we have two cases

Case $\hat{v} = (\hat{v}_1 \mid \hat{v}_2)$: From DT-BRACKET-VALUES, we have that
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_1 : \tau'
\]
(41)
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash \hat{v}_2 : \tau'
\]
(42)
\[
\Pi \vdash H^\tau \leq \tau'
\]
(43)
Since values can be typed under any $\text{pc}$ which acts for the place under which value is typed (Lemma 13), we have that $\Pi; \Gamma; \Theta; p; \text{pc}' \vdash \hat{v}_i : \tau'$ for all $i \in \{1, 2\}$. Applying induction to the premises (37) and (38), we get
\[
\Pi; \Gamma; \Theta; p; \text{pc}' \vdash e_1[x \mapsto \hat{v}_1] : \tau
\]
(44)
\[
\Pi; \Gamma; \Theta; p; \text{pc}' \vdash e_2[x \mapsto \hat{v}_2] : \tau
\]
(45)
and thus
\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash (e_1[x \mapsto \hat{v}_1] \mid e_2[x \mapsto \hat{v}_2]) : \tau
\]
Case $\hat{v} \neq (\hat{v}_1 | \hat{v}_2)$: Since values can be typed under any $pc$ which acts for the place under which value is typed (Lemma 13), we have that $\Pi; \Gamma; \Theta; p; pc' \vdash \hat{v}'$. Applying induction to the premises (37) and (38), we get

$$\Pi; \Gamma; \Theta; p; pc' \vdash e_1[x \mapsto \hat{v}]: \tau \quad (46)$$

$$\Pi; \Gamma; \Theta; p; pc' \vdash e_2[x \mapsto \hat{v}]: \tau \quad (47)$$

and thus

$$\Pi; \Gamma; \Theta; p; pc \vdash (e_1[x \mapsto \hat{v}] | e_2[x \mapsto \hat{v}]): \tau$$

Lemma 15 (Projection Preserves Types). If $\Pi; \Gamma; \Theta; p; pc \vdash e: \tau$, then $\Pi; \Gamma; \Theta; p; pc \vdash [e]_i: \tau$ for $i = \{1, 2\}$.

Proof. Proof is by induction on the typing derivation. Interesting case is DT-BRACKET. Applying Lemma 11 to the expression typing judgments, we have the proof.

Lemma 16 (Type Substitution Preserves Well-Typedness). Let $\tau'$ be well-formed in $\Gamma, X, \Gamma'$. If $\Pi; \Gamma, X, \Gamma'; \Theta; p; pc \vdash e: \tau$ then $\Pi; \Gamma; \Gamma'[X \mapsto \tau']; \Theta; p; pc \vdash e[X \mapsto \tau']: \tau[X \mapsto \tau']$.

Proof. Proof is by induction on the typing derivation $\Pi; \Gamma; \Theta; p; pc \vdash e: \tau$.

Case DT-Var: Depending on whether $\tau$ uses $X$ two cases arise. Either $\Pi; \Gamma, x : \tau, X, \Gamma'; pc; x: \tau \vdash o: r \Pi; \Gamma, X, x : \tau, \Gamma'; \Theta; p; pc \vdash x: \tau$. In the former case, since $\Gamma, x : \tau$ does not use $X$, we have that $\tau[X \mapsto \tau']$ is same as $\tau$.

In the latter case, we have that $\Pi; \Gamma, X, x : \tau[X \mapsto \tau'], \Gamma'[X \mapsto \tau']; \Theta; p; pc \vdash x: \tau[X \mapsto \tau']$.

Case DT-Unit: Straightforward from the corresponding typing judgment.

Case DT-Del: Straightforward from the corresponding typing judgment.

Case DT-Lam: Given $\Pi; \Gamma, X, \Gamma'; \Theta; p; pc \vdash \lambda(x : \tau_1)[pc', \Theta, \Pi_\lambda], e : \tau_1 \vdash pc', \Theta, \Pi_\lambda, \tau_2$. From DT-LAM $\Pi; \Gamma, x : \tau, X, \Gamma', \Theta; p; pc \vdash e[X \mapsto \tau']: \tau_2[X \mapsto \tau']$. From DT-LAM, we thus have $\Pi; \Gamma, \Gamma'[X \mapsto \tau']; \Theta; p; pc \vdash \lambda(x : \tau_1)[pc', \Theta, \Pi_\lambda], e[X \mapsto \tau']: \tau_2[X \mapsto \tau']$. Applying induction hypothesis to the premise, we have $\Pi; \Gamma, \Gamma'[X \mapsto \tau'] : \Theta; p; pc \vdash e[X \mapsto \tau'] : \tau_1[X \mapsto \tau']$.

Case DT-App: Given $\Pi; \Gamma, X, \Gamma'; \Theta; p; pc \vdash e' : \tau_2$. Applying induction hypothesis to the premises, we have

$$\Pi; \Gamma, X, \Gamma'[X \mapsto \tau']: \Theta; p; pc \vdash e[X \mapsto \tau'] : \tau_1[X \mapsto \tau'] \quad (48)$$

$$\Pi; \Gamma, X, \Gamma'[X \mapsto \tau']: \Theta; p; pc \vdash e'[X \mapsto \tau'] : \tau_1[X \mapsto \tau'] \quad (49)$$

From DT-APP, we thus have $\Pi; \Gamma, X, \Gamma'[X \mapsto \tau']: \Theta; p; pc \vdash e[X \mapsto \tau'] \vdash e'[X \mapsto \tau'] : \tau_2[X \mapsto \tau']$.

Case DT-TLam: Given $\Pi; \Gamma, X, \Gamma'; \Theta; p; pc \vdash \Lambdasrc{Y}[pc', \Theta', \Pi'] : \forall Y[pc', \Theta', \Pi'], \tau$. Applying induction hypothesis to the premise of DT-TLAM we have

$$\Pi; \Gamma, (\Gamma', Y)[X \mapsto \tau'] : \Theta; p; pc \vdash e[X \mapsto \tau']: \tau[X \mapsto \tau']$$

So from DT-TLAM, we have

$$\Pi; \Gamma, \Gamma'[X \mapsto \tau'] : \Theta; p; pc \vdash \Lambdasrc{Y}[pc', \Theta', \Pi'] : \forall Y[pc', \Theta', \Pi'], \tau[X \mapsto \tau']$$

Case DT-TApp: Given $\Pi; \Gamma, X, \Gamma'; \Theta; p; pc \vdash e'' : \tau[Y \mapsto \tau''$]. Applying induction hypothesis to the premises of T-TAPP, we have $\Pi; \Gamma, X, \Gamma'[X \mapsto \tau'] : \Theta; p; pc \vdash e[X \mapsto \tau'] : \forall Y[pc', \Theta', \Pi'], \tau[X \mapsto \tau']$. From DT-TAPP, we thus have $\Pi; \Gamma, \Gamma'[X \mapsto \tau'] : \Theta; p; pc \vdash e[X \mapsto \tau'] : \tau''[X \mapsto \tau'][Y \mapsto \tau''$]

Case DT-Pair: Straightforward application of induction hypothesis to the premises.

Case DT-UnPair: Straightforward application of induction hypothesis to the premises.

Case DT-Inj: Straightforward application of induction hypothesis to the premises.

Case DT-UnitM: Straightforward application of induction hypothesis to the premises.

Case DT-BindM: Straightforward application of induction hypothesis to the premises.

Case DT-Assume: Straightforward application of induction hypothesis to the premises.

Case DT-Where: Straightforward application of induction hypothesis to the premises.

Case DT-TEE: Straightforward application of induction hypothesis to the premises.

Case DT-TEEG: Straightforward application of induction hypothesis to the premises.

Case DT-Halt: Straightforward application of induction hypothesis to the premises.

Case DT-Halt2: Straightforward application of induction hypothesis to the premises.

Case DT-Spawn: Straightforward application of induction hypothesis to the premises.

Case DT-SendDel: Straightforward application of induction hypothesis to the premises.
Case DT-Send: Straightforward application of induction hypothesis to the premises.

Case DT-Receive: Straightforward application of induction hypothesis to the premises.

Case DT-BRACKET: Straightforward application of induction hypothesis to the premises.

Case DT-BRACKET-VALUES: Straight forward application of induction hypothesis to the premises.

Lemma 17 (Non-bracket values sent over a channel). Let \( \forall \Pi. \Pi \vdash p \trianglerighteq pc \implies \Pi; \emptyset; p; pc \vdash v : \tau \). Then \( \Pi'; \emptyset; q; pc' \vdash v : \tau \) for any \( pc' \) and \( q \) such that \( \Pi' \vdash q \trianglerighteq pc' \).

Proof. Induction on the structure of values.

Case Unit: Given that \( \Pi' \vdash q \trianglerighteq pc' \). From DT-UNIT, we thus have \( \Pi'; \emptyset; q; pc' \vdash () : \text{unit} \).

Case Pair: Given \( \Pi; \emptyset; p; pc \vdash (v_1, v_2) : \tau_1 \times \tau_2 \) for all \( \Pi \) such that \( \Pi \vdash p \trianglerighteq pc \). Inverting DT-PAIR, we have \( \Pi; \emptyset; p; pc \vdash v_1 : \tau_1 \). Also given that \( \Pi' \vdash q \trianglerighteq pc' \). Applying induction hypothesis, we have \( \Pi'; \emptyset; q; pc' \vdash v_1 : \tau_1 \). From DT-PAIR, we thus have all premises to prove \( \Pi'; \emptyset; q; pc' \vdash (v_1, v_2) : \tau_1 \times \tau_2 \).

Case Del: Similar to the above case.

Case Inj: Similar to the above case.

Case Seal: Similar to the above case.

Case Where: Similar to the above case.

Case Lam: Given \( \Pi; \emptyset; p; pc \vdash \lambda (x : \tau)\chi[pc', \Theta', \Pi, \lambda_\chi].e : \tau \). From the premises it follows that \( \Pi' \vdash q \trianglerighteq pc' \). Combining this with above, from DT-LAM, we thus have all premises to prove \( \Pi'; \emptyset; q; pc' \vdash \lambda (x : \tau)\chi[pc', \Theta', \Pi, \lambda_\chi].e : \tau \).

Case TLam: Similar to above.

Using the above lemmas, we are now ready to prove type preservation for a process.

Lemma 18 (Type Preservation for a Process). Let \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash e_j : \tau_j \), for every \( j \in \{1 \ldots m\} \). If \( p_j, D_j \vdash e_j \triangleright e_j' \) such that for every \( \langle p \trianglerighteq q \rangle \in \Pi_j \), \( \langle p \trianglerighteq q \rangle \in D_j \) then \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash e_j' : \tau_j \).

Proof. We prove by inducting the on the step relation \( p_j, D_j \vdash e_j \triangleright e_j' \). We only prove interesting cases. Remaining cases (DE-AP\( \Pi \), DE-TAP\( \Pi \), DE-CASE' DE-UNITM', DE-BINDM', DE-ASSUME', DE-SEND') follow from straightforward application of induction hypothesis to the premises.

Case DE-APP: Given \( e_j = (\lambda (x : \tau)\chi[pc', \Theta, \Pi, \lambda_\chi].e) v \) and \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash (\lambda (x : \tau)\chi[pc', \Theta, \Pi, \lambda_\chi].e) v : \tau_2 \). From the premises we have,
\[ \Pi_j; \Gamma_j; x : \tau; \Theta; p_j; pc_j \vdash e : \tau_j \text{ and } \Pi_j; \Gamma_j; x : \tau; \Theta; p_j; pc_j \vdash v : \tau' \]
Since \( \Pi_j \vdash p_j \trianglerighteq pc_j \), invoking PC reduction (Lemma 11), we have that \( \Pi_j; \Gamma_j; x : \tau; \Theta; p_j; pc_j \vdash e : \tau_j \). We have \( e_j' = e_j(x/v) \). Invoking Lemma 14, we have \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash e_j(x/v) : \tau_j \).

Case DE-TAPP: Similar to above case.

Case DE-CASE1: Given
\[ e_j = \text{case } (\text{inj}_j v) \text{ of } \text{inj}_j(x). e_j1 \mid \text{inj}_j2(x). e_j2 \]
and \( e_j' = e_{j1}[x/v] \). Also,
\[ \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash \text{case } (\text{inj}_j v) \text{ of } \text{inj}_j(x). e_j1 \mid \text{inj}_j2(x). e_j2 : \tau \]. From the premises we have,
\[ \Pi_j; \Gamma_j; x : \tau; \Theta; p_j; pc_j \vdash \text{inj}_j v : \tau_j \text{ and } \Pi_j; \Gamma_j; x : \tau; \Theta; p_j; pc_j \vdash e_{j1} : \tau_j \]. From DT-INJ, we have \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash v : \tau_j' \). Invoking Lemma 14, we have \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash e_{j1}[x/v] : \tau_j \).

Case DE-CASE2: Similar to above.

Case DE-UNITM: Given \( e_j = \eta \ell v \) and \( e_j' = \eta \ell v \). Also, \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash \eta \ell v : \ell \text{ says } \tau \). From the premises it follows that \( \Pi_j; \Gamma_j; \Theta; p_j; pc_j \vdash \eta \ell v : \ell \text{ says } \tau \).

Case DE-BINDM: Given \( e_j = \text{bind } x = v \text{ in } e' \) and \( e_j' = e'(x/v) \) Also, \( \Pi_j; \Gamma; \Theta; p; pc \vdash \text{bind } x = v \text{ in } e' : \tau \). From the premises, we have the following:
\[ \Pi; \Gamma; \Theta; p; pc \vdash v : \tau' \]
\[ \Pi; \Gamma; x : \tau'; \Theta; p; pc \vdash \text{bind } x = v : \tau \]
\[ \Pi \vdash pc \cup \ell \leq \tau \]
\[ \Pi \vdash p \trianglerighteq pc \]
We have to prove that $\Pi; \Gamma; \Theta; p; pc \vdash e'(x/v) : \tau$. Since we have that $\Pi \vdash p \geq pc$, applying PC reduction (Lemma 11) to the premise (51), we have $\Pi; \Gamma; x : \tau'; \Theta; p; pc \vdash e' : \tau$.

Invoking Lemma 14, we thus have $\Pi; \Gamma; \Theta; p; pc \vdash e'(x/v) : \tau$.

**Case DE-ASSUME:** Given $e = \text{assume } v \text{ in } e'$ and $e' = e'$ where $v$. Also, $\Pi; \Gamma; \Theta; p; pc \vdash \text{assume } v \text{ in } e' : \tau$. Let $D$ be such that for every $\langle p \geq q \rangle \in \Pi$, $\langle p \geq q \rangle \in D$. From DT-ASSUME, we have

$$
\Pi; \Gamma; \Theta; p; pc \vdash e' \text{ where } v: \tau
$$

Comparing with the given premises, we already have the required premises.

$$
\Pi; \Gamma; \Theta; p; pc \vdash v: (p \geq q) \tag{54}
\Pi \vdash p \geq \nabla(q) \tag{55}
\Pi \vdash \nabla(p'\nabla) \geq \nabla(q\nabla) \tag{56}
\Pi \vdash p' \geq pc \tag{57}
$$

We need to prove:

$$
\Pi; \Gamma; \Theta; p; pc \vdash e' \text{ where } v: \tau
$$

Hence proved.

**Case DE-WHERE:** Given $e = e$ where $v$ and $e' = e'$ where $v$. Also, $\Pi; \Gamma; \Theta; p; pc \vdash e$ where $v: \tau$. Let $D$ be such that for every $\langle p \geq q \rangle \in \Pi$, $\langle p \geq q \rangle \in D$. From DT-WHERE, we have that

$$
\Pi \cdot (a \geq b); \Gamma; \Theta; p; pc \vdash e: \tau \tag{60}
\Pi; \Gamma; \Theta; p; pc \vdash v: (a \geq b) \tag{60}
$$

From DE-WHERE, we have $p, D \cdot v \vdash e \rightarrow e'$. So, we have that $v \in D, v \in \Pi \cdot (a \geq b)$. Applying inductive to the premise (66), we have that $\Pi \cdot (a \geq b); \Gamma; \Theta; p; pc \vdash e': \tau$ and so $\Pi; \Gamma; \Theta; p; pc \vdash e': \tau$ where $v: \tau$.

**Case DE-TEE:** Given $e = \text{TEE}^t e$ and $e' = \text{runTEE}^t e$. Also given $\Pi; \Gamma; \Theta; p; pc \vdash \text{TEET}^t e: \text{unit}; \Box$. From DT-TEEG, we trivially have that $\Pi; \Gamma; \Theta; p; pc \vdash \text{runTEE}^t e: \text{unit}; \Box$.

**Case DE-TEEG':** Given $e = \text{runTEE}^t e$ and $e' = \text{runTEE}^t e'$. Also given $\Pi; \Gamma; \Theta; p; pc \vdash \text{runTEE}^t e: \text{unit}; \Box$. From the premises of DT-TEEG, we have

$$
\Pi; \emptyset; \Theta; t; t^{-} \land pc^{-} \vdash e: \tau \tag{68}
$$

From the premises of DE-TEEG, we have

$$
t, D \vdash e \rightarrow e'
$$

Applying induction hypothesis to the premise, we have

$$
\Pi; \emptyset; \Theta; t; t^{-} \land pc^{-} \vdash e': \tau
$$

Hence $\Pi; \Gamma; \Theta; p; pc \vdash \text{runTEE}^t e': \tau$.

**Case DE-TEEG:** Given $e = \text{TEE}^t v$ and $e' = (\Box)$; $\Box$. Also given $\Pi; \Gamma; \Theta; p; pc \vdash \text{runTEE}^t v: \text{unit}; \Box$. From DT-UNIT and DT-HALT, we have that $\Pi; \Gamma; \Theta; p; pc \vdash (\Box): \text{unit}; \Box$. Thus $\Pi; \Gamma; \Theta; p; pc \vdash e': \text{unit}; \Box$.

**Case DE-SEND:** Given $p, D \vdash \langle p, \text{send } v \cdot \text{then } e_{sn} \rangle \rightarrow \langle p, \text{send } v \cdot v' \cdot \text{then } e_{sn}; \Box \rangle$ where $v' = \text{closure_del}(D, v)$. Also, $\Pi; \Gamma; \Theta; p; pc \vdash \text{send } v \cdot \text{then } e_{sn}; \Box$. We need to prove that

$$
\Pi; \Gamma; \Theta; p; pc \vdash \text{send } v \cdot v' \cdot \text{then } e_{sn}; \Box: \hat{t}; \Box
$$
Inverting the premises of DT-SEND, we have that

\[
\Pi; \Gamma; \Theta; pl; pc \vdash ch : \text{chan}_{p \leftarrow q} \quad \text{pc}_{ch} \stackrel{\ast}{\downarrow}
\]

\[
\Pi; \Gamma; \emptyset; p; pc \vdash v : \emptyset
\]

\[
\Pi; \Gamma; \Theta; p; pc' \vdash e_{\text{nk}} : \tau'
\]

\[
\tau' \in \{\emptyset^\ast, \emptyset^\downarrow\}
\]

\[
\Pi \models pc \subseteq pc_{ch}
\]

\[
\Pi \models pc_{ch} \subseteq pc'
\]

\[
\Pi \models pc' \leq \tau'
\]

\[
\Pi \models p \triangleright pc
\]

\[
\Pi \models p \triangleright pc_{ch}
\]

\[
(69)
\]

\[
(70)
\]

\[
(71)
\]

\[
(72)
\]

\[
(73)
\]

\[
(74)
\]

\[
(75)
\]

\[
(76)
\]

\[
(77)
\]

To prove what we want, we have all the required premises except

\[
\Pi'; \emptyset; \emptyset; p; pc \vdash v' : \tau
\]

for some \(\Pi'\) such that \(\Pi' \subseteq \Pi\). Since \text{closure}_\text{del}(D, v)\) closes over required delegations, \(v'\) requires no further assumptions on delegations. We still need to prove that the variable typing context can be empty. Because of substitution semantics, if we start with a closed program then an expression that gets evaluated to a value is closed as well (Lemma 6). Hence proved.

**Case DE-HALT\textsuperscript{1}:** Applying induction hypothesis to the premise, we have the conclusion.

**Case DE-HALT\textsuperscript{2}:** Given \(e = v; \emptyset\emptyset\emptyset\) and \(\epsilon' = v; \emptyset\emptyset\emptyset\). From DT-HALT, the type of \(v; \emptyset\emptyset\emptyset\) is \(\emptyset; \emptyset\emptyset\emptyset\). From DT-HALT\textsuperscript{2}, the type of \(e\) is thus \(\emptyset; \emptyset\emptyset\emptyset\). We already have that the type of \(\epsilon'\) is also \(\emptyset; \emptyset\emptyset\emptyset\). Hence proved.

**Case DW-APP:** Given \(e = (v \text{ where } \langle p \triangleright q \rangle) \ v'\) and \(\epsilon' = (v v') \text{ where } \langle p \triangleright q \rangle\). We have to prove that

\[
\Pi; \Gamma; \Theta; p; pc \vdash (v v') \text{ where } \langle p \triangleright q \rangle : \tau
\]

From DT-APP we have:

\[
\Pi; \Gamma; \Theta; pl; pc \vdash v \text{ where } \langle p \triangleright q \rangle : \tau_1 \quad \text{pc}' \quad \Theta' \quad \Pi \models \text{pc} \triangleright pc' \quad pl \in P \quad \Theta \models \text{dom}(\Theta') = \Theta' \quad \forall \langle p \triangleright q \rangle \in \Pi' \quad \Pi \models pl \triangleright pc
\]

\[
(78)
\]

\[
(79)
\]

\[
(80)
\]

\[
(81)
\]

\[
(82)
\]

\[
(83)
\]

\[
(84)
\]

Rule DT-WHERE gives us the following:

\[
\Pi \cdot \langle p \triangleright q \rangle; \Gamma; \Theta; p; pc \vdash (p \triangleright q) ; (\langle p \triangleright q \rangle) \\
\Pi \models pc \triangleright \text{q}(q) \\
\Pi \models \text{q}(q) \triangleright \text{q}(q) \\
\Pi \cdot \langle p \triangleright q \rangle; \Gamma; \Theta; p; pc \vdash v : \tau_1 \quad \text{pc}' \quad \Theta' \quad \Pi \models \text{pc} \triangleright pc' \quad pl \in P \\
\Theta \models \text{dom}(\Theta') = \Theta' \\
\forall \langle a \triangleright b \rangle \in \Pi' \quad \Pi \cdot \langle p \triangleright q \rangle \models \langle a \triangleright b \rangle \\
\Pi \cdot \langle p \triangleright q \rangle \models pl \triangleright pc
\]

\[
(85)
\]

\[
(86)
\]

\[
(87)
\]

\[
(88)
\]

We now want to show that \(v\) is well typed via DT-WHERE. The key premise is to show that the subexpression \((v v')\) is well-typed via DT-APP. That is,

\[
\Pi \cdot \langle p \triangleright q \rangle; \Gamma; \Theta; p; pc \vdash v v' : \tau
\]

\[
(89)
\]

First, from (79), (80), (89) to (84) and lemma extending delegation contexts (Lemma 10 and Lemma 8):

\[
\Pi \cdot \langle p \triangleright q \rangle; \Gamma; \Theta; p; pc \vdash v' : \tau_1
\]

\[
(90)
\]

\[
\Pi \cdot \langle p \triangleright q \rangle \models pc \subseteq pc' \quad pl \in P \\
\Theta \models \text{dom}(\Theta') = \Theta' \\
\forall \langle a \triangleright b \rangle \in \Pi' \quad \Pi \cdot \langle p \triangleright q \rangle \models \langle a \triangleright b \rangle \\
\Pi \cdot \langle p \triangleright q \rangle \models pl \triangleright pc
\]

\[
(91)
\]

\[
(92)
\]

\[
(93)
\]

\[
(94)
\]
Combining with (88), we have (89). Remaining clauses give us

\[ \Pi; \Gamma; \Theta; p; pc \vdash (v \; v') \text{ where } \langle p \triangleright q \rangle : \tau \]

**Case DW-TAPP:** Given \( e = (v \text{ where } \langle p \triangleright q \rangle) \) \( \tau \) and
\( e' = (v' \; \tau') \text{ where } \langle p \triangleright q \rangle \). We have to prove that

\[ \Pi; \Gamma; \Theta; p; pc \vdash (v' \; \tau') \text{ where } \langle p \triangleright q \rangle : \tau[X \mapsto \tau'] \]

From DT-TAPP we have:

\[ \Pi; \Gamma; \Theta; p; pl; pc \vdash v \text{ where } \langle p \triangleright q \rangle : \forall X[pc', \Theta', \varphi, \Pi']. \tau \]

\[ \Pi \vdash pc \sqsubseteq pc' \] (96)

**Rule DT-WHERE** gives us the following:

\[ \Pi; \Gamma; \Theta; p; pc \vdash (p \triangleright q) : (p \triangleright q) \]

\[ \Pi \vdash pc \triangleright \triangleright (q) \] (98)

\[ \Pi \vdash \triangleright (p^{-}) \triangleright \triangleright (q^{-}) \] (99)

\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; pc \vdash v : \forall X[pc', \Theta', \varphi, \Pi']. \tau \] (100)

We now want to show that \( e' \) is well typed via DT-WHERE. The key premise is to show that the subexpression \((v \; \tau')\) is well-typed via DT-TAPP. That is,

\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; pc \vdash v' : \tau[X \mapsto \tau'] \] (101)

First, from (96) and extending delegation context (Lemma 8), we get:

\[ \Pi \cdot (p \triangleright q) \vdash pc \sqsubseteq pc' \] (102)

Combining with (100), we have (101). Remaining clauses give us

\[ \Pi; \Gamma; \Theta; p; pc \vdash (v' \; \tau') \text{ where } \langle p \triangleright q \rangle : \tau[X \mapsto \tau'] \]

**Case DW-UNPAIR:** Given \( e = \text{proj}_i ((v_1, v_2) \text{ where } \langle p \triangleright q \rangle) \) and \( e' = (\text{proj}_i (v_1, v_2)) \text{ where } \langle p \triangleright q \rangle \). We have to prove that

\[ \Pi; \Gamma; \Theta; p; pc \vdash (\text{proj}_i (v_1, v_2)) \text{ where } \langle p \triangleright q \rangle : \tau_i \]

From DT-UNPAIR, we have:

\[ \Pi; \Gamma; \Theta; p; pc \vdash (v_1, v_2) \text{ where } \langle p \triangleright q \rangle : \tau_1 \times \tau_2 \] (103)

From (103) and DT-WHERE, we have:

\[ \Pi; \Gamma; \Theta; p; pc \vdash (p \triangleright q) : (p \triangleright q) \] (104)

\[ \Pi \vdash pc \triangleright \triangleright (q) \] (105)

\[ \Pi \vdash \triangleright (p^{-}) \triangleright \triangleright (q^{-}) \] (106)

\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; pc \vdash (v_1, v_2) : \tau_1 \times \tau_2 \] (107)

From (107) and DT-UNPAIR, we have: \( \Pi \cdot (p \triangleright q)pc \sqcup \ell; \Gamma; \Theta; p; pc \vdash \text{proj}_i (v_1, v_2) : \tau_i \). Combining with remaining premises ((104) to (107)) we have \( \Pi; \Gamma; \Theta; p; pc \vdash \text{proj}_i (v_1, v_2) : \tau_i \). 

**Case DW-CASE:** Given \( e = \text{case } (v \text{ where } \langle p \triangleright q \rangle) \text{ of } \text{inj}_1(x). e_1 | \text{inj}_2(x). e_2 \) and \( e' = (\text{case } v \text{ of } \text{inj}_1(x). e_1 | \text{inj}_2(x). e_2) \text{ where } \langle p \triangleright q \rangle \). We have to prove that

\[ \Pi; \Gamma; \Theta; p; pc \vdash (\text{case } v \text{ of } \text{inj}_1(x). e_1 | \text{inj}_2(x). e_2) \text{ where } \langle p \triangleright q \rangle : \tau \]

From DT-CASE we have:

\[ \Pi; \Gamma; \Theta; p; pc \vdash v \text{ where } \langle p \triangleright q \rangle : \tau_1 + \tau_2 \] (108)

\[ \Pi \vdash pc \sqcup \ell \leq \tau \] (109)

\[ \Pi \vdash pc \sqsubseteq \ell \] (110)

\[ \Pi; \Gamma; x : \tau_1; \Theta; p; pc \sqcup \ell \vdash e_1 : \tau \] (111)

\[ \Pi; \Gamma; x : \tau_2; \Theta; p; pc \sqcup \ell \vdash e_2 : \tau \] (112)
From (108) and rule DT-WHERE, we get the following:

\[ \Pi; \Gamma; \Theta; p; \phi \vdash \langle p \triangleright q \rangle; (p \triangleright q) \quad (113) \]
\[ \Pi \vdash \phi \triangleright \nabla(q) \quad (114) \]
\[ \Pi \vdash \nabla(p^{-}) \triangleright \nabla(q^{-}) \quad (115) \]
\[ \Pi; \Gamma; \Theta; p; \phi \vdash v: \tau_{1} + \tau_{2} \quad (116) \]

The key premise to prove is that:

\[ \Pi \cdot \langle p \triangleright q \rangle; \phi \vdash \Gamma; \Theta; p; \phi \vdash \langle \text{case } v \text{ of inj}_{1}(x). e_{1} \mid \text{inj}_{2}(x). e_{2} \rangle; \tau \quad (120) \]

which follows from (116) and extending delegations in equations (109) to (112) (Lemma 10). Combining with remaining premises, we have

\[ \Pi; \Gamma; \Theta; p; \phi \vdash \langle \text{case } v \text{ of inj}_{1}(x). e_{1} \mid \text{inj}_{2}(x). e_{2} \rangle; \tau \quad (129) \]

**Case DW-UNITM:** Given \( e = \eta_{\ell} (v \text{ where } (p \triangleright q)) \) and \( e' = (\eta_{\ell} v) \text{ where } (p \triangleright q) \). We have to prove that

\[ \Pi; \Gamma; \Theta; p; \phi \vdash \eta_{\ell} v \text{ where } (p \triangleright q); \ell \text{ says } \tau \quad (117) \]

From DT-UNITM we have:

\[ \Pi; \Gamma; \Theta; p; \phi \vdash v \text{ where } (p \triangleright q); \tau \quad (118) \]

From DT-WHERE, we have:

\[ \Pi; \Gamma; \Theta; p; \phi \vdash \langle p \triangleright q \rangle; (p \triangleright q) \quad (119) \]
\[ \Pi \vdash \phi \triangleright \nabla(q) \quad (120) \]
\[ \Pi \vdash \nabla(p^{-}) \triangleright \nabla(q^{-}) \quad (121) \]
\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; \phi \vdash v: \tau \quad (122) \]

From (121) and DT-UNITM, we have

\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; \phi \vdash \eta_{\ell} v: \ell \text{ says } \tau \quad (123) \]

Combining with other premises ((118) to (121)), we have \( \Pi; \Gamma; \Theta; p; \phi \vdash \eta_{\ell} v \text{ where } (p \triangleright q); \ell \text{ says } \tau \).

**Case DW-BINDM:** Given \( e = \text{bind } x = (v \text{ where } (p \triangleright q)) \text{ in } e \) and \( e' = (\text{bind } x = v \text{ in } e) \text{ where } (p \triangleright q) \). We have to prove that

\[ \Pi; \Gamma; \Theta; p; \phi \vdash (\text{bind } x = v \text{ in } e) \text{ where } (p \triangleright q); \tau \quad (118) \]

From DT-BINDM we have:

\[ \Pi; \Gamma; \Theta; p; \phi \vdash (v \text{ where } (p \triangleright q)); \ell \text{ says } \tau' \quad (124) \]
\[ \Pi; \Gamma, x: \tau'; \Theta; p; \phi \vdash e: \tau \quad (125) \]
\[ \Pi \vdash \phi \vdash x: \tau'; \ell \text{ says } \tau' \quad (126) \]
\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; \phi \vdash v: \ell \text{ says } \tau' \quad (127) \]

From (122) and rule DT-WHERE, we get the following:

\[ \Pi; \Gamma; \Theta; p; \phi \vdash \langle p \triangleright q \rangle; (p \triangleright q) \quad (128) \]
\[ \Pi \vdash \phi \triangleright \nabla(q) \quad (129) \]
\[ \Pi \vdash \nabla(p^{-}) \triangleright \nabla(q^{-}) \quad (130) \]
\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; \phi \vdash v: \ell \text{ says } \tau' \quad (131) \]

We now want to show that \( e' \) is well typed via DT-WHERE. That is, we need the following premises,

\[ \Pi \cdot (p \triangleright q); \Gamma; \Theta; p; \phi \vdash \text{bind } x = v \text{ in } e: \tau \quad (129) \]
\[ \Pi; \Gamma; \Theta; p; \phi \vdash (p \triangleright q); (p \triangleright q) \quad (130) \]
\[ \Pi \vdash \phi \triangleright \nabla(q) \quad (131) \]
\[ \Pi \vdash \nabla(p^{-}) \triangleright \nabla(q^{-}) \quad (132) \]

Extending the delegation context (Lemma 10) in the premises (123), (124) and from (128) we have (129).
We already have (130) from (125); (131) from (126); (132) from (127). Combining, we have

\[ \Pi; \Gamma; \Theta; p; pc \vdash \langle bind \ x = v \ in \ e \rangle \text{ where } \langle p \gg q \rangle : \tau \]

**Case DW-ASSUME:** Given \( e = \text{assume } v \text{ where } \langle a \gg b \rangle \text{ in } e \) and \( e' = \text{assume } v \text{ in } e \text{ where } \langle a \gg b \rangle \). From DT-ASSUME, we have

\[ \Pi; \Gamma; \Theta; p; v \text{ where } \langle a \gg b \rangle \vdash \langle r \gg s \rangle : \]

\[ \Pi \cdot \langle r \gg s \rangle; \Gamma; \Theta; p; pc \vdash e : \tau \]

\[ \Pi \vdash pc \gg \nabla(s) \]

\[ \Pi \vdash \nabla(r^-) \gg \nabla(s^-) \]

Expanding the first premise using DT-WHERE, we have

\[ \Pi; \Gamma; \Theta; p; pc \vdash \langle a \gg b \rangle : \langle a \gg b \rangle \]

\[ \Pi \cdot \langle a \gg b \rangle; \Gamma; \Theta; p; pc \vdash v : \langle r \gg s \rangle \]

\[ \Pi \vdash pc \gg \nabla(b) \]

\[ \Pi \vdash \nabla(a^-) \gg \nabla(b^-) \]

We want to show

\[ \Pi; \Gamma; \Theta; p; pc \vdash \langle a \gg b \rangle : \langle a \gg b \rangle \]

\[ \Pi \cdot \langle a \gg b \rangle; \Gamma; \Theta; p; pc \vdash \text{assume } v \text{ in } e : \tau \]

\[ \Pi \vdash pc \gg \nabla(b) \]

\[ \Pi \vdash \nabla(a^-) \gg \nabla(b^-) \]

Extending delegation context (Lemma 10) in the premises (134), (135), (140) and combining with topmost premise we have the (142). Remaining premises follow from (137), (139) and (140).

**Case DW-TEEG:** Given \( e = \text{runTEE}^r (\hat{v} \text{ where } \langle a \gg b \rangle) \) and \( e' = (\text{runTEE}^l \hat{v}) \text{ where } \langle a \gg b \rangle \). Also given \( \Pi; \Theta; p; pc \vdash e : \text{unit} □ \). From DT-TEEG, we have

\[ \Pi; \emptyset; \Theta'; t; t^- \land pc^- \vdash \hat{v} \text{ where } \langle a \gg b \rangle : \tau \]

\[ \tau \in \{ \text{unit}, \text{unit} □ \} \]

\[ C = \{ \text{ch} | \Theta'(ch) = \text{chan}_{t=pc'} p c' \land \Pi \vdash pc \subseteq pc' \} \]

\[ \Theta |_{C'} = \Theta' \]

\[ \Pi \vdash p \gg pc \]

Expanding the first premise using DT-WHERE, we have

\[ \Pi; \emptyset; \Theta'; t; pc^- \land t^- \vdash \langle a \gg b \rangle : \langle a \gg b \rangle \]

\[ \Pi \cdot \langle a \gg b \rangle; \emptyset; \Theta'; t; pc^- \land t^- \vdash \hat{v} : \tau \]

\[ \Pi \vdash pc \gg \nabla(b) \]

\[ \Pi \vdash \nabla(a^-) \gg \nabla(b^-) \]

\[ \Pi \vdash t \gg pc^- \land t^- \]

To prove \( \Pi; \Gamma; \Theta; p; pc \vdash e' : \text{unit} □ \), we need the following premises:

\[ \Pi; \Gamma; \Theta; p; pc \vdash \langle a \gg b \rangle : \langle a \gg b \rangle \]

\[ \Pi \cdot \langle a \gg b \rangle; \Gamma; \Theta; p; pc \vdash \text{runTEE}^l \hat{v} : \text{unit} □ \]

\[ \Pi \vdash pc \gg \nabla(b) \]

\[ \Pi \vdash \nabla(a^-) \gg \nabla(b^-) \]

\[ \Pi \vdash p \gg pc \]

(155) follows from (149) using DT-UNIT. To prove (156), we need the following premises

\[ \Pi; \emptyset; \Theta'; t; t^- \land pc^- \vdash \hat{v} : \tau \]

\[ C' = \{ \text{ch} | \Theta'(ch) = \text{chan}_{t=pc} p c' \land \text{ch} \cdot \langle a \gg b \rangle \vdash pc \subseteq pc' \} \]

\[ \Theta |_{C'} = \Theta'' \]

\[ \Pi \vdash p \gg pc \]
Invoking lemma that values do not depend on the channel environment (Lemma 12) on premise (151), we have (160). From premises (161) to (163) we thus have the premise (156). Weakening the delegation context in (148), we have (163). Premises (157) and (158) follow from (152) and (153). Hence proved.

**Case DW-HALT:** Given \( e = p, D \vdash (\langle \hat{v} \text{ where } \langle a \bowtie b \rangle \rangle \square) \) and \( e' = p, D \vdash (\hat{v} \square \text{ where } \langle a \bowtie b \rangle \rangle \). Straight forward proof.

**Case BD-STEP:** Given \( e = (e_1 \mid e_2)\) and \( e' = (e'_1 \mid e'_2) \). Also \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash (e_1 \mid e_2) : \tau \). We have to prove

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash (e'_1 \mid e'_2) : \tau
\]

Without loss of generality, let \( i = 1 \). Thus from the premises of BD-STEP, we have \( e_1 \rightarrow e'_1 \) and \( e'_2 = e_2 \). Since sealed values can not take a step, inverting the well-typedness of bracket is only possible through DT-BRACKET-VALUES. From DT-BRACKET, we have

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash e_1 : \tau \\
\Pi; \Gamma; \Theta; \rho; \rho c \vdash e_2 : \tau \\
\Pi \vdash (H^\rho \sqsubseteq \rho c) \subseteq \rho c'
\]

\[
\Pi \vdash H^\rho \leq \tau
\]

Since (164) holds, applying induction to the premise \( e_1 \rightarrow e'_1 \), we have that \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash e'_1 : \tau \). Combining with remaining premises ((165) to (167)) we thus have that \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash (e'_1 \mid e'_2) : \tau \).

**Case BD-APP:** Invalid case as DP-FUN is removed and so a bracket of function values is no longer well-typed.

**Case BD-TAPP:** Invalid case as DP-TFUN is removed and so a bracket of type abstraction values is no longer well-typed.

**Case BD-UNPAIR:** Given \( e = \text{proj}_i ((v_{11}, v_{12}) \mid (v_{21}, v_{22})) \) and \( e' = (v_{1i} \mid v_{2i}) \). Also \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash \text{proj}_i ((v_{1i}, v_{12}) \mid (v_{21}, v_{22})) : \tau_i \). We have to prove

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash (v_{1i} \mid v_{2i}) : \tau_i
\]

From DT-UNPAIR, we have:

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash ((v_{11}, v_{12}) \mid (v_{21}, v_{22})) : \tau_1 \times \tau_2
\]

Since they are already values, they can be inverted using DT-BRACKET-VALUES. This approach is more conservative.

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash (v_{11}, v_{12}) : \tau_1 \times \tau_2 \\
\Pi; \Gamma; \Theta; \rho; \rho c \vdash (v_{21}, v_{22}) : \tau_1 \times \tau_2 \\
\Pi \vdash H^\rho \leq \tau_1 \times \tau_2
\]

From (169), (170) and DT-UNPAIR, we have \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash v_{1i} : \tau_i \) and \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash v_{2i} : \tau_i \) for \( i = \{1, 2\} \).

From (171) and DP-PAIR, we have \( \Pi \vdash H^\rho \leq \tau_i \). Combining with other premises, from DT-BRACKET-VALUES, we have \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash (v_{1i} \mid v_{2i}) : \tau_i \).

**Case BD-BINDM1:** Invalid case as the given bracket expression isn’t well-typed.

**Case BD-BINDM3:** Given \( e = \text{bind } x = (\eta_{\ell_1}, v_1 \mid \eta_{\ell_2}, v_2) \) in \( e \). Here \( e' = ([e]_1[x \mapsto v_1] \mid [e]_2[x \mapsto v_2]) \). Also \( \Pi; \Gamma; \Theta; \rho; \rho c \vdash \text{bind } x = (\eta_{\ell_1}, v_1 \mid \eta_{\ell_2}, v_2) \) in \( e : \tau \). From DT-BINDM, we have

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash (\eta_{\ell_1}, v_1 \mid \eta_{\ell_2}, v_2) : \ell \text{ says } \tau' \\
\Pi; \Gamma; x : \tau'; \Theta; \rho; \rho c \sqcup \ell \vdash e : \tau \\
\Pi \vdash \rho \bowtie \rho c \\
\Pi \vdash H^\rho \leq \ell
\]

From (172) and DT-BRACKET-VALUES, we have

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash \eta_{\ell_1} : \ell \text{ says } \tau' \\
\Pi; \Gamma; \Theta; \rho; \rho c \vdash \eta_{\ell_2} : \ell \text{ says } \tau' \\
\Pi \vdash H^\rho \leq \ell
\]

We have to prove that

\[
\Pi; \Gamma; \Theta; \rho; \rho c \vdash ([e]_1[x \mapsto v_1] \mid [e]_2[x \mapsto v_2]) : \tau
\]
For some \( \widehat{\text{pc}} \) we need the following premises to satisfy DT-\textsc{Bracket}:

\[
\begin{align*}
\Pi; \Gamma; \Theta; p; \widehat{\text{pc}} \vdash e[x \rightarrow v_1]: \tau \\
\Pi; \Gamma; \Theta; p; \widehat{\text{pc}} \vdash e[x \rightarrow v_2]: \tau \\
\Pi \vdash (H^\pi \sqcup \text{pc}) \sqsubseteq \widehat{\text{pc}} \\
\Pi \vdash H^\pi \leq \tau
\end{align*}
\] (179) (180) (181) (182)

A natural choice for \( \widehat{\text{pc}} \) is \( \text{pc} \sqcup \ell \). From (173)

\[
\Pi; \Gamma, x : \tau'; \Theta; p; \widehat{\text{pc}} \vdash e: \tau
\]

Invoking clearance lemma (Lemma 7) on the premise (173) we have that

\[
\Pi \vdash p \gg \text{pc} \sqcup \ell
\]

From the lemma that values can be typed under any \( \text{pc} \) such that they are bound by the place’s authority (Lemma 13), we have

\[
\Pi; \Gamma; \Theta; p; \widehat{\text{pc}} \vdash v_1 : \tau'
\]

Lemma 15 (bracket projection) and Lemma 14 (substitution), we have (179) and (180). From (178) and (174) we have (182).

Thus we have all premises necessary to prove \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e]: \tau' \) and \( \Pi' \mid \text{dom}(\Pi) = \Pi \).

**Case BD-CASE:** Does not occur. Not well-typed.

**Case BD-WHERE :** Does not occur. Not well-typed.

Before we prove type preservation for the distributed system, we show that holes in the evaluation contexts can be typed under a suitable \( \text{pc} \). We express this using the relation

\[
\Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \triangleright \Pi'; \Gamma'; \Theta'; p' ; \text{pc}' \vdash e : \tau'
\]

Figure 24 defines the relation using inference rules. The following lemma proves that every hole is well-typed under some \( \text{pc} \).

**Lemma 19 (Hole has a Suitable \( \text{pc} \)).** If \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \) then \( \exists \Pi', \Gamma', p', \text{pc}', \tau' \) such that \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \triangleright \Pi'; \Gamma'; \Theta'; p' ; \text{pc}' \vdash e : \tau' \) and \( \Pi' \mid \text{dom}(\Pi) = \Pi \).

**Proof.**

The following two lemmas prove that channel environment can be extended for a well-typed expression and context.

**Lemma 20 (Weakening Channel Environment for Expressions).** If \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash e : \tau \) then \( \Pi; \Gamma; \Theta' ; p; \text{pc} \vdash e : \tau \) such that \( \Theta' \mid \text{dom}(\Theta) = \Theta \).

**Proof.** Straightforward induction on the typing judgement.

**Lemma 21 (Weakening Channel Env for Contexts).** If \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \) then for all \( \Theta' \) such that \( \Theta' \mid \text{dom}(\Theta) = \Theta \), \( \Pi; \Gamma; \Theta'; p; \text{pc} \vdash W[e] : \tau \).

**Proof.** Straightforward invocation of Lemma 20.

**Lemma 22 (Hole Substitution).** Let \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \) and \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e] : \tau \triangleright \Pi'; \Gamma'; \Theta'; p' ; \text{pc}' \vdash e : \tau' \). If \( \Pi'; \Gamma'; \Theta'; p' ; \text{pc}' \vdash e' : \tau' \) then \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[e'] : \tau \).

**Proof.** Proof is by induction on the structure of \( W \).

**Lemma 23 (Hole Substitution by Weakening Channel Env).** Let \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash W[\text{spawn} \ @ q (pl, ch_1[pc_1; \tau_1], ch_2[pc_2; \tau_2]), e_1 \ e_2] : \tau \). Then \( \Pi; \Gamma; \Theta'; p; \text{pc} \vdash W[e_2] : \tau \) where

\[
\Theta' = \Theta[ch_1 \mapsto \text{chan}_{p \rightarrow q} pc_1 \tau_1, ch_2 \mapsto \text{chan}_{p \rightarrow q} pc_2 \tau_2]
\]

**Proof.** Proof is by induction on the structure of \( W \). (This is actually \( E \) since \( \text{spawn} \) doesn’t occur inside TEE).

**Case \([\cdot \ : \cdot] \)** Given \( \Pi; \Gamma; \Theta; p; \text{pc} \vdash [\text{spawn} \ @ q (pl, ch_1[pc_1; \tau_1], ch_2[pc_2; \tau_2]), e_1 \ e_2] : \tau \). Inverting using DT-HOLE and then using DT-\text{SPAWN}, we have that:

\[
\Pi; \Gamma; \Theta'; p; \text{pc} \vdash e_2 : \tau
\]

Using DT-HOLE , we have \( \Pi; \Gamma; \Theta'; p; \text{pc} \vdash [e_2] : \tau \).
Case \( e \): Given \( \Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } q \ (pl, ch1[pc1]; \tau1), \ ch2[pc2]; \tau2)] \). \( e1 \) then \( e2 \) \( e: \tau \). Inverting using DT-APP, we have that:

\[
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } e] : \tau' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } \tau1] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } \tau2] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } \text{inj}_1(x)] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{spawn } \text{inj}_2(x)] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{unit}] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{send } e] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{recv } e] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{case } e] : \tau'' \\
\Pi; \Gamma; \Theta; p; pc \vdash W[\text{where } e] : \tau'' \\
\]

Fig. 24: Extracting holes from well-typed contexts
The condition \( \Theta' \mid_{\text{dom}(\Theta')} = \Theta'' \) still holds. Thus from DT-APP, we have
\[
\Pi; \Gamma; \Theta'; p; pc \vdash W[e_2] \ e : \tau
\]

Case \( v \ W \): Straight forward application of induction hypothesis.
Case \( \langle W, e \rangle \): Straight forward application of induction hypothesis.
Case \( \langle v, W \rangle \): Straight forward application of induction hypothesis.
Case \( \text{proj} \ W \): Straight forward application of induction hypothesis.
Case \( \text{inj} \ W \): Straight forward application of induction hypothesis.
Case \( \eta \ W \): Straight forward application of induction hypothesis.
Case \( \text{bind} \ x = W \ in \ e \): Straight forward application of induction hypothesis.
Case \( W \ \tau \): Straight forward application of induction hypothesis.
Case \( \text{assume} \ W \ in \ e \): Straight forward application of induction hypothesis.
Case \( \text{send} \ v \ W \ then \ e_{\text{sk}} \): Straight forward application of induction hypothesis.
Case \( \text{case} \ W \ of \ \text{inj}_1(x). \ e \ | \ \text{inj}_2(x). \ e \): Straight forward application of induction hypothesis.
Case \( W \ where \ v \): Straight forward application of induction hypothesis.
Case \( \text{runTEE}' \ W \): Syntactically not possible.

\( \square \)

**Lemma 24** (High Contexts without TEEs have High Holes). Let \( \Pi; \Gamma; \Theta; p; pc \vdash E[e]: \tau \supset \Pi'; \Gamma'; p'; pc' \vdash e: \tau' \) and \( T \) be the set of TEE principals in \( W \). If \( \exists pc_H \).

1. \( \Pi \vdash H^\pi \cup pc \sqsubseteq pc_H \)
2. \( \Pi; \Gamma; \Theta; p; pc_H \vdash W[e]: \tau \)
3. \( \Pi \vdash H^\pi \leq \tau \)

then \( \exists pc_H' \) such that

1. \( \Pi' \vdash H^\pi \cup pc' \sqsubseteq pc_H' \)
2. \( \Pi'; \Gamma'; \Theta'; p'; pc_H' \vdash e: \tau' \)
3. \( \Pi' \vdash H^\pi \leq \tau' \)

**Proof.** Proof by induction on the relation \( \Pi; \Gamma; \Theta; p; pc \vdash E[e]: \tau \supset \Pi'; \Gamma'; p'; pc' \vdash e: \tau' \).

Case \( W\text{-HOLE} \): Straightforward as \( pc' = pc \). Choose \( pc_H' = pc_H \).
Case \( W\text{-LAM} \): Straightforward application of induction hypothesis.
Case \( W\text{-APP} \): Straightforward application of induction hypothesis.
Case \( W\text{-ARG} \): Straightforward application of induction hypothesis.
Case \( W\text{-PAIR-LEFT} \): Straightforward application of induction hypothesis.
Case \( W\text{-PAIR-RIGHT} \): Straightforward application of induction hypothesis.
Case \( W\text{-PROJ} \): Straightforward application of induction hypothesis.
Case \( W\text{-INJ} \): Straightforward application of induction hypothesis.
Case \( W\text{-UNITM} \): Straightforward application of induction hypothesis.
Case \( W\text{-RUNTEE} \): Does not occur syntactically.
Case \( W\text{-BINDM} \): Straightforward application of induction hypothesis.
Case \( W\text{-ASSUME} \): Straightforward application of induction hypothesis.
Case \( W\text{-SEND} \): Straightforward application of induction hypothesis.
Case \( W\text{-CASE} \): Straightforward application of induction hypothesis.
Case \( W\text{-WHERE} \): Applying induction to the premises, we have

\[
\begin{align*}
\Pi \cdot (r \gg s) \vdash H^\pi \cup pc \sqsubseteq pc_H & \land \\
\Pi \cdot (r \gg s) ; \Gamma; \Theta; p; pc_H \vdash E[e]: \tau & \land \\
\Pi \cdot (r \gg s) \vdash H^\pi \leq \tau & \\
\implies \\
\exists pc_H'. \ 
\Pi' \vdash H^\pi \cup pc' \sqsubseteq pc_H' & \land \\
\Pi'; \Gamma'; \Theta'; p'; pc_H' \vdash e: \tau' & \land \\
\Pi \vdash H^\pi \leq \tau' & \quad (186)
\end{align*}
\]

We are given that

\[
\Pi \vdash H^\pi \cup pc \sqsubseteq pc_H \text{ and} \\
\Pi \vdash H^\pi \leq \tau
\]
Then the following hold.

**Proof** is by induction on the system step relation (⇒).

**PAR-STEP:** Given

$$\Pi_i; \Theta_i; p_i; pc_i \vdash e_i : \tau_i \parallel \cdots \parallel \Pi_m; \Theta_m; p_m; pc_m \vdash e_m : \tau_m$$

From the premises of DT-SYS, we have that $$\Pi_k; \Gamma_k; k; \Theta_k; p_k; pc_k \vdash e_k : \tau_k$$ for every $$k \in \{1 \ldots m\}$$.

Also given, $$\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \implies \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \implies \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle$$.

From the premises, we have that $$p_i, \emptyset \vdash e_i \rightarrow e_i'$$ for some $$i \in \{1 \ldots m\}$$. Thus we have $$n = m$$ and $$e_i = e_k$$ for every $$k \in \{1 \ldots m\} \setminus \{i\}$$.

For $$k = i$$, invoking type preservation for single process (Lemma 18) gives us that $$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash e_i' : \tau_i$$. Thus the new configuration is well-typed by DT-SYS.

**a)** **Proof for remaining results:** Since $$n = m$$, we trivially have the results stated in 3 and 4.

**PAR-SPAWN:** Given

$$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash e_i : \tau_i \parallel \cdots \parallel \Pi_m; \Gamma_m; \Theta_m; p_m; pc_m \vdash e_m : \tau_m$$

From the premises of DT-SYS, we have that $$\Pi_k; \Gamma_k; \Theta_k; p_k; pc_k \vdash e_k : \tau_k$$ for every $$k \in \{1 \ldots m\}$$.

Also given, $$\langle p_1, e_1 \parallel \cdots \parallel \langle p_m, e_m \parallel \cdots \parallel \langle p_m, e_m \rangle \implies \langle p_1, e_1 \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle$$.

From the premises, we have that $$p_i, \emptyset \vdash e_i \rightarrow e_i'$$ for some $$i \in \{1 \ldots m\}$$. Thus we have $$n = m$$ and $$e_i = e_k$$ for every $$k \in \{1 \ldots m\} \setminus \{i\}$$.

For $$k = i$$, invoking type preservation for single process (Lemma 18) gives us that $$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash e_i' : \tau_i$$. It remains to show that

$$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash W[e'] : \tau_i$$

Invoking Lemma 19, we have $$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash W[\text{spawn } @ q (pl, ch_r[pc; \tau]; ch_s[pc; \tau])] \vdash e' \vdash \Pi_i'; \Gamma_i'; \Theta_i'; p_i' ; pc_i' \vdash W[e'] : \tau_i$$.

**b)** **Proof for remaining results:** Since $$n = m + 1$$, we have that $$\Pi_n = \Pi_i$$ and so $$\Pi_n \subseteq \Pi_i$$ (Lemma ??). Thus we have the results stated in condition 3 and 4.

**PAR-SEND-RECV:** Here $$n = m$$. Without loss of generality, let $$(e_i, e_j) = (W[\text{send } \nu \text{ as } x in e], W[\text{recv } \nu \text{ as } x in e])$$.

Then $$(e_i', e_j') = (W[e_{\nu}; \emptyset], W[e_{\nu}; \emptyset])$$.

We thus have $$\Pi_k; \Gamma_k; \Theta_k; p_k; pc_k \vdash e_k \vdash \tau_k$$ where $$\Theta_k = \Theta_k$$ and $$e_k = e_k$$ for all $$k \in \{1 \ldots m\} \setminus \{i, j\}$$.

Given $$\Pi_i; \Gamma_i; e_i; pc_i \vdash W[\text{send } v \text{ as } x \text{ in } e] : \tau_i$$ and $$\Pi_j; \Gamma_j; e_j; pc_j \vdash W[\text{recv } \nu \text{ as } x \text{ in } e] : \tau_j$$.

Since holes can be typed under some $$pc_i'$$ (Lemma 19), we have that $$\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash W[\text{send } \nu \text{ as } x \text{ in } e] : \tau_i$$.

From DT-SENDDEL, we have that $$\Pi_i; \emptyset; \emptyset; p_i; pc_i \vdash v : \tau$$ for some $$\Pi_i'$$ such that $$\Pi_i' \subseteq \Pi_i$$ and $$\Pi_i' \vdash \Pi_i' \vdash \Pi_i' : p_i' \vdash v : \tau$$.

Since non-bracket values sent over a channel can be typed on other places such that $$\Pi_i' \vdash v' \vdash p_i'$$ (Lemma 17), we have $$\Pi_i' : \emptyset; \emptyset; p_i' ; pc_i' \vdash v : \tau$$.

If $$v = (v_1 \parallel v_2)$$, a bracket value, then from DT-BRACKET-VALUES, we have that $$\Pi_i' \vdash H^\pi \subseteq \tau$$. From the given conditions on the upward closure of $$H^\pi$$, we have that $$\Pi_i' \vdash H^\pi \subseteq \tau$$. From the given conditions on the upward closure of $$H^\pi$$, we have that $$\Pi_i' \vdash H^\pi \subseteq \tau$$.

This implies $$\Pi_i' ; \Gamma_i; \Theta_i; p_i; pc_i \vdash v : \tau$$ (weakening delegation and variable contexts, channel environment by invoking Lemma 12, Lemma 10).
From DT-RECV, we have $\Pi'_j; \Gamma'_j; x : \tau; \Theta'_j; p'_j; p\gamma'_j \vdash e_n : \tau'_j$. Applying PC reduction (Lemma 11), we have that $\Pi'_j; \Gamma'_j; x : \tau; \Theta'_j; p'_j; p\gamma'_j \vdash e_n : \tau'_j$. Invoking variable substitution (Lemma 14), we have that $\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; p\gamma'_j \vdash e_n[x \rightarrow v]; \Box : \tau'_j$. Invoking hole substitution (Lemma 22), on $\Pi_j; \Gamma_j; \Theta_j; p_j; p\gamma_j \vdash W[\text{recv} \nu \text{ as } x \in e; \Box] : \tau_j$, we have

$$\Pi_j; \Gamma_j; \Theta_j; p_j; p\gamma_j \vdash e'_j : \tau_j$$

From DT-SENDDEL, we also have that $\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; p\gamma'_j \vdash e_n : \nu$. Applying PC reduction (Lemma 11), we have that $\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; p\gamma'_j \vdash e_n : \nu$. Invoking hole substitution (Lemma 22), on $\Pi_j; \Gamma_j; \Theta_j; p_j; p\gamma_j \vdash W[\text{send} \nu \text{ then } e_n] : \tau_i$, we have

$$\Pi_i; \Gamma_i; \Theta_i; p_i; p\gamma_i \vdash e'_j : \tau_i$$

c) **Proof for remaining results:** Since $n = m$, we trivially have the results stated in 3 and 4.

**BD-PAR-STEP-L:** Given

$$(p_1, e_1) \parallel \cdots \parallel (p_i, (e_{i_1} \mid e_{i_2})) \parallel \cdots \parallel (p_m, e_m) \implies (p_1, e'_1) \parallel \cdots \parallel (p_i, e'_i) \parallel \cdots \parallel (p_n, e'_n)$$

Let $e_i = (e_{i_1} \mid e_{i_2})$. From the premises of BD-PAR-STEP, we have

$$(p_1, e_1) \parallel \cdots \parallel (p_i, e_{i_1}) \parallel \cdots \parallel (p_m, e_m) \implies (p_1, e'_1) \parallel \cdots \parallel (p_i, e'_i) \parallel \cdots \parallel (p_n, e'_n)$$

Also from the remaining premises we have

$$i \in \{1 \ldots m\} \quad (187)$$

$$e'_i = (e'_{i_1} \mid |e_{i_2}|) \quad (188)$$

$$j = \begin{cases} 
  n & \text{if } n \neq m \\
  \in \{1 \ldots m\} & \text{otherwise} 
\end{cases} \quad (189)$$

$$e'_j = \begin{cases} 
  e'_i & \text{if } n = m \text{ and } j = i \\
  e'_{i_1} & \text{if } n \neq m \text{ and } j \neq i \\
  (e'_{j_1} \mid e_{j_2}) & \text{if } n = m, j \neq i \text{ and } e_j = (e_{j_1} \mid e_{j_2}) \\
  W[(e'_{j_1} \mid e)] & \text{if } n = m, j \neq i \text{ and } e_j = W[e] 
\end{cases} \quad (190)$$

From DT-SYS, we have that

$$\Pi_i; \Gamma_i; \Theta_i; p_i; p\gamma_i \vdash e_i : \tau_i$$

$$\Pi_j; \Gamma_j; \Theta_j; p_j; p\gamma_j \vdash e_j : \tau_j$$

We have four cases.

**Case $n = m$ and $j = i$:** Trivial case. Invoking type preservation for single process (Lemma 18), we get the conclusion.

**d) Proof for remaining results:** Since $n = m$, we trivially have the results stated in 3 and 4.

**Case $n \neq m$ and $j \neq i$:** $e_i$ is a spawn expression. That is, we have

$$e_i = (W_i[\text{spawn} @ q (pl, ch_r[p\gamma_i; \tau_i], ch_s[p\gamma_i; \tau_i])). e \text{ then } e_n] \mid e_{i_2})$$

$$e'_i = (W_i[e_n] \mid e_{i_2})$$

$$e'_j = e'_n = (e \mid \bullet)$$

$$\Pi_i; \Gamma_i; \Theta_i; p_i; p\gamma_i \vdash e_i : \tau_i$$

(192)

Note that TEE’s do not have any spawn statement. Thus $W_i$ is actually $E_i$. That is,

$$e_i = (E_i[\text{spawn} @ q (pl, ch_r[p\gamma_i; \tau_i], ch_s[p\gamma_i; \tau_i])). e \text{ then } e_n] \mid e_{i_2})$$

$$e'_i = (E_i[e_n] \mid e_{i_2})$$

$$e'_j = e'_n = (e \mid \bullet)$$

$$\Pi_i; \Gamma_i; \Theta_i; p_i; p\gamma_i \vdash e_i : \tau_i$$

(194)
From the DT-BRACKET (since it is not a sealed bracket, we cannot invert using DT-BRACKET-VALUES), we have that \( e_i \) is protected for some \( pc'_i \) and so:

\[
\Pi_i, \Gamma'_i; \Theta'_i; p_i; pc'_i \vdash W_i[\text{spawn } @ q (pl, ch_r[pc_r; \tau_r], ch_s[pc_s; \tau_s], e) \text{ then } e_n]; \tau_i \leq \tau_i
\]  

(195)

\[
\Pi_i, \Gamma_i; \Theta_i; p_i; pc'_i \vdash W_i[\text{spawn } @ q (pl, ch_r[pc_r; \tau_r], ch_s[pc_s; \tau_s], e) \text{ then } e_n]; \tau_i
\]  

(196)

\[
\Pi_i, \Gamma_i; \Theta_i; p_i; pc'_i \vdash e_i; \tau_i \implies \Pi'_i, \Gamma'_i; \Theta'_i; p'_i; pc''_i \vdash \text{spawn } @ q (pl, ch_r[pc_r; \tau_r], ch_s[pc_s; \tau_s], e) \text{ then } e_n; \tau_i
\]  

(197)

\[
\Pi_i \vdash pc''_i \subseteq pc'_i
\]  

(198)

Invoking Lemma 19 on (196), we have \( \Pi_i, \Gamma_i; \Theta_i; p_i; pc'_i \vdash e_i; \tau_i \implies \Pi'_i, \Gamma'_i; \Theta'_i; p'_i; pc''_i \vdash \text{spawn } @ q (pl, ch_r[pc_r; \tau_r], ch_s[pc_s; \tau_s], e) \text{ then } e_n; \tau_i \). Inverting the above expression using DT-SPAWN, we have:

\[
\Pi'_i, \Gamma'_i; \Theta'_i; p_i; pc'_i \vdash e_i; \tau_i
\]  

(199)

\[
\Pi'_i, \Gamma'_i; \Theta'_i; p_i; pc'_i \vdash pc''_i
\]  

(200)

\[
\Pi_i \vdash pc''_i \leq pc'_i
\]  

(201)

\[
\Pi_i \vdash pc''_i \leq pc'_i
\]  

(202)

\[
\Pi_i \vdash pc''_i \leq \tau_i
\]  

(203)

Invoking hole substitution (Lemma 23) with (200) we have

\[
\Pi_i; \Gamma_i; \Theta'_i; p_i; pc'_i \vdash W_i[e_n]; \tau_i
\]  

where \( \Theta'_i = \Theta_i[ch_r \mapsto \text{chan}_{p'_i \rightarrow q} pc_r, \tau_r, ch_s \mapsto \text{chan}_{p'_i \rightarrow q} pc_s, \tau_s] \). Weakening channel environment on (210) using Lemma 20, we have Thus, from DT-BRACKET, we have that

\[
\Pi_i; \Gamma_i; \Theta'_i; p_i; pc_i \vdash W_i[e_n]; e_i; \tau_i
\]  

We still need to show that for some \( \Pi_n, \Gamma_n, \Theta_n, pc_n \) and \( \tau_n \), the following holds.

\[
\Pi_n; \Gamma_n; \Theta_n; q; pc_n \vdash (e | \bullet); \tau_n
\]  

We need the following premises to prove that \( e'_i \) is well-typed.

\[
\Pi'_i \vdash (H^\pi \sqcup pc''_i) \subseteq pc''_H
\]  

(204)

\[
\Pi'_i \vdash \emptyset; \Theta'_i; q; pc''_H \vdash e; \tau_n
\]  

(205)

\[
\Pi'_i \vdash H^\pi \leq \tau_n
\]  

(206)

Since high contexts (without TEE’s) have high holes, applying Lemma 24 on (195), we have that

\[
\exists pc''_H. \Pi'_i \vdash H^\pi \sqcup pc''_H \subseteq pc''_H
\]  

(207)

\[
\Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc''_H \vdash \text{spawn } @ q (pl, ch_r[pc_r; \tau_r], ch_s[pc_s; \tau_s], e) \text{ then } e_n; \tau_i
\]  

(208)

\[
\Pi'_i \vdash H^\pi \leq \tau
\]  

(209)

Inverting (208) using DT-SPAWN we have that for some \( pc''_H \):

\[
\Pi'_i; \emptyset; \Theta'; q; pc''_H \vdash e; \tau_n
\]  

(210)

\[
\Pi \vdash pc''_H \subseteq pc''_H
\]  

(211)

Premise (204) follows from (207) and (211). Premise (206) follows from (203) Premise (205) follows from (210). We thus have that \( (e | \bullet) \) is well-typed.

The new configuration is also well-formed as the channels inherit parent delegation contexts.
e) Proof for remaining results: Since \( n = m + 1 \), we have that \( \Pi_n = \Pi_i \) and so \( \bar{\Pi}_n \subseteq \bar{\Pi}_i \) (Lemma ??). Thus we have the results stated in condition 3 and 4.

Case \( j \neq i \), \( n = m \) and \( e_j = (e_{j1} \mid e_{j2}) \): If \( e_j \) is a bracket value, then it must be the case that the left projections are senddel/recv expressions. Without loss of generality, assume the following:

\[
e_i = (W_i[send \nu v \text{ then } e_{sk}] \mid e_{i2})
\]

\[
e_j = (W_j[recv \nu as x \text{ in } e_n] \mid e_{j2})
\]

\[
e_i' = (W_i[e_{sk}] \mid e_{i2})
\]

\[
e_j' = (W_j[e_n[x \mapsto v]] \mid e_{j2})
\]

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i
\]

\[
\Pi; \Gamma; \Theta; p_j; pc_j \vdash e_j; \tau_i
\]

We have to show that:

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i \parallel \cdots \parallel \Pi; \Gamma; \Theta; p_j; pc_j \vdash e_j; \tau_i
\]

Inverting (216) and (217) using DT-BRACKET (since it is not a sealed bracket, we cannot invert using DT-BRACKET-VALUES), we have that

\[
\Pi; \Gamma; \Theta; p_i; pc_i' \vdash W_i[send \nu v \text{ then } e_{sk}] \parallel \Pi; \Gamma; \Theta; p_j; pc_j' \vdash e_{i2} \parallel \Pi; \Gamma; \Theta; p_j; pc_j' \vdash e_{j2} \parallel \Pi; \Gamma; \Theta; p_j; pc_j' \vdash e_j; \tau_i
\]

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i \parallel \Pi; \Gamma; \Theta; p_j; pc_j \vdash e_j; \tau_i
\]

We have to show that:

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i \parallel \cdots \parallel \Pi; \Gamma; \Theta; p_j; pc_j \vdash e_j; \tau_i
\]

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i \parallel \Pi; \Gamma; \Theta; p_j; pc_j' \vdash e_{j2} \parallel \Pi; \Gamma; \Theta; p_j; pc_j' \vdash e_j; \tau_i
\]

\[
\Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i; \tau_i \parallel \Pi; \Gamma; \Theta; p_j; pc_j \vdash e_j; \tau_i
\]

f) Proof for remaining results: Since \( n = m \), we trivially have the results stated in 3 and 4.

Case \( n = m \) and \( e_j = W[e] \): Applying induction hypothesis to the premise of BD-PAR-STEP-L, we have that \( \Pi; \Gamma; i; p_i; pc_i' \vdash e_{i1} \parallel \tau_i \). We already have \( \Pi; \Gamma; i; p_i; pc_i' \vdash e_{i2} \parallel \tau_i \) and so from DT-BRACKET (proving for a more restrictive case) we have

\[
\Pi; \Gamma; i; p_i; pc_i \vdash e_i' \parallel \tau_i
\]

Since both \( e_i \) and \( e_j \) take a step, it must be the case that they are senddel and recv pairs. We then have two more cases.
\textbf{Case $e_j = \text{send } \nu \mathbin{v}$ then $e_{ak}$}: Without loss of generality assume that $e_j = \text{send } \nu \mathbin{v}$ then $e_{ak}$. We then have the following:

\begin{align*}
e_i &= (W_i[\text{recv } \nu \mathbin{as } x \in e_n] \mid e_i) \\
\tau_i &= (W_i[e_n[x \mapsto v];[\square]] \mid e_{i2})
\end{align*}

\begin{align*}
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i &\vdash (W_i[\text{recv } \nu \mathbin{as } x \in e_n] \mid e_{i2}): \tau_i \\
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i &\vdash (W_i[e_n[x \mapsto v];[\square]] \mid e_{i2}): \tau_i \\
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j &\vdash W_j[(\text{send } \nu \mathbin{v} \mathbin{then } e_{ak});[\square]]: \tau_j
\end{align*}

We have to show that

\[ \Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash W_j[(\text{send } \nu \mathbin{v} \mathbin{then } e_{ak});[\square]]: \tau_j \]

From (230) and DT-BRACKET (since it is not a sealed bracket, we cannot invert using DT-BRACKET-VALUES), we have the following:

\begin{align}
\Pi_i \vdash (H^\pi \sqsubseteq pc_i) &\subseteq pc_i' \\
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i &\vdash W_i[\text{recv } \nu \mathbin{as } x \in e_n]; \tau_i \\
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i &\vdash \text{recv } \nu \mathbin{as } x \in e_n; \tau_i
\end{align}

Since holes can be typed under an appropriate $pc_i$, invoking Lemma 19 on (235) we have

\begin{align}
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i &\vdash W_i[\text{recv } \nu \mathbin{as } x \in e_n]; \tau_i \\
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i'' &\vdash \text{recv } \nu \mathbin{as } x \in e_n; \tau_i
\end{align}

There are two cases depending on whether or not $W_i$ includes runTEE. That is $W_i = E_i$ or $W_i \neq E_i$.

\textbf{Case $W_i = E_i$}: Invoking Lemma 24 on (238) we have that

\begin{align}
\Pi_i' \vdash (H^\pi \sqsubseteq pc_i) &\subseteq pc_i'' \\
\Pi_i'; \Gamma_i'; \Theta_i'; p_i'; pc_i'' &\vdash \text{recv } \nu \mathbin{as } x \in e_n; \tau_i
\end{align}

Inverting (240) using DT-RECV we have

\begin{align}
\Pi_i' \vdash \text{recv } \nu \mathbin{as } x \in e_n; \tau_i' &\subseteq pc_i'' \\
\Pi_i'; \Gamma_i'; \Theta_i'; p_i'; pc_i'' &\vdash \text{recv } \nu \mathbin{as } x \in e_n; \tau_i'
\end{align}

Applying R-TRANS to (239) and (245), we have

\begin{align}
\Pi_i' \vdash H^\pi &\subseteq pc_i''
\end{align}

\textbf{Case $W_i \neq E_i$}: Consider the premises in (235) and (234). If $W_i$ includes runTEE and $W_i[\text{recv } \nu \mathbin{as } x \in e_n]$ is well-typed under $pc_i'$, then $\exists pc_i''$ (since nested TEEs aren’t allowed and from Lemma 24 we have monotonic $pc$) such that

\begin{align}
\Pi_i'' \vdash H^\pi &\subseteq pc_i'' \\
\Pi_i''; \Gamma_i''; \Theta_i''; p_i; pc_i'' &\vdash \text{runTEE}[E_i[\text{recv } \nu \mathbin{as } x \in e_n]]; \text{unit}$
\end{align}

Recall the premise from DT-TEEG

\[ C = \{ ch \mid ch \in \Theta(ch) = \text{chan}_{t=p} pc' \mathbin{\tau } \mathbin{\land } \Pi_i''' \vdash pc_i'' \subseteq pc' \} \]
for some \( \Pi''_i \) such that \( \Pi''_i \mid_{\text{dom}(\Pi'_i)} = \Pi''_i \). Since \( E_i \) cannot contain any \texttt{runTEE} or \texttt{spawn}, we have \( \Theta''_i |_{\text{c}} = \Theta'_i \). This implies

\[
\Pi''_i \vdash H^\pi \sqsubseteq \Pi'_{\text{ch}} \tag{254}
\]

Let us now focus on \( j \)-th node. Since holes can be typed under appropriate \( pc \), invoking Lemma 19 on (233), we have

\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash W_j[\text{send } \nu \ then \ e_{\text{sn}}]; \ \tau_j \triangleright \tag{255}
\]

From (255) and DT-SEND_DEL we have

\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{j_s} \vdash e: \text{chan } p'_j \rightarrow p'_j, pc_{e} \square \tau \tag{256}
\]

\[
\Pi'_j; \emptyset; \emptyset; p'_j; pc_{j_s} \vdash e: \tau \tag{257}
\]

\[
\Pi'_j \vdash pc_{j_s} \sqsubseteq pc_{e} \tag{258}
\]

\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc'_{j_s} \vdash e_{\text{sn}}: \tau'_j \tag{259}
\]

\[
\tau'_j = \{ \tau'_j, \emptyset \} \tag{260}
\]

\[
\Pi'_j \vdash pc_{e} \sqsubseteq pc'_{j_s} \tag{261}
\]

\[
\Pi'_j \vdash \Pi'_{j} \sqsubseteq \Pi'_j \tag{262}
\]

\[
\Pi'_j \vdash p'_j \triangleright p''_j \tag{263}
\]

\[
\Pi'_j \vdash pc'_{j_s} \tag{264}
\]

\[
\Pi'_j \vdash p'_j \triangleright pc'_{j_s} \tag{265}
\]

\[
\Pi'_j \vdash \Pi'_{j} \sqsubseteq pc_{e} \tag{266}
\]

To show that \( e'_{j} \) is well-typed, we first need to find a \( pc'_{j_s} \) such that DT-BRACKET holds for \( e_{\text{sn}: \emptyset} \mid \text{send } \nu \ then \ e_{\text{sn}} \). That is, we need the following premises of the DT-BRACKET to type

\[
\Pi'_j; \Gamma'_j; \emptyset; \emptyset; p'_j; pc_{j_s} \vdash (e_{\text{sn}}: \emptyset \mid \text{send } \nu \ then \ e_{\text{sn}}); \ \tau'_j \tag{271}
\]

\[
\Pi'_j \vdash H^\pi \sqsubseteq \Pi'_{\text{ch}} \tag{270}
\]

Let \( pc'_{j_s} = pc_{e} \). From premises (251), (254) and given conditions on the upward closure of \( H^\pi \) we have

\[
\Pi'_j \vdash H^\pi \sqsubseteq pc_{e} \tag{270}
\]

Since we have that \( \Pi'_j \vdash p'_j \sqsubseteq pc_{e} \) (from (268)), combining with the premise (240), we have (268). Premise (269) holds because the label on the channel is bounded by the place \( p'_j \) authority and premise (270) follows from \( \Pi'_j \vdash pc'_{j_s} \leq \tau'_j \). We then have

\[
\Pi'_j; \Gamma'_j; \emptyset; \emptyset; p'_j; pc_{j_s} \vdash (e_{\text{sn}}: \emptyset \mid \text{send } \nu \ then \ e_{\text{sn}}); \ \tau'_j \tag{271}
\]

Invoking hole substitution (Lemma 22) on (233) using the above premise and (255) we have the required proof.

\( g \) \textbf{Proof for remaining results:} Since \( n = m \), we trivially have the results stated in 3 and 4.

\textbf{Case} \( e_j = \text{recv } \nu \ as \ x \ in \ e \): Given \( e_j = \text{recv } \nu \ as \ x \ in \ e \). We then have the following:

\[
e_i = (W_i[\text{send } \nu \ then \ e_{\text{sn}}] \mid e_{i2}) \tag{272}
\]

\[
e'_i = (W_i[e_{\text{sn}}: \emptyset] \mid e_{i2}) \tag{273}
\]

\[
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash e_i: \tau_i \tag{274}
\]

\[
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash e'_i: \tau_i \tag{275}
\]

\[
e'_j = W_j[(e_{\text{sn}}[x \mapsto v]: \emptyset \mid \text{recv } \nu \ as \ x \ in \ e_{\text{sn}})] \tag{274}
\]

\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash e_j: \tau_j \tag{277}
\]

We have to show that

\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash e'_j: \tau_j \tag{277}
\]
From (274) and DT-BRACKET (since it is not a sealed bracket, we cannot invert using DT-BRACKET-VALUES), we have the following:

\[ \Pi_i \vdash \langle H \Delta \cup pc_i \rangle \subseteq pc'_i \]  
(278)

\[ \Pi_i; \Gamma_i; \Theta_i; p_i; pc'_i \vdash W_i[send \nu \ v \ then \ e_{\text{ex}}]; \tau_i \]  
(279)

\[ \Pi_i; \Gamma_i; \Theta_i; p_i; pc'_i \vdash e_{\text{ex}}; \tau_i \]  
(280)

\[ \Pi_i \vdash H^\pi \leq \tau_i \]  
(281)

Since holes can be typed under an appropriate \( pc_{i_5} \), invoking Lemma 19 on (279) we have

\[ \Pi_i; \Gamma_i; \Theta_i; p_i; pc'_i \vdash W_i[send \nu \ v \ then \ e_{\text{ex}}]; \tau_i \]  
(282)

From DT-SENDDEL we have

\[ \Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc_{i_5} \vdash \nu: \text{chan}_p \to p_i \vdash pc_{ch} \tau \]  
(283)

\[ \Pi'_i; \Theta; p'_i; pc_{i_5} \vdash v: \tau \]  
(284)

\[ \Pi'_i \vdash pc_{i_5} \subseteq pc_{ch} \]  
(285)

\[ \Pi_i; \Gamma; \Theta; p; pc'_{i_5} \vdash e_{\text{ex}}; \tau'_i \]  
(286)

\[ \tau'_i = \{ i'_1, i'_2 \} \]  
(287)

\[ \Pi'_i \vdash pc_{i_5} \subseteq pc'_{i_5} \]  
(288)

\[ \Pi'_i \vdash pc_{ch} \subseteq pc''_{i_5} \]  
(289)

\[ \Pi'_i \vdash pc''_{i_5} \leq \tau'_i \]  
(290)

\[ \Pi''_i \subseteq \Pi'_i \]  
(291)

\[ \Pi''_i \vdash p'_i \equiv pc_{i_5} \]  
(292)

\[ \Pi''_i \vdash p'_i \equiv pc_{is} \]  
(293)

\[ \Pi''_i \vdash p'_i \equiv pc_{ch} \]  
(294)

There are two cases depending on whether or not \( W_i \) includes runTEE. That is \( W_i = E_i \) or \( W_i \neq E_i \).

**Case** \( W_i = E_i \): Invoking Lemma 24 on (282) we have that

\[ \Pi'_i \vdash \langle H \Delta \cup pc_{i_5} \rangle \subseteq pc'_{i_5} \]  
(295)

\[ \Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc'_{i_5} \vdash \text{send} \nu \ v \ then \ e_{\text{ex}}; \tau'_i \]  
(296)

\[ \Pi'_i \vdash H^\pi \leq \tau'_i \]  
(297)

Applying R-TRANS to (296) and (295), we have

\[ \Pi'_i \vdash H^\pi \subseteq pc_{ch} \]  
(298)

**Case** \( W_i \neq E_i \): Consider the premises in (279) and (278). If \( W_i \) includes runTEE and \( W_i[send \nu \ v \ then \ e_{\text{ex}}] \) is well-typed under \( pc'_i \), then \( \exists pc''_i \) (nested TEEs aren’t allowed and for non-TEEs we have monotonic \( pc \) from Lemma ??) such that

\[ \Pi''_i \vdash H^\pi \subseteq pc''_i \]  
(299)

\[ \Pi''_i; \Gamma''_i; \Theta''_i; p_i; pc''_i \vdash \text{runTEE}[E_i[send \nu \ v \ then \ e_{\text{ex}}]]; \text{unit} \]  
(300)

\[ \Pi''_i \vdash H^\pi \leq \text{unit} \]  
(301)

Recall the premise from DT-TEEG

\[ C = \{ c_i | c_i \in \Theta(ch) = \text{chan}_{p \to p} \ pc' \tau \wedge \Pi''_i \vdash pc''_i \subseteq pc' \} \]

Since \( E_i \) cannot contain any runTEE or spawn, we have \( \Theta''_i |_C = \Theta'_i \). This implies

\[ \Pi''_i \vdash H^\pi \subseteq pc_{ch} \]  
(302)
Let us now focus on \( j^{th} \) node. Since holes can be typed under an appropriate \( pc_{jr} \), invoking Lemma 19 on (277), we have
\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash W_j[\text{recv } \nu as x in e_n]: \tau_j >
\]
Inverting (303) using DT-RECEIVE we have
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{jr} \vdash \text{recv } \nu as x in e_n: \tilde{\tau}_j \square
\]
To show that \( e'_j \) is well-typed, we first need to find a \( pc'_{jr} \) such that DT-BRACKET holds for \((e_n[x \mapsto v]) | \text{recv } \nu as x in e_n)\). That is, we need the following premises to get
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{jr} \vdash (e_n[x \mapsto v]) | \text{recv } \nu as x in e_n): \tau_r.
\]
\[
\Pi'_j \vdash (H^\pi \sqcup pc_{jr}) \sqsubseteq pc'_{jr} \tag{306}
\]
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc'_{jr} \vdash \text{recv } \nu as x in e_n: \tilde{\tau}_r \square \tag{307}
\]
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc'_{jr} \vdash e_n[x \mapsto v]: \tilde{\tau}_r \square \tag{308}
\]
\[
\Pi'_j \vdash H^\pi \leq \tilde{\tau}_r \square \tag{309}
\]
Let \( pc'_{jr} = pc_{ch} \). From premises (298), (302) and given input conditions on the upward closure of \( H^\pi \) we have
\[
\Pi'_j \vdash H^\pi \sqsubseteq pc_{ch}
\]
Premise (306) follows from (305). We prove (307) as shown below: Inverting (277) using DT-RECV \( \omega \). Summarized as inference rule below.
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{ch} \vdash ch; chan_{p_{\omega} = q} pc_{ch} \tau \quad \Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc'_{jr} \vdash e_n: \tau'_r
\]
\[
\tau'_r = \{\tilde{\tau}_r, \tilde{\tau}_r \square\}
\]
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{ch} \vdash \text{recv } \nu as x in e_n: \tilde{\tau}_r \square \tag{307}
\]
To show (308), we first have to show that
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc'_{jr} \vdash \text{recv } \nu as x in e_n: \tilde{\tau}_r \square
\]
For this, we invoke the lemma that states that values sent are well-typed under places that satisfy the clearance requirement (Lemma 17) to (284). Premise (308) now follows from invoking substitution lemma on the premises of (307). We still need to show that \( \Pi'_j \vdash H^\pi \leq \tilde{\tau}_r \square \) holds. Consider the above premises: \( \Pi'_j \vdash pc_{ch} \sqsubseteq pc'_{jr} \) and \( \Pi'_j \vdash pc'_{jr} \leq \tilde{\tau}_r \). From these (306) and DP-HALT, the result follows. We now have all the premises to show
\[
\Pi'_j; \Gamma'_j; \Theta'_j; p'_j; pc_{jr} \vdash (e_n[x \mapsto v]) | \text{recv } \nu as x in e_n): \tilde{\tau}_r \square
\]
Invoking hole substitution (Lemma 22) on (277) using the above premise we have the full proof.
h) Proof for remaining results: Since \( n = m \), we trivially have the results stated in 3 and 4.
\[\]
Case BD-PAR-SPAWN : Similar to the spawn sub case in the proof of BD-PAR-STEP (i.e., Case \( n \neq m \) and \( j \neq i \)).
Case BD-PAR-SEND-RECV-L: Given
\[
e_i = W_i(\text{send } \nu v then e_{sk} | e) \tag{311}
\]
\[
e_j = W_j(\text{recv } \nu as x in e_n | e') \tag{312}
\]
\[
e'_i = W_i(\text{send } \nu v then e_{sk} | e') \tag{313}
\]
\[
e'_j = W_j(\text{recv } \nu as x in e_n | e') \tag{314}
\]
Also given that \( e_i \) and \( e_j \) are well-typed.
\[
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash W_i(\text{send } \nu v then e_{sk} | e): \tau_i \tag{311}
\]
Invoking Lemma 19 we have that \( (\text{send } \nu v then e_{sk}; \Box | e) \) is well typed under some \( pc_{sk} \). That is,
\[
\Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc'_i \vdash \text{send } \nu v then e_{sk}; \Box | e: \tilde{\tau}'_i \square \quad \Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc'_i \vdash e: \tilde{\tau}'_i \square \quad \Pi'_i \vdash H^\pi \leq \tilde{\tau}'_i \square
\]
\[
\Pi'_i; \Gamma'_i; \Theta'_i; p'_i; pc_{sk} \vdash (\text{send } \nu v then e_{sk} | e): \tilde{\tau}'_i \square
\]
We already have $H^\pi \leq \tau'_s \square$. Applying PC reduction (Lemma 11 to the fourth premise, we have $\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc'_s \vdash e_{\text{sc}}; \square \vdash \tau'_s$. So

$$\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc'_s \vdash (e_{\text{sc}}; \square \mid e') : \tau'_s \square$$

Depending on whether or not $e_j$ is bracketed, we have two more cases.

**Case** $e_j = (W_j[\text{recv } \nu \text{ as } x \text{ in } e_\kappa] \mid e')$:

We are given that:

$$e_j = W_j[(\text{recv } \nu \text{ as } x \text{ in } e_\kappa] \mid e')]$$ (325)
$$e'_j = W_j[(e_\kappa[x \mapsto v] \mid e')]$$ (326)

Also given

$$\Pi; \Gamma; \Theta; p_j; pc_j \vdash W_j[(\text{recv } \nu \text{ as } x \text{ in } e_\kappa] \mid e') : \tau_j$$

Invoking Lemma 19 we have that $(\text{recv } \nu \text{ as } x \text{ in } e_\kappa] \mid e')$ is well typed for some $pc_r$. That is,

$$\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc'_s ; (\text{recv } \nu \text{ as } x \text{ in } e_\kappa] \mid e') : \tau'_r \square$$

So, from the second premise (following DT-RECEIVE) we have:

$$\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc'_s \vdash \text{ch}: \text{chan}_{p'_j \tau'} p'_i pc_{ch} \tau$$ (327)

$$\Pi'_i; \Gamma'_i; x : \tau; \Theta'_j; p'_j; pc'_r \vdash e_\kappa : \tau'_r$$ (328)

$$\tau'_r = \{\tau'_s, \tau'_\square\}$$ (329)

$$\Pi'_i \vdash pc'_r \subseteq pc_{ch}$$ (330)

$$\Pi'_i \vdash pc_{ch} \subseteq pc'_{\tau}$$ (331)

$$\Pi'_i \vdash pc'_{\tau} \leq \tau'_r$$ (332)

$$\Pi'_i \vdash p'_j \gg pc'_r$$ (333)

$$\Pi'_i \vdash p'_j \gg pc_{ch}$$ (334)

We now prove the recv part. First we have to prove that the value $v$ is well-typed under the new context. From premise (315), we have that $v$ can be typed under empty variable and channel environment, and a delegation context that enables a place to act for the pc.

Applying Lemma 17 to (315) using (333), it then follows that

$$\Pi'_i; \emptyset; \emptyset; p'_j; pc''_r \vdash v : \tau$$

Weakening channel environment for values (Lemma 12) and variable context in the above judgment, we have:

$$\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc''_r \vdash v : \tau$$

From variable substitution lemma (Lemma 14) we have

$$\Pi'_i; \Gamma'_i; \Theta'_j; p'_j; pc''_r \vdash e_\kappa[x \mapsto v] : \tau'_r$$
Thus we have
\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_r \vdash (e_k[x \mapsto v]; | e'); \ \hat{\tau}_j
\]

Invoking hole substitution (Lemma 22), we thus have
\[
\Pi_i; \Gamma_i; \Theta_i; p_i; pc_i \vdash W_i[(e_{sa}; | e)]; \ \tau_i
\]
and
\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash W_j[(e_k[x \mapsto v]; | e')]; \ \tau_j
\]

Case \( e_j = W_j[\text{recv } \nu \text{ as } x \text{ in } e_k] \):

\[
e_j = W_j[\text{recv } \nu \text{ as } x \text{ in } e_k] \quad (335)
\]

\[
e_j' = W_j[(e_k[x \mapsto v]; | \text{recv } \nu \text{ as } x \text{ in } e_k)] \quad (336)
\]

The proof is similar to the case \( e_j = \text{recv } \nu \text{ as } v \text{ in } e_k \) in the proof of BD-PAR-STEP-L.

\( i \) Proof for remaining results: Since \( n = m \), we trivially have the results stated in 3 and 4.

**Case BD-PAR-SEND-RECV-R**: Similar to above case except that we prove when right projection takes a step.

**Case BD-PAR-RECV-SEND-L**: Similar to the proof of BD-PAR-SEND-RECV-L except that we now prove
\[
W_j[\nu | \text{send } \nu \text{ when } e_{sa}] \text{ is well-typed.}
\]

**Case BD-PAR-RECV-SEND-R**: Similar to the proof of BD-PAR-SEND-RECV-R.

This completes the type preservation proof for the distributed system.

Using the Lemma 25, we will prove Theorem 1 restated as below. The key insight is that a well-typed DFLATE configuration cannot evaluate to bracketed final values on the attacker node \( p_j \).

**Theorem 3** (DFLATE noninterference with clearance). Let \( e_k \text{ for } k \in \{1, \ldots, m\} \) be DFLATE programs such that
\[
\Pi_{init}; \Gamma_k; \Theta_k; p_k; pc_k \vdash e_k; \ \tau_k, \text{ where } \tau_j = \ell \text{ says } \text{bool} \text{ and } x : H \text{ says } \hat{\nu} \in \Gamma, \text{ for some } H, \text{ and for all } k
\]
Processes do not downgrade \( H \):
\[
\forall k \in \{1, \ldots, m\}, \forall \ell'. \Pi_{init} \vdash H \sqsubseteq \ell' \iff \Pi_k' \vdash H \sqsubseteq \ell'
\]
The attacker has limited authority:
\[
\Pi_{init} \not\vdash H \not\sqsubseteq \ell
\]

then for all \( v_z, z \in \{1, 2\} \) such that \( \Pi; \Gamma; \Theta; p_i; pc_i \vdash v_z; H \text{ says } \hat{\nu}, \text{ if } \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_i, e_i[x \mapsto v_z] \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \) steps to \( \langle p_1, e_1' \rangle \parallel \cdots \parallel \langle p_i, e_i' \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \) we have \( \hat{v}_j = \hat{v}_j' \).

**Proof.** Given that
\[
\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_i, e_i[x \mapsto v_z] \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \implies^* \langle p_1, e_1' \rangle \parallel \cdots \parallel \langle p_j, \hat{v}_j' \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_n, e_n' \rangle
\]

Also given that
\[
\Pi; \Gamma, x : H^\tau \text{ says } \hat{\nu}; \Theta; p_i; pc_i \vdash e_i; \tau_i
\]

From DT-BRACKET-VALUES, we have that \( \Pi; \Gamma; \Theta; p_i; pc_i \vdash (v_1 | v_2) : H^\tau \text{ says } \hat{\nu} \). Invoking substitution lemma (Lemma 14) we have that \( \Pi; \Gamma; \Theta; p_i; pc_i \vdash e_i[x \mapsto (v_1 | v_2)]; \tau_i \). From the type preservation (Lemma 25) we have that for all \( k \in \{1, \ldots, n\} \)
\[
\Pi_k; \Gamma_k; \Theta_k; p_k; pc_k \vdash e_k; \tau_k \text{ such that } \Theta_k \supseteq \Theta_k. \text{ We prove for each case separately.}
\]

From the completeness property (Lemma 5), we have that
\[
\langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_i, e_i[x \mapsto (v_1 | v_2)] \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \implies^* \langle p_1, e_1' \rangle \parallel \cdots \parallel \langle p_j, \hat{v}_j' \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \parallel \cdots \parallel \langle p_n, e_n' \rangle
\]
and from the soundness property (Lemma 2), we have that \( \nu_j' = \nu_j' \text{ for } z \in \{1, 2\} \).

From the type preservation lemma, we have that \( \Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash e_j'; \ell \text{ says } \text{bool} \). We have to prove that
\[
[\hat{v}_j]_1 = [\hat{v}_j']_2
\]

Let us assume that \( \hat{v}_j' \) is a bracket value. Then by DT-BRACKET-VALUES, we have the following:
\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash [\hat{v}_j']_1; \tau_j \quad (339)
\]
\[
\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash [\hat{v}_j']_2; \tau_j \quad (340)
\]
\[
\Pi_j \vdash H^\tau \leq \tau_j \quad (341)
\]
From (341), we require that \( \Pi_j \vdash H^\tau \subseteq \ell \) (rule DP-LBL). But from condition (338) we are already given that \( \Pi_j \nvdash H^{-} \land H^{-} \subseteq \ell \). Thus \( \Pi_j ; \Gamma_j ; \Theta_j ; p_j ; \mathcal{P}_j \vdash \tilde{\mathcal{v}}^j_j : \tau_j \) cannot be well-typed by DT-BRACKET-VALUES and so \( \tilde{\mathcal{v}}^j_j \) itself isn’t a bracket term.

Note that \( \tilde{\mathcal{v}}^j_j \) can only be a where term. We can prove this by inducting on the structure of the value. The only interesting case is \( \eta_{\ell} \mathcal{v}'' \). In this case, we can easily prove that \( \mathcal{v}'' \) cannot be a bracket term because sumtypes (\( \text{bool} = \text{unit} + \text{unit} \)) cannot protect any principal.

Without loss of generality, let \( \tilde{\mathcal{v}}^j_j = w \) where \( \langle p \gg q \rangle \). We now need to prove that \( w \) is not a bracket term. We prove this by induction on the number of where (either bracket or non-bracket) terms. For base case, where there are zero where terms, we already proved that \( \tilde{\mathcal{v}}^j_j \) is not a bracket term.

\( j) \) Induction Hypothesis: Let \( \tilde{\mathcal{v}}^j_j = w \) where \( n^{-1}\langle p_{n-1} \gg q_{n-1} \rangle \) where \( \langle p_n \gg q_n \rangle \). For \( k = \{0 \ldots n - 1\} \), \( w \) where \( k(p_k \gg q_k) \) is not a bracket term.

\( k) \) Induction Step: For the induction step, we need to prove that \( \mathcal{v}''' \) is \( w \) where \( \langle p \gg q \rangle \) where \( n^{-1}\langle p_{n-1} \gg q_{n-1} \rangle \) where \( \langle p_n \gg q_n \rangle \) is not a bracket term. From the induction hypothesis, we already have that \( w \) where \( \langle p \gg q \rangle \) where \( k(p_k \gg q_k) \) where \( \langle p_n \gg q_n \rangle \) is not a bracket term for \( k = \{1 \ldots n\} \). Thus it remains to prove that \( w \) where \( \langle p \gg q \rangle \) is not a bracket term.

Case \( \pi = \rightarrow \): From condition (338), we have \( \Pi_j \nvdash H^{-} \subseteq \ell^{-} \) and so we have that \( \Pi_j \nvdash \ell^{-} \gg H^{-} \) where \( \Pi_j = \Pi_j, \langle p \gg q \rangle \).

Note that this is same as \( \Pi_j \nvdash H^{-} \subseteq \ell^{-} \). Running an argument similar to the base case, we have that \( w \) cannot be a bracket term (as \( \Pi_j \nvdash \mathcal{P}_j \leq \tau_j \)).

Case \( \pi = \leftarrow \): Similar to above the above case, using condition (338), we have \( \Pi_j \nvdash H^{-} \subseteq \ell^{-} \). Again \( w \) cannot be a bracket term.

This completes the proof that \( \mathcal{v}''' \) isn’t a bracket term and so \( [\mathcal{v}''']^j_1 = [\mathcal{v}''']^j_2 \).

We elaborate the erasure/observation function in the below table.

| \( \mathcal{O}(x, \Pi, A) \) | \( = x \) |
| \( \mathcal{O}(\eta_{\ell} e', \Pi, A) \) | \( = \eta_{\ell} \mathcal{O}(e', \Pi, A) \) |
| \( \mathcal{O}(\pi e', \Pi, A) \) | \( = \mathcal{O}(\pi e', \Pi, A) \) |
| \( \mathcal{O}(\lambda (x: \tau) [pc, \Theta, \mathcal{P}, \Pi_\lambda]. e', \Pi, A) \) | \( = \lambda (x: \tau) [pc, \Theta, \mathcal{P}, \Pi_\lambda]. \mathcal{O}(e', \Pi, A) \) |
| \( \mathcal{O}(e''', \Pi, A) \) | \( = \mathcal{O}(e', \Pi, A) \mathcal{O}(e''', \Pi, A) \) |
| \( \mathcal{O}(\langle e_1, e_2 \rangle, \Pi, A) \) | \( = \langle \mathcal{O}(e_1, \Pi, A), \mathcal{O}(e_2, \Pi, A) \rangle \) |
| \( \mathcal{O}(\text{proj}_j e', \Pi, A) \) | \( = \text{proj}_j \mathcal{O}(e', \Pi, A) \) |
| \( \mathcal{O}(\text{inj}_j e', \Pi, A) \) | \( = \text{inj}_j \mathcal{O}(e', \Pi, A) \) |
| \( \mathcal{O}(\text{case } e' \text{ of } \text{inj}_1(x). e_1 | \text{inj}_2(x). e_2, \Pi, A) \) | \( = \text{case } \mathcal{O}(e', \Pi, A) \mathcal{O}(e_1, \Pi, A) | \mathcal{O}(e_2, \Pi, A) \) |
| \( \mathcal{O}(\text{bind } x = e' \text{ in } e''', \Pi, A) \) | \( = \text{bind } x = \mathcal{O}(e', \Pi, A) \text{ in } \mathcal{O}(e''', \Pi, A) \) |
| \( \mathcal{O}(\text{assume } e' \text{ in } e''', \Pi, A) \) | \( = \text{assume } \mathcal{O}(e', \Pi, A) \text{ in } \mathcal{O}(e''', \Pi, A) \) |
| \( \mathcal{O}(e' \text{ where } v, \Pi, A) \) | \( = \mathcal{O}(e', \Pi, A) \text{ where } v \) |
| \( \mathcal{O}(\text{TEE}^t e', \Pi, A) \) | \( = \mathcal{O}(\text{TEE}^t e', \Pi, A) \) |
| \( \mathcal{O}(\text{runTEE}^t e', \Pi, A) \) | \( = \mathcal{O}(\text{runTEE}^t e', \Pi, A) \) |
| \( \mathcal{O}(\text{spawn } \mathcal{q} (pl.ch[pc; \tau], ch[pc; \tau]), e' \text{ then } e''', \Pi, A) \) | \( = \text{spawn } \mathcal{q} (pl.ch[pc; \tau], ch[pc; \tau]). \mathcal{O}(e', \Pi, A) \text{ then } \mathcal{O}(e''', \Pi, A) \) |
| \( \mathcal{O}(\text{send } ch e' \text{ then } e_{\text{sen}}, \Pi, A) \) | \( = \text{send } ch \mathcal{O}(e', \Pi, A) \text{ then } \mathcal{O}(e_{\text{sen}}, \Pi, A) \) |
| \( \mathcal{O}(\text{recv } ch \text{ as } x \text{ in } e', \Pi, A) \) | \( = \text{recv } ch \text{ as } x \text{ in } \mathcal{O}(e', \Pi, A) \) |

Erasure functions for sealed values and TEEs are defined in Figure 25. We lift the erasure of expressions to a process as follows.

\( \mathcal{O}(q, e, \Pi, A) = \langle q, \mathcal{O}(e, \Pi, A) \rangle \)

Duplicates in traces are removed using the below definition. It simply compares the pairwise elements and eliminates the duplicated value if they are the same.
Definition C.1 (Trace Deduplication).

\[ U(O((q, e^1), (q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p))) = \begin{cases} O((q, e^1), (q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p)) & \text{if } O((q, e^1), (q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p)) = O((q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p)) \\ U(O((q, e^1), (q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p))) & \text{if } O((q, e^1), (q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p)) = O((q, e^2), (q, e^3), (q, e^m), (q, e^n), (q, e^p)) \end{cases} \]

Two traces are equivalent if they are equal after removing the duplicates.

Definition C.2 (Trace Equivalence). Two traces, \( T_{r1} \) and \( T_{r2} \), are equivalent, denoted as \( T_{r1} \approx T_{r2} \), if they are equal after removing the duplicates. That is

\[ T_{r1} \approx T_{r2} \iff U(T_{r1}) = U(T_{r2}) \]

Before we prove the noninterference for the stronger model, we prove that if two non-bracketed executions (left and right) compute a value on a process running on the attacker node \( p_j \), then there exists a bracketed execution such that the projections of the traces of the process in the bracketed execution is equal to that of the left and right executions. We first introduce some scheduling terminology.

Definition C.3 (Schedule). Let a system take \( k \) number of steps. Then schedule, \( S \), is the sequence of set of nodes on which the expressions are reduced in each step. If

\[ \langle p_1, e^1_1 \rangle \parallel \cdots \parallel \langle p_i, e^1_i \rangle \parallel \cdots \parallel \langle p_j, e^1_j \rangle \parallel \cdots \parallel \langle p_m, e^1_m \rangle \]

then, schedule, \( S = (P_i)_{i=1}^k \) where \( P_i \) is the set of nodes reduced at \( i^{th} \) step.

Remark C.1. Note that the cardinality of \( p_i \) is at least 1 and at most 2.

Example C.1. Consider the following evaluation.

\[ \langle p, \text{bind } x = v \text{ in send ch } x \text{ then } e_{ss} \rangle \parallel \langle q, \text{recv ch as y in y} \rangle \]

\[ \Rightarrow \]

\[ \langle p, \text{send ch } v \text{ then } e_{ss} \rangle \parallel \langle q, \text{recv ch as y in y} \rangle \]

\[ \Rightarrow \]

\[ \langle p, () \rangle \parallel \langle q, v \rangle \]

Then, \( S = \{ p \}, \{ p, q \} \).

Definition C.4 (Scheduling order of a node). Let \( S \) be the schedule of a system. Then scheduling order of a node \( p \), written, \( [S]_p \), is the sequence of elements in the schedule containing \( p \). That is,

\[ [S]_p = \begin{cases} \text{hd}(S), [\text{tail}(S)]_p & \text{if } p \in \text{hd}(S) \\ [\text{tail}(S)]_p & \text{if } p \notin \text{hd}(S) \end{cases} \]

Example C.2. In the example C.1, \( [S]_p = \{ p \}, \{ p, q \} \) and \( [S]_q = \{ p, q \} \)

Lemma 26 (Race Freedom). send and recv are race-free.

Proof Sketch: DFLATE channels are unidirectional. It has two endpoints and can either send or recv between these endpoints. Thus the order of communication is deterministic and race-free.

Lemma 27 (Evaluation of a process is deterministic). Let \( S_1 \) and \( S_2 \) be two different schedules of a DFLATE program.

\[ S_1 = \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_i, e_i \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \Rightarrow^* \langle p_1, e^1_k \rangle \parallel \cdots \parallel \langle p_i, e^i_k \rangle \parallel \cdots \parallel \langle p_n, e^n_k \rangle \]

\[ S_2 = \langle p_1, e_1 \rangle \parallel \cdots \parallel \langle p_i, e_i \rangle \parallel \cdots \parallel \langle p_m, e_m \rangle \Rightarrow^* \langle p_1, e^1_k \rangle \parallel \cdots \parallel \langle p_i, e^i_k \rangle \parallel \cdots \parallel \langle p_n, e^n_k \rangle \]

such that \( [S_1]_p = [S_2]_p \), then \( \langle p_i, e^i_k \rangle = \langle p_i, e^i_k \rangle \)

Proof Sketch: Given that DFLATE programs are race-free (Lemma 26), evaluation of a process is deterministic.

Definition C.5 (Commuting Schedules). Two schedules \( S_1 \) and \( S_2 \) are said to be commuting if they preserve the scheduling order of all nodes. That is,

\[ \forall p, [S_1]_p = [S_2]_p \]
Example C.3.

\[ S_1 = \{ p \}, \{ p, q \}, \{ r \}, \{ p \}, \{ p, r \} \]
\[ |S|_p \]
\[ S_2 = \{ p \}, \{ p, q \}, \{ p, r \}, \{ r \} \]
\[ S_3 = \{ p \}, \{ p, q \}, \{ p, r \} \]
\[ \{ p \}, \{ p, q \}, \{ p, r \} \]

From the second column, we see that \( S_1 \) and \( S_2 \) are commuting but \( S_3 \) does not commute with either \( S_1 \) or \( S_2 \).

**Lemma 28** (Trace of a process in commuting schedules). The trace of a process in two commuting schedules is equal (upto stuttering).

**Proof.** Since commuting schedules preserve the scheduling order of a process, trace remains unchanged (upto stuttering).

A schedule for bracketed execution contains a sequence of pairs of processes scheduled at each step.

**Lemma 29** (Bracketed Schedule is a Prefix of First Execution). Let \( S_l \) and \( S_r \) be the left and right executions shown below.

\[ S_l = \{ p_1, e_1 \} \parallel \cdots \parallel \{ p_k, e_k[x \mapsto v_l(x)] \} \parallel \cdots \parallel \{ p_m, e_m \} \Rightarrow* \{ p_1, e_1^l \} \parallel \cdots \parallel \{ p_k, e_k^l \} \parallel \cdots \parallel \{ p_j, v_{j1} \} \parallel \cdots \parallel \{ p_m, e_m^l \} \]

\[ S_r = \{ p_1, e_1 \} \parallel \cdots \parallel \{ p_k, e_k[x \mapsto v_r(x)] \} \parallel \cdots \parallel \{ p_j, e_j \} \parallel \cdots \parallel \{ p_m, e_m \} \Rightarrow* \{ p_1, e_1^r \} \parallel \cdots \parallel \{ p_j, e_j^r \} \parallel \cdots \parallel \{ p_m, e_m^r \} \]

Let \( S(\mid) \) be the corresponding bracketed execution shown below.

\[ S(\mid) = \{ p_1, e_1 \} \parallel \cdots \parallel \{ p_k, e_k[x \mapsto (v_l \mid v_r)] \} \parallel \cdots \parallel \{ p_m, e_m \} \Rightarrow* \{ p_1, e_1^* \} \parallel \cdots \parallel \{ p_j, e_j^* \} \parallel \cdots \parallel \{ p_m, e_m^* \} \]

If \( P \) is the set of processes in \( S(\mid) \), then \( \forall p \in P, |S(\mid)|_p \leq |S_l|_p \) where \( \leq \) is a prefix where \( |S(\mid)|_1 \) is the left projection of the bracketed schedule. Similar statement holds for right projection of the bracketed schedule.

**Proof.** We induct on the length of the \( S(\mid) \) which is same as the number of the evaluation steps of the bracketed execution.

**Base Case:** Given \( |S(\mid)|_1 = P_1 \). Hence

\[ |S(\mid)|_1 = \begin{cases} P_1 & \text{if } p \in P_1 \\ \emptyset & \text{otherwise} \end{cases} \]

Note that both executions start with the same configuration. Since evaluation of a process for the same number of steps is deterministic (Lemma 27), we have \( |S_l|_p = P_1 \) if \( p \in P_1 \) else \( \emptyset \). Thus,

\[ \forall p \in P, |P_1|_p \leq |S_l|_p \]

This proves the base case.

**Inductive Case:** Let \( |S(\mid)| = k + 1 \). Given that

\[ \forall p, |S(\mid)|_p \leq |S_l|_p \]

Let \( |S(\mid)|_{k+1} = P_{k+1} \). Hence

\[ |S(\mid)|_{k+1} = \begin{cases} |S(\mid)|_p |P_{k+1} & \text{if } p \in P_{k+1} \\ |S(\mid)|_{k+1} & \text{otherwise} \end{cases} \]

Let \( |S(\mid)|_{k+1} \) be a prefix where \( |S(\mid)|_{k+1} \) is \( k' \) number of steps for process \( p \). From induction hypothesis, we have that \( |S(\mid)|_p \leq |S_l|_p \). That implies, \( |S(\mid)|_p = |S_l|_p \). That is, the left execution has also taken \( k' \) number of steps on node \( p \). Thus the configuration is same for process \( p \) (Lemma 27).

If \( |P_{k+1}| = 2 \), then \( \exists q \) such that \( q \neq p \) and \( q \in P_{k+1} \). Following similar reasoning, we have that the configuration for process \( q \) is also same in both bracketed and left execution. Thus they take step in the same way and so \( |S(\mid)|_{k+1} \) is \( 1: k+1 \). Thus \( |S(\mid)|_{k+1} \leq |S_l|_p \).

Similar proof follows for right execution. Hence the proof.

We now prove the noninterference for stronger confidentiality attacker stated in Theorem 2 (and restated in an equivalent way below). Note that the statement expands the definition of trace equivalence using the observation function.

**Theorem 4** (DFLATE Noninterference with Stronger Observational Model). Let \( e_k \) for \( k \in \{ 1 \ldots m \} \) be DFLATE programs such that \( \Pi_{init} \vdash \Gamma_k \Theta_k : P_k; p_c k \vdash e_k : \tau_k \) and \( x : H \rightarrow \) says \( \tau \in \Gamma_i \) for some \( H \) and, and

\[ \text{Processes do not downgrade } H \rightarrow : \] (342)

\[ \forall k \in \{ 1 \ldots m \}, \forall \ell. \Pi_{init} \vdash H \subseteq \ell \Leftrightarrow \Pi_{init}^P \vdash H \subseteq \ell \]

\[ \Pi_{init} \not\vdash p_j \supseteq H \rightarrow \] (343)

The attacker has limited authority:
For all $v_z, z \in \{1, 2\}$ such that $\Pi_i; \Gamma; \Theta; p_i; pc_i \vdash v_z : H \Rightarrow \text{say } \tau$, if

$$S_l = \langle p_1, e_1 \rangle \mid \cdots \mid \langle p_i, e_i[x \mapsto v_i] \rangle \mid \cdots \mid \langle p_j, e_j \rangle \mid \cdots \mid \langle p_m, e_m \rangle$$

then traces of the process executing on node $p_j$, as observed by the attacker on node $p_j$, are equivalent in both executions. That is,

$$O((p_j, e_j_1, \hat{\Pi}_{D(j)}), p_j^-), \ldots, O((p_j, e_j_1, \hat{\Pi}_{D(j)}), p_j^-)$$

where $e^s$ represents the $s^{th}$ step of the system.

Proof. We have that the initial configuration of both left and right executions can be projected from bracketed execution. That is,

$$\langle p_1, e_1 \rangle \mid \cdots \mid \langle p_i, e_i[x \mapsto (v_l \mid v_r)] \rangle \mid \cdots \mid \langle p_m, e_m \rangle$$

From the completeness property (Lemma 5), we have that

$$S_l = \langle p_1, e_1 \rangle \mid \cdots \mid \langle p_i, e_i[x \mapsto (v_l \mid v_r)] \rangle \mid \cdots \mid \langle p_m, e_m \rangle \Rightarrow^* \langle p_1, e_1^s \rangle \mid \cdots \mid \langle p_j, e_j^s \rangle \mid \cdots \mid \langle p_m, e_m^s \rangle$$

and from the soundness property (Lemma 2), we have that $[v_j^s]_1 = v_{ji}$ and $[v_j^s]_2 = v_{jr}$. Now, consider any intermediate step: $(p_j, e_j')$. From type preservation (Lemma 25), we have $\Pi_j; \Gamma_j; \Theta_j; p_j; pc_j \vdash e' : \tau$. We have to prove that:

$$O((p_j, e_j'), \hat{\Pi}_{D(j)}, p_j^-) = O((p_j, e_j'), \hat{\Pi}_{D(j)}, p_j^-)$$

It is clear that $e_j'$ itself is not a bracket expression as it would violate the clearance requirement if it were well-typed as per DT-BRACKET. That is, as per DT-BRACKET, we have $\Pi_j \vdash pc' \subseteq H \Rightarrow$ for some $pc'$ such that $\Pi_j \not\vdash p_j^- \Rightarrow pc'$ (due to clearance). However, we are given that $\Pi_j \not\vdash p_j^- \Rightarrow H \Rightarrow$ which is a contradiction. Thus $e_j'$ itself is not a bracket.

Now, assume that:

$$O((p_j, e_j'), \hat{\Pi}_{D(j)}, p_j^-) \neq O((p_j, e_j'), \hat{\Pi}_{D(j)}, p_j^-)$$

Then there should be some subterm $e$ of $e_j'$ such that

$$O((p_j, e_1), \hat{\Pi}_{D(j)}, p_j^-) \neq O((p_j, e_2), \hat{\Pi}_{D(j)}, p_j^-)$$

We have 2 cases: either $e$ itself is some bracketed value ($v_1$ or $v_2$) that is well-typed by DT-BRACKET-VALUES or $e$ is some well-typed TEE$^t (e_1 | e_2)$. 

Case 1: Let $(v_1 | v_2)$ be a well-typed bracket term. That is, $\Pi'''_j; \Gamma'''_j; \Theta'''_j; p_j; pc'''_j \vdash (v_1 | v_2) : \tau$. Then from DT-BRACKET-VALUES, we require that $\Pi' || H \Rightarrow \subseteq \tau$. This implies that for $k \in \{1, 2\}$, $v_k \in \{(p, q), (v', v'), (p \geq q), (\eta, v', v', \omega, \text{runTEE}^t \hat{v}) \}$ as other values cannot be protected. Note that $(v_1 \mid v_2 \mid v_3 \mid v_4)$ is not well-typed if $v_2 \neq v_3$.

Case 2: Let the expression be TEE$^t (e_1 | e_2)$. Then from DT-BRACKET, we require that $\Pi' || H \Rightarrow \subseteq \tau$ and $\Pi'_j; \Theta'_j; t; pc'_i \vdash e_i : \tau$ for $i \in \{1, 2\}$. Then from the clearance (Lemma 7), we have that $\Pi' || t^- \Rightarrow H \Rightarrow$. This implies that $\Pi' || t^- \Rightarrow H \Rightarrow$ and so $O($TEE$^t e_1, \hat{\Pi}_{D(j)}, \hat{\ell}^-) = $TEE$^t \bullet$ and $O($TEE$^t e_2, \hat{\Pi}_{D(j)}, \hat{\ell}^-) = $TEE$^t \bullet$. Hence they are equal after erasure. Remaining terms can be proved using induction that the erasure results in same terms.
Repeatedly applying type preservation will thus give that
\[
O((p_j, [e_j]_1), \hat{\Pi}_{D_j(j)}, p_j^{-\top})\ldots, O((p_j, [v_j^*]_1), \hat{\Pi}_{D_i(j)}, p_j^{-\top})
\approx
O((p_j, [e_j]_2), \hat{\Pi}_{D_j(j)}, p_j^{-\top})\ldots, O((p_j, [v_j^*]_2), \hat{\Pi}_{D_i(j)}, p_j^{-\top})
\]
However, we still need to prove that the trace for \(p_j\) node in bracketed execution equals the trace in corresponding executions. Consider the following two executions: given individual execution with schedule \(S_i\) and bracketed execution projected on the left side with schedule, \(S_i(1)\). Further, we assume that the length of the \(S_i\) is the minimum number of steps required to reduce \([e_j]_1\) on node \(p_j\) to a value. Note that this may incur evaluating other nodes due to communication. However, the schedule for the bracketed execution eliminates evaluation on unnecessary nodes.

We now argue that \(S_i(1)\) commutes with \(S_i\) for the first \(s\) steps. Lemma 29 allows us to reschedule \(S_i\) to \(S_{i'}\) such that \([S_i(1)]1:s = S_{i'}1:s\) (upto stuttering). The schedule can be completed such that it satisfies the property
\[
\forall p, [S_i]_{pi} = [S_{i'}]_{pi}
\]
Note that such a construction exists but we elide the actual details. Since \(S_i\) and \(S_{i'}\) are commuting, their traces for node \(p_j\) are same (Lemma 28).

Also note that trace of node \(p_j\) after \(s\) steps in \(S_{i'}\) is simply a chain of \(\langle p_j, v_j^\top \rangle\) and deduplication removes them completely. So we only consider the trace generated by the schedule \(S_{i'}[1:s]\). However, this is equal to the \(S_i(1)\). This implies that if both executions start in the same configuration, they take same number of steps on each node and thus end in same configuration. Thus traces generated by \(S_{i'}\) (and hence \(S_i\)) and \(S_i(1)\) are equal.

Similarly, we can prove that trace of node \(p_j\) generated by the second execution \(S_i\) and bracketed execution \(S_i(1)\) is same. This implies that we can now use traces of bracketed execution to reason about the trace equivalence. From (345), we already proved that the projected traces of a bracketed execution are equal. Hence proved.

\section*{Appendix D}

\textbf{FLAM and FLAC}

The following lemma establishes a tight connection between FLAM and FLAC.

\textbf{Lemma 30} (FLAC implies FLAM). Let \(\Pi = H(c)\). If \(\Pi; pc; \ell \vdash p \gg q\), then \(H; c; pc; \ell \vdash p \gg q\).

\textbf{Proof}. Proof is by induction on the derivation of the robust assumption \(\Pi; pc; \ell \vdash p \gg q\). Interesting case is R-Assume.

\textbf{Case R-Assume}: From the premises, we have, that
\[
\langle p \gg q | \ell \land \nabla(q) \rangle \in \Pi
\]

\textbf{Case R-Static}: Since \(L \models p \gg q\), we have that
\[
H; c; \bot \land \top ; \ell \vdash p \gg q
\]

\textbf{Case R-ConjR}: Straightforward.

\textbf{Case R-DisjL}: Straightforward.

\textbf{Case R-Trans}: Straightforward.

\textbf{Case R-Weaken}: Straightforward.