A Language-Based Approach to Secure Quorum Replication

Lantian Zheng
Google Inc.
zlt@google.com

Andrew C. Myers
Computer Science Department
Cornell University
andru@cs.cornell.edu

Abstract

Quorum replication is an important technique for building distributed systems because it can simultaneously improve both the integrity and availability of computation and storage. Information flow control is a well-known method for enforcing the confidentiality and integrity of information. This paper demonstrates that these two techniques can be integrated to simultaneously enforce all three major security properties: confidentiality, integrity and availability. It presents a security-typed language with explicit language constructs for supporting secure quorum replication. The dependency analysis performed by the type system of the language provides a way to formally verify the end-to-end security assurance of complex replication schemes. We also contribute a new multilevel timestamp mechanism for synchronizing code and data replicas while controlling previously ignored side channels introduced by such synchronization.

1. Introduction

Distributed systems are ubiquitous and typically contain host machines that may fail benignly (fail-stop) or malignly (Byzantine). A significant challenge for such systems is to enforce system-wide security policies. Information flow control and replication are two distinct—but, we argue, complementary—techniques for building secure distributed systems. Information flow control can enforce end-to-end confidentiality and integrity policies, whereas replication is the standard technique to prevent failed hosts from compromising the integrity and availability of distributed systems. This paper introduces a new way to combine information flow control and replication in distributed systems. The result is a way to achieve strong assurance of end-to-end confidentiality, integrity and availability for distributed systems.

To balance the requirements of availability and integrity in distributed systems, it is necessary to replicate information and computation across the distributed system, and to coordinate this replication via distributed protocols. In particular, *quorum replication* [3, 4, 9] is a frequently used approach. Our new idea is to use the analysis of information flow to reason soundly about the integrity and availability offered by quorum replication. Compared to prior work on quorum replication, this approach offers the advantage that the replication strategy is based on high-level information security policies, rather than on simplistic, uniform assumptions about host failures (e.g., that no more than a fixed number of host failures can occur).

To integrate information flow analysis and quorum replication, we demonstrate that the integrity and availability guarantees of a quorum system can be analyzed elegantly using a lattice-based label model. We develop the first type-based dependency analysis that addresses the interaction of integrity and availability created by distributed protocols that aim to provide both properties. Previous work [2, 18] has used information flow analysis to guide the use of replication in the secure partitioning framework [16], addressing confidentiality and integrity but not availability; in fact, its replication schemes can reduce availability.

Previous work on quorum replication has largely ignored the possibility that information can leak via distributed protocols. We identify a new information channel related to the timestamps that are needed to ensure data consistency in quorum replication schemes. To prevent possible confidentiality violations via this information channel, we propose a novel scheme of multilevel labeled timestamps.

The rest of the paper is organized as follows. Section 2 introduces a security-typed imperative language with quorum constructs. Its operational semantics formalizes quorum replication. Section 3 describes the type system of the language. This type system embodies a dependency analysis of end-to-end security properties. Section 4 states the security theorem: well-typed programs enforce noninterference and are semantically secure. (See the appendix for proofs.) Section 5 covers related work, and Section 6 concludes.

PLAS '14, July 29, 2014, Uppsala, Sweden.

http://dx.doi.org/10.1145/

2. A language for replicated computation

We describe our approach in the context of Qimp, a simple imperative language extended with constructs for replicated storage and computation. Qimp is designed for the common distributed computing paradigm in which a client host machine may use a set of server hosts to store data and perform computation. Server hosts may fail, so it is important to use replication to ensure high integrity and availability. Qimp models this replication explicitly.

2.1 Quorum replication vs. information flow control

Replicating data in quorum systems is a well-known technique for increasing availability and integrity [3, 4, 9]. A *quorum system* is a collection of subsets of a set of hosts where data is replicated. Availability is improved because a read or write operation on replicated data is able to complete even when only a suitable subset of the hosts (a quorum) responds. Quorum replication has three major ingredients:

- A *failure model* that specifies which hosts can fail and in what ways. Typically the failure model is specified in terms of the maximum number of host failures that can be tolerated, though other formalizations exist, such as *survivor sets* [5] and *fail-prone systems* [9].
- *Read and write protocols* for reading and writing data replicated in a quorum system.
- *Quorum intersection constraints* which require that quorums overlap enough to ensure data consistency.

More recently, language-based information flow control has been used to analyze end-to-end security properties of replication [2, 18] from the following angles:

- *Lattice-based security labels* that offer an abstract and expressive way to model failures that affect integrity, availability, or confidentiality. Potential failures in distributed systems can be modeled by assigning labels to hosts.
- *Replicated computation* that executes the same code on different hosts and synthesizes the final result using multiple received responses.
- *Dependency analyses*, often formalized as security type systems, that derive security constraints based on data dependencies caused by information flow within programs.

There are parallels between these two lines of work on building trustworthy distributed systems. Indeed, this paper demonstrates for the first time that language-based information flow control can be used to analyze quorum replication, simultaneously enforcing confidentiality, integrity and availability. We show that quorum reads and writes can be viewed as replicated computation and that the language-based approach can be instantiated to derive a quorum construct similar to *masking quorum systems* [9].

2.2 Replicated computation in Qimp

A Qimp program is implicitly run on a trusted client host machine. For simplicity, we assume the client host has no local storage, so each memory location m must be replicated onto a set of server hosts H. Replicated storage offers the ability to query and update storage locations. For example, the client can query all the server hosts in H for the contents of location m, allowing the correct value to be obtained even if some hosts in H are compromised by an adversary.

Replicated computation is a natural generalization of replicated storage. For example, we can view a query to storage location m as a replicated computation because it is equivalent to invoking (in parallel on each host in H) a remote function that evaluates the dereference expression $!m$, and then determining the value of $!m$ based on the return values from each replicated invocation. Similarly, to update m with value v, the client host can ask the hosts in H to evaluate (in parallel) the assignment expression $m := v$.

The Qimp language provides a generic construct for evaluating an expression e on multiple server hosts H and determining the correct value of the expression based on the values returned by those hosts. In general, it is possible that some hosts in H may experience availability failures and consequently not respond. Therefore, the client host must be able to figure out the correct value of e using only the responses from a subset of H. Such a subset is called a *quorum*. Qimp requires that quorums be explicitly specified when evaluating an expression e using H. The host set H together with the set of all valid quorums $Q_1, \ldots, Q_n$ constitute a quorum system $\mathcal{Q}$. The Qimp construct for replicating computation has the following form:

$$\texttt{remote}\, e : \tau[\mathcal{Q}]$$

where τ is the type of the `remote` expression. Operationally, to evaluate this expression, the client host instructs the hosts in $\mathcal{Q}$ to evaluate e and return the result. The client host waits until it receives responses from every host in *some* quorum of $\mathcal{Q}$. Then the value of $\texttt{remote}\, e : \tau[\mathcal{Q}]$ is determined based on the return values from that quorum. Given a location m replicated in $\mathcal{Q}$, quorum read and write operations for m are implemented in Qimp as follows:

$$\begin{array}{ll} \text{Write:} & \texttt{remote}\, m := v : \tau[\mathcal{Q}] \\ \text{Read:} & \texttt{remote}\, !m : \tau'[\mathcal{Q}] \end{array}$$

Consider a quorum write $\texttt{remote}\ m := v : \tau[\mathcal{Q}]$. To provide availability, expression finishes evaluation after all the hosts in some quorum Q_i complete the update. However, this means that some hosts in H may hold an outdated value of m, and they will return these outdated values when they are asked to evaluate $!m$. In that case, the client needs a way to distinguish an old value from the most recent value of m. The natural solution is to use timestamps: when $m := v$ is evaluated on some host, the current timestamp is stored with v as the value of m. Accordingly, the Qimp language

Host sets	H, Q	$\subseteq$	$\mathcal{H}$
Locations	$\mathcal{Q}$	$::=$	$\langle H; \overline{Q} \rangle$
Base labels	l	$\in$	$\mathcal{L}$
Labels	ℓ	$::=$	$\{l_C, l_I, l_A\}$
Base types	β	$::=$	$\texttt{int} \mid \texttt{unit} \mid \tau^{\mathcal{Q}}\ \texttt{ref}$
Security types	τ	$::=$	$\beta_\ell \mid {\beta_\ell}^{\mathcal{Q}}$
Timestamps	t	$::=$	$\langle \overline{l:n}, n \rangle$
Values	v	$::=$	$x \mid n \mid () \mid m \mid v \cdot t$
Expressions	e	$::=$	$v \mid !e \mid e_1 + e_2$
		$\mid$	$\texttt{remote}\ e : \tau[\mathcal{Q}]$
		$\mid$	$v := e \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
		$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e' \mid \texttt{while}\ e\ \texttt{do}\ e'$

Figure 1. Syntax of the Qimp language

provides *stamped values* $v \cdot t$, where t is the timestamp of v. In general, $!m$ evaluates to a stamped value $v \cdot t$ so that the client host can determine the most recent value of m.

2.3 Syntax

The syntax of Qimp is shown in Figure 1. Except for the `remote` expression and stamped values, Qimp is a simple, standard imperative language. In Qimp, values include variable x, integer n, unit value $()$ and memory location m. Expressions include the dereference expression $!e$, addition $e_1 + e_2$, assignment $v := e$, conditional expression $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, `while` expression $\texttt{while}\ e\ \texttt{do}\ e'$ and `let` expression $\texttt{let}\ x = e\ \texttt{in}\ e'$.

A type τ can be either a labeled base type β_ℓ or a *located type* ${\beta_\ell}^{\mathcal{Q}}$ with a location component $\mathcal{Q}$. Label ℓ specifies the security requirements for any value with type β_ℓ. Values with type ${\beta_\ell}^{\mathcal{Q}}$ are replicated in $\mathcal{Q}$. A stamped value $v \cdot t$ has type ${\beta_\ell}^{\mathcal{Q}}$ if v has type β_ℓ and is replicated in $\mathcal{Q}$.

A quorum system $\mathcal{Q}$ has the form $\langle H; \overline{Q} \rangle$, where $\overline{Q}$ represents a list of quorums (subsets of H). We write $|\mathcal{Q}|$ for H, and $Q_i \in \mathcal{Q}$ for $Q_i \in \{\overline{Q}\}$, and $h \in \mathcal{Q}$ for $h \in H$.

Base types include integer type `int`, the unit type, and reference type $\tau^{\mathcal{Q}}\ \texttt{ref}$. A memory location of type $\tau^{\mathcal{Q}}\ \texttt{ref}$ is replicated in $\mathcal{Q}$.

2.3.1 Security labels

Qimp uses a *unified label model* introduced in previous work [19], in which security levels are represented by base labels from a lattice $\mathcal{L}$, no matter which of confidentiality, integrity and availability is considered.

Let l range over $\mathcal{L}$, where $l_1 \leq l_2$ denotes that l_1 is a label lower than or equal to l_2. Let $\bot$ be the lowest security level in $\mathcal{L}$ and $\top$ the highest.

If a base label l is applied to a security property such as confidentiality, the base label intuitively denotes how hard it is for adversaries to compromise the underlying security property. We model the adversary with a security level $l_{\mathtt{A}}$ that represents the security properties the adversary has the inherent power to compromise. A security property labeled with l will be compromised if $l \leq l_{\mathtt{A}}$. For example, suppose l is a confidentiality label on some data. Then the data has high confidentiality if $l \not\leq l_{\mathtt{A}}$, meaning that the adversary cannot directly read the data. Similarly, if l is the availability (or integrity) label of some data, then $l \not\leq l_{\mathtt{A}}$ means the adversary cannot directly compromise its availability (or integrity). The goal of the security type system, then, is to ensure that the adversary cannot exploit existing computations or construct new computations to *indirectly* compromise any of these three security properties.

A security label ℓ contains three base labels l_C, l_I and l_A, respectively representing the confidentiality, integrity and available levels. Suppose $\ell = \{l_1, l_2, l_3\}$. Then notations $C(\ell)$, $I(\ell)$ and $A(\ell)$ represent l_1, l_2 and l_3, respectively. An ordering relation $\sqsubseteq$ between security labels is used to track information flows and data dependencies, where $\sqsubseteq$ is defined by the following rule:

$$\frac{C(\ell_1) \leq C(\ell_2) \quad I(\ell_2) \leq I(\ell_1) \quad A(\ell_2) \leq A(\ell_1)}{\ell_1 \sqsubseteq \ell_2}$$

For example, suppose e_1 has security label ℓ_1 and e_2 has label ℓ_2. Then $e_1 + e_2$ has a label ℓ such that $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$, because the value of $e_1 + e_2$ depends on the values of e_1 and e_2. Based on the above rule, $C(\ell_1) \sqcup C(\ell_2) \leq C(\ell)$ because information about e_1 and e_2 can be learned from the value of $e_1 + e_2$, while $I(\ell) \leq I(\ell_1) \sqcap I(\ell_2)$ because the integrity of $e_1 + e_2$ is at most that of either e_1 or e_2.

A given host h also has a security label. We use $C(h)$, $I(h)$ and $A(h)$ to denote its confidentiality, integrity and availability levels. If $C(h) \leq l_{\mathtt{A}}$, the adversary can read data on h; if $I(h) \leq l_{\mathtt{A}}$, the adversary can change outputs of h; if $A(h) \leq l_{\mathtt{A}}$, the adversary can make h not respond.

For convenience, we use the following notation throughout the paper.

- $C_\sqcap(H)$, $I_\sqcap(H)$ and $A_\sqcap(H)$ represent $\bigsqcap_{h \in H}(C(h))$, $\bigsqcap_{h \in H}(I(h))$ and $\bigsqcap_{h \in H}(A(h))$, respectively.
- $\tau \sqcup \ell'$ represents $\beta_{\ell \sqcup \ell'}$ if $\tau = \beta_\ell$.
- $C(\tau)$ represents $C(\ell)$ if $\tau = \beta_\ell$.
- $\ell \sqsubseteq \tau$ represents $\ell \sqsubseteq \ell'$ if $\tau = \beta_{\ell'}$.

2.3.2 Multilevel secure timestamps

The use of timestamps generates covert *implicit information flows*. Timestamps are incremented as execution proceeds, and therefore contain information about the path taken by execution. An assignment statement needs to store timestamps on server hosts. In order for this to be secure, those hosts must be trusted to learn whatever information may be inferred from the timestamps. For example, consider a conditional expression $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Suppose the timestamp is incremented for different times in e_1 and e_2. It is then possible for a host to learn which branch is taken and the value of e by examining the timestamp at run time. This implicit information flow needs to be controlled.

The covert channel related to timestamps is not technically a *covert timing channel*, because it is based on observing timestamp values rather than actual execution time. Control of timing channels is largely an orthogonal problem, and partially addressed in previous work [1, 12, 17].

The main challenge with controlling the implicit flows caused by timestamps is similar to the label creep problem: the security label of a timestamp keeps increasing along with execution, and eventually the timestamp may become too restrictive to use. To address the challenge, we introduce *multilevel timestamps* that carry multiple components, each tracking execution history at a particular confidentiality level. The key property of a multilevel timestamp is that it can be incremented at a given confidentiality level l such that its value only depends on the part of execution path with a confidentiality label less than or equal to l.

Abstractly, a multilevel timestamp scheme needs to define a labeled increment operation $\mathit{inc}(t,\, l)$ that increments timestamp t at label l, and an ordering relation between multilevel timestamps, which satisfy the following properties:

$$\begin{array}{ll} \textbf{T-Security} & t_1 \approx_l t_2 \implies \mathit{inc}(t_1,\, l) = \mathit{inc}(t_2,\, l) \\ \textbf{T-Soundness} & t < \mathit{inc}(t,\, l) \end{array}$$

where $t_1 \approx_l t_2$ denotes that t_1 and t_2 are indistinguishable at label l, meaning that all components having a label less than or equal to l are equal in t_1 and t_2. The T-Security property guarantees that the information that can be inferred from $\mathit{inc}(t,\, l)$ has a label less than or equal to l, and thus $\mathit{inc}(t,\, l)$ can be safely sent to host h with $l \leq C(h)$. The T-Soundness property ensures that the timestamp is monotonically increasing.

In Qimp, a multilevel timestamp t has the form $\langle \overline{l : n}, n' \rangle$, where $\overline{l : n}$ is a list of pairs $l_1 : n_1, \ldots, l_k : n_k$ such that $l_1 \leq \ldots \leq l_k$, and $n_1, \ldots, n_k$ are integers. The component $l_i : n_i$ means that the timestamp has been incremented n_i times at label l_i. Sometimes it is useful to just increment t at no particular confidentiality level. The unlabeled component n' is included for that purpose. For simplicity, we write $\langle \overline{l : n} \rangle$ for $\langle \overline{l : n}, 0 \rangle$.

When a multilevel timestamp t is incremented at label l, the component of t associated with l is incremented, and the components of t that are high-confidentiality with respect to l are discarded, because those components are not needed to track time at the l level, and discarding them makes the timestamp less restrictive to use while satisfying T-Security. When comparing two timestamps, high-confidentiality components are less significant than low ones, because they are discarded during incrementation.

Suppose $t = \langle l_1 : n_1, \ldots, l_k : n_k, n' \rangle$. Then incrementing t at level l is carried out by the following formula:

$$\mathit{inc}(t,\, l) = \begin{cases} \langle l_1 : n_1, \ldots, l_i : n_i + 1 \rangle & \text{if } l_i = l \sqcap l_{i+1} \\ \langle l_1 : n_1, \ldots, l_i : n_i, l \sqcap l_{i+1} : 1 \rangle & \text{if } l_i \neq l \sqcap l_{i+1} \\ \langle l_1 \sqcap l : 1 \rangle & \text{if } l_1 \not\leq l \end{cases}$$

where $l_i \leq l$, and $l_{i+1} \not\leq l$ or $k = i$, and let $l_{k+1} = \top$.

The ordering on timestamps is determined by the following rules.

$$\frac{(l_1 \leq l_2 \texttt{ and } l_2 \not\leq l_1) \texttt{ or } (l_1 = l_2 \texttt{ and } n_1 < n_2)}{\langle \overline{l : n}, l_1 : n_1, \ldots \rangle < \langle \overline{l : n}, l_2 : n_2, \ldots \rangle}$$

$$\frac{n_1 < n_2}{\langle \overline{l : n}, n_1 \rangle < \langle \overline{l : n}, n_2 \rangle}$$

In general, two multilevel timestamps may be incomparable. For example, $\langle l : 2 \rangle$ and $\langle l' : 3 \rangle$ are incomparable if $l \not\leq l'$ and $l' \not\leq l$. However, this is not a problem for Qimp because all the timestamps generated during the evaluation of a Qimp program are comparable due to T-Soundness. With these definitions, we can prove the following theorem.

Theorem 2.1. The multilevel timestamp scheme of Qimp satisfies T-Security and T-Soundness.

In Qimp, the timestamp is incremented at label $C(\tau)$ when $\texttt{remote}\ e : \tau[\mathcal{Q}]$ is evaluated. So memory updates in different `remote` expressions can be ordered. Memory updates in the same `remote` expression are ordered by incrementing the unlabeled components of timestamps during evaluation of assignments.

For a full example of multilevel timestamps in action, consider evaluating the following expression at timestamp $\langle l_L : 1 \rangle$.

$$\begin{array}{l} \texttt{let}\ x = (\texttt{if}\ e\ \texttt{then remote}\ e_1 : \texttt{int}_{\{l_H,\, l,\, l'\}}[\mathcal{Q}]\ \texttt{else}\ 1) \\ \quad \texttt{in remote}\ e' : \texttt{int}_{\{l_L,\, l,\, l'\}}[\mathcal{Q}'] \end{array}$$

Suppose e has a high confidentiality label, and l_L and l_H represent low and high labels with $l_L \leq l_H$. If the value of e is positive, then $\texttt{remote}\ e_1 : \texttt{int}_{\{l_H,\, l,\, l'\}}[\mathcal{Q}]$ is evaluated, and the timestamp is incremented at level l_H to become $\langle l_L : 1, l_H : 1 \rangle$. Otherwise, the timestamp remains the same. So after evaluating the conditional expression, the timestamp is either $\langle l_L : 1, l_H : 1 \rangle$ or $\langle l_L : 1 \rangle$. When evaluating expression $\texttt{remote}\ e' : \texttt{int}_{\{l_L,\, l,\, l'\}}[\mathcal{Q}']$, the timestamp is incremented at level l_L and the high level component is discarded. So the timestamp becomes $\langle l_L : 2 \rangle$ regardless of which branch of the conditional expression is taken. In addition, we have $\langle l_L : 1, l_H : 1 \rangle < \langle l_L : 2 \rangle$ and $\langle l_L : 1 \rangle < \langle l_L : 2 \rangle$ as evidence of T-Soundness.

2.4 Operational semantics

Figure 2 shows the small-step operational semantics of Qimp. On host h, a Qimp expression is evaluated with the memory state of h and the current timestamp. Thus, a local evaluation step of Qimp is a transition from configuration $\langle e,\, M,\, t \rangle$ to another configuration $\langle e',\, M',\, t \rangle$, written as $\langle e,\, M,\, t \rangle \longrightarrow \langle e',\, M',\, t \rangle$, or simply $\langle e,\, M \rangle \longrightarrow \langle e',\, M' \rangle$ if t is not used in the evaluation.

An expression e is evaluated globally with respect to a global memory state $\mathcal{M}$, which is a map from hosts to their local memories. A global evaluation configuration needs to

track the current timestamp and the set of *delayed evaluations* resulted from quorum replication. The evaluation of $\texttt{remote}\ e : \tau[\mathcal{Q}]$ may complete while some hosts in $\mathcal{Q}$ are still in the middle of evaluating e, resulting in delayed evaluations.

Thus, a global Qimp evaluation configuration is a tuple containing four components: expression e, memory $\mathcal{M}$, delayed evaluations $\mathcal{D}$ and timestamp t. $\mathcal{D}$ maps a tuple $\langle e, h, t\rangle$ to an expression e' or `nil`. If $\mathcal{D}[\langle e, h, t\rangle] = e'$, then the evaluation of e at time t is delayed on host h and evaluated to e' so far. If $\mathcal{D}[\langle e, h, t\rangle] = \texttt{nil}$, it means the evaluation of e is not delayed on h.

A global evaluation step is a transition from configuration $\langle e,\ \mathcal{M},\ \mathcal{D},\ t\rangle$ to another configuration $\langle e',\ \mathcal{M}',\ \mathcal{D}',\ t'\rangle$, written $\langle e,\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle e',\ \mathcal{M}',\ \mathcal{D}',\ t'\rangle$.

Rules (E1) through (E8) are local evaluation rules, and rules (E9) through (E14) are global evaluation rules. Local evaluation rules are mostly standard except for (E7). In (E7), the memory location m to be updated is replicated on quorum system $\mathcal{Q}$, and the existing value of m is a stamped value $v' \cdot t'$. Suppose $t = \langle \overline{l : n}, n'\rangle$. Then we write $\lfloor t \rfloor$ for $\langle \overline{l : n}\rangle$, and $t+1$ for $\langle \overline{l : n}, n'+1\rangle$. If the timestamp t is less than $\lfloor t' \rfloor$, this is an old update to be ignored. Otherwise, the update is new and shall be performed. The timestamp of the new value is $t'' = \mathit{max}(t, t') + 1$, which increments the unlabeled component of $\mathit{max}(t, t')$.

In rule (E9), a `remote` expression is expanded to the following form:

$$\texttt{remote}\ e@h_1, \ldots, e@h_n : \tau[\mathcal{Q}]$$

which denotes evaluations of e on hosts h_1 through h_n. The expanded form makes it convenient to track each individual evaluation step at a host, as shown in rule (E10). Rule (E9) also increments the timestamp at label $C(\tau)$, which both ensures that a later update always has a larger timestamp and purges high-confidentiality information from the timestamp. For each h_i, the evaluation configuration starts to track $\langle e, h_i, t'\rangle$, mapping it to `nil` initially. The multiple updates of $\mathcal{D}$ are represented by notation $\mathcal{D}[\langle e, h_i, t\rangle \mapsto \texttt{nil} \mid h_i \in Q]$.

Rule (E11) computes the final value of an expanded `remote` expression. Suppose there exists a quorum Q of $\mathcal{Q}$ such that all the hosts in Q already completed the evaluation: that is, for each $h_i \in Q$, e_i is a value. Then the final value of this expression is *resolved* based on the values returned from Q and type τ:

$$v = \mathit{resolve}(\{v_i@h_i \mid h_i \in Q\}, \tau)$$

where $\{v_i@h_i \mid h_i \in Q\}$ is an abbreviation for the set of values $\{v_{j1}@h_{j1}, \ldots, v_{jm}@h_{jm}\}$ returned by $Q = \{h_{j1}, \ldots, h_{jm}\}$. The *resolve* function returns the most up-to-date *qualified value* among $v_{j1}, \ldots, v_{jm}$. A qualified value is a value with sufficient integrity. More formally, a value v is qualified with respect to τ, if it is returned by a set

$$\text{(E1)}\quad \frac{M(m) = v}{\langle !m,\ M\rangle \longrightarrow \langle v,\ M\rangle}$$

$$\text{(E2)}\quad \frac{n = n_1 + n_2}{\langle n_1 + n_2,\ M\rangle \longrightarrow \langle n,\ M\rangle}$$

$$\text{(E3)}\quad \frac{n > 0}{\langle \texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ M\rangle \longrightarrow \langle e_1,\ M\rangle}$$

$$\text{(E4)}\quad \frac{n \leq 0}{\langle \texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ M\rangle \longrightarrow \langle e_2,\ M\rangle}$$

$$\text{(E5)}\quad \langle \texttt{let}\ x = v\ \texttt{in}\ e,\ M\rangle \longrightarrow \langle e[v/x],\ M\rangle$$

$$\text{(E6)}\quad \langle \texttt{while}\ e\ \texttt{do}\ e',\ M\rangle \longrightarrow \langle \texttt{if}\ e\ \texttt{then let}\ x = e'\ \texttt{in while}\ e\ \texttt{do}\ e'\ \texttt{else}\ (),\ M\rangle$$

$$\text{(E7)}\quad \frac{M(m) = v' \cdot t' \quad t'' = \mathit{max}(t, t') + 1 \quad M' = (\mathit{if}\ t < \lfloor t' \rfloor\ \mathit{then}\ M\ \mathit{else}\ M[m \mapsto v \cdot t''])}{\langle m := v,\ M,\ t\rangle \longrightarrow \langle (),\ M',\ t\rangle}$$

$$\text{(E8)}\quad \frac{\langle e,\ M,\ t\rangle \longrightarrow \langle e',\ M',\ t\rangle}{\langle E[e],\ M,\ t\rangle \longrightarrow \langle E[e'],\ M',\ t\rangle}$$

$$\text{(E9)}\quad \frac{|\mathcal{Q}| = \{h_1, \ldots, h_n\} \quad t' = \mathit{inc}(t, C(\tau)) \quad \mathcal{D}' = \mathcal{D}[\langle e, h_i, t'\rangle \mapsto \texttt{nil} \mid h_i \in \mathcal{Q}]}{\langle \texttt{remote}\ e : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle \texttt{remote}\ e@h_1, \ldots, e@h_n : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D}',\ t'\rangle}$$

$$\text{(E10)}\quad \frac{\langle e_i,\ \mathcal{M}(h_i),\ t\rangle \longrightarrow \langle e_i',\ M',\ t\rangle}{\langle \texttt{remote}\ \ldots e_i@h_i \ldots : \tau[\mathcal{Q}],\ \mathcal{M},\ t\rangle \longrightarrow \langle \texttt{remote}\ \ldots e_i'@h_i \ldots : \tau[\mathcal{Q}],\ \mathcal{M}[h_i \mapsto M'],\ t\rangle}$$

$$\text{(E11)}\quad \frac{\exists Q \in \mathcal{Q}\ \forall h_i \in Q\ e_i = v_i \quad \mathcal{D}' = \mathcal{D}[\langle e, h_k, t\rangle \mapsto e_k \mid h_k \notin Q] \quad v = \mathit{resolve}(\{v_i@h_i \mid h_i \in Q\},\ \tau)}{\langle \texttt{remote}\ e_1@h_1, \ldots, e_n@h_n : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle v,\ \mathcal{M},\ \mathcal{D}',\ t\rangle}$$

$$\text{(E12)}\quad \frac{\langle e',\ \mathcal{M}(h),\ t\rangle \longrightarrow \langle e'',\ M',\ t\rangle \quad \mathcal{M}' = \mathcal{M}[h \mapsto M'] \quad \mathcal{D}[\langle e_0, h, t\rangle] = e' \quad \mathcal{D}' = \mathcal{D}[\langle e_0, h, t\rangle \mapsto e'']}{\langle e,\ \mathcal{M},\ \mathcal{D}\rangle \longrightarrow \langle e,\ \mathcal{M}',\ \mathcal{D}'\rangle}$$

$$\text{(E13)}\quad \frac{\langle e,\ M\rangle \longrightarrow \langle e',\ M\rangle}{\langle e,\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle e',\ \mathcal{M},\ \mathcal{D},\ t\rangle}$$

$$\text{(E14)}\quad \frac{\langle e,\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle e',\ \mathcal{M}',\ \mathcal{D}',\ t'\rangle}{\langle E[e],\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \langle E[e'],\ \mathcal{M}',\ \mathcal{D}',\ t'\rangle}$$

$$\begin{array}{rcl} E[\cdot] & ::= & [\cdot] + e \mid v + [\cdot] \mid \texttt{if}\ [\cdot]\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & v := [\cdot] \mid\ !\,[\cdot] \mid \texttt{let}\ x = [\cdot]\ \texttt{in}\ e \end{array}$$

Figure 2. Operational semantics of Qimp

of hosts with a combined (joined) integrity label as high as $I(\tau)$. That is, there exists a subset H' of Q such that all the hosts in H' return v and $I(\tau) \leq I_{\sqcup}(H')$ holds.

If $I(\tau) \not\leq l_{\mathtt{A}}$, then the adversary cannot fabricate a qualified value of type τ, because it cannot compromise a set of hosts with a combined integrity label as high as $I(\tau)$. Therefore, a qualified value has sufficient integrity.

The most up-to-date qualified value is simply the qualified value with the largest timestamp. If the returned values are not stamped values, then any qualified value could be viewed as the most up-to-date one. If no qualified value is found, the most up-to-date value is returned by *resolve* function, and in this case the integrity of the value is known to be compromised.

For hosts that are not in Q, the evaluation may not complete yet. So $\mathcal{D}'$ in the resulting configuration needs to track those delayed evaluations by mapping $\langle e, h_k, t\rangle$ to e_k in $\mathcal{D}$ for all h_k that is not in Q.

Rule (E12) shows a delayed evaluation step. Suppose $\langle e_0, h, t\rangle$ is mapped to e' in $\mathcal{D}$, and $\langle e',\ \mathcal{M}(h),\ t\rangle$ is evaluated to $\langle e'',\ M',\ t\rangle$. Then $\langle e_0, h, t\rangle$ is mapped to e'' after this evaluation step, while the global memory state becomes $\mathcal{M}[h \mapsto M']$.

Rule (E13) shows an evaluation step on the client host, which does not update memory.

A compromised host may evaluate expression e not based on the rules in Figure 2. For simplicity, we assume that a compromised host may conduct only two kinds of attacks. First, it may conduct an integrity attack, returning an arbitrary value as the result of e. Second, it may conduct an availability attack, returning no value. These two attacks are formalized as two additional evaluation rules (A1) and (A2). In rule (A1), suppose $I(h_i) \leq l_{\mathtt{A}}$ holds, then host h_i is a low-integrity host whose integrity may be compromised. Thus, any expression e_i to be evaluated on h_i may result in an arbitrary value v. For simplicity, we assume v is still well-typed (of type $\tau^{\mathcal{Q}}$). In rule (A2), host h_i is a low-availability host since $A(h_i) \leq l_{\mathtt{A}}$. Thus, host h_i may become unavailable, and the evaluation of e_i cannot continue, which is simulated by removing the term $e_i@h_i$.

$$\text{(A1)}\quad \frac{I(h_i) \leq l_{\mathtt{A}}}{\begin{array}{l}\langle \mathtt{remote}\ \ldots e_i@h_i \ldots : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \\ \quad \langle \mathtt{remote}\ \ldots v@h_i \ldots : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D},\ t\rangle\end{array}}$$

$$\text{(A2)}\quad \frac{A(h_i) \leq l_{\mathtt{A}}}{\begin{array}{l}\langle \mathtt{remote}\ \ldots e_i@h_i \ldots : \tau[\mathcal{Q}],\ \mathcal{M},\ \mathcal{D},\ t\rangle \longrightarrow \\ \quad \langle \mathtt{remote}\ \ldots e_{i-1}@h_{i-1}, e_{i+1}@h_{i+1}, \ldots : \tau[\mathcal{Q}], \\ \quad \mathcal{M}, \mathcal{D}, t\rangle\end{array}}$$

2.5 Examples

The simplicity of the Qimp language helps focus on the basic constructs for supporting quorum replication. However, Qimp is expressive enough to illustrate some real-world distributed computations and their associated security issues.

2.5.1 Cloud storage

At its core, cloud storage is similar to a remote memory whose value can be read and updated. The following code simulates storing a value in the cloud and then retrieving it. To make the code more readable, we use $e_1; e_2$ as syntactic sugar for $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ where x is fresh.

$$\begin{array}{l}\mathtt{remote}\ m := 42 : \mathtt{unit}_{\{\bot,\ \top,\ l\}}[\mathcal{Q}]; \\ \mathtt{remote}\ !m : \mathtt{int}_{\ell}[\mathcal{Q}]\end{array}$$

Suppose $\mathcal{Q} = \langle\{h_1, h_2, h_3\}; \{h_1, h_2\}, \{h_2, h_3\}, \{h_1, h_3\}\rangle$, which means that m is replicated on three hosts and that every update or read operation needs at least two hosts to complete. We can imagine that h_1, h_2 and h_3 represent three independent cloud storage providers. Thus, replicating m in $\mathcal{Q}$ can tolerate the availability failure of any single provider, achieving higher availability than just storing the data at one place.

It is common for cloud storage providers to keep data access logs. Thus, accessing cloud storage has side effects, generating implicit flows. Consider the following code.

$$\begin{array}{l}\mathtt{let}\ x = \mathtt{remote}\ !m : \mathtt{int}_{\{l_H,\ l,\ l'\}}[\mathcal{Q}]\ \mathtt{in} \\ \quad \mathtt{if}\ x\ \mathtt{then}\ \mathtt{remote}\ !m_1 : \tau[\mathcal{Q}_1]\ \mathtt{else} \\ \quad\quad \mathtt{remote}\ !m_2 : \tau[\mathcal{Q}_2]\end{array}$$

where l_L represents a low security level, and l_H a high level. The code returns the value of m_1 or m_2 depending on the value of m. Suppose both m_1 and m_2 store low-confidentiality data. So it seems that hosts in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ may be low-confidentiality hosts. However, a host in $\mathcal{Q}_1$ or $\mathcal{Q}_2$ may learn about the high-confidentiality value of m by knowing whether $!m_1$ or $!m_2$ is evaluated. To control this implicit flow, we require that the program counter label *pc* of a `remote` expression running at $\mathcal{Q}$ satisfy the following constraint:

$$C(pc) \leq C_{\sqcap}(|\mathcal{Q}|)$$

That ensures that all the hosts in $\mathcal{Q}$ have a confidentiality level at least as high as $C(pc)$.

2.5.2 Timed data deletion

Timed data deletion is often used to ensure confidentiality of data stored remotely. For example, a popular mobile messaging app allows users to back up their messaging histories on remote servers, but backup data is deleted from remote servers after a week. This practice is illustrated by the following code:

$$\begin{array}{l}\mathtt{remote}\ m := 42 : \mathtt{unit}_{\{\bot,\ \top,\ l_H\}}[\mathcal{Q}]; \\ \mathtt{while}\ \mathtt{let}\ x = \mathtt{remote}\ !m_1 : \mathtt{int}_{\{l_L,\ l_H,\ l_H\}}[\mathcal{Q}_1]\ \mathtt{in} \\ \quad (\mathtt{remote}\ m_1 := x - 1 : \mathtt{unit}_{\{\bot,\ \top,\ l_H\}}[\mathcal{Q}_1];\ x)\ \mathtt{do}\ (); \\ \mathtt{remote}\ m := 0 : \mathtt{unit}_{\{\bot,\ \top,\ l_H\}}[\mathcal{Q}]\end{array}$$

Suppose m stores the backup data, $m := 42$ represents making a new backup that happens to be 42, and $m := 0$ represents deleting the backup. The deletion happens after a counter m_1 counts down to 0. Besides using replication to

ensure integrity and availability of the backup data, another security concern in this case is to ensure that deletion happens. This concern is represented by the high availability label l_H of expression $m := 0$, meaning that the adversary cannot affect whether this expression terminates. Intuitively, we need to ensure both high integrity and availability of the counter m_1. This security requirement is captured by the high integrity and availability labels of $!m_1$.

3. Security typing

In Qimp, security is formalized in terms of noninterference properties that are enforced through type checking. The type system of Qimp ensures that any well-typed program satisfies the noninterference properties.

3.1 Secure quorum systems

Depending on the security labels of its hosts, a quorum system can provide certain security guarantees for the data stored in it, as formalized in the following definition.

Definition 3.1 (Secure replication)**.** It is secure to replicate data of type τ in quorum system $\mathcal{Q}$, written $\mathcal{Q} \vdash \tau$, if $\mathcal{Q} \vdash \tau$ is derived by the following rule:

$$\text{(Q1)}\quad \frac{C(\tau) \leq C_{\sqcap}(|\mathcal{Q}|) \quad A(\tau) \leq \bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q)) \quad \forall Q_1, Q_2 \in \mathcal{Q},\ Q_1 \cap Q_2 \vdash I(\tau)}{\mathcal{Q} \vdash \tau}$$

The three constraints in (Q1) respectively guarantee confidentiality, availability and integrity of data replicated in $\mathcal{Q}$.

The confidentiality constraint $C(\tau) \leq C_{\sqcap}(|\mathcal{Q}|)$ ensures that all the hosts in $\mathcal{Q}$ have a confidentiality label at least as high as $C(\tau)$ so that they are allowed to store data of type τ.

The availability constraint $A(\tau) \leq \bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q))$ is based on that evaluating an expression in $\mathcal{Q}$ results in a value as long as a quorum of $\mathcal{Q}$ complete the evaluation, and the availability of $\mathcal{Q}$ is captured by label $\bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q))$.

The integrity constraint requires that the intersection of any two quorums in $\mathcal{Q}$ contains enough correct hosts so that any quorum is able to determine the most up-to-date value of a memory location replicated in $\mathcal{Q}$. Here the notion of "enough correct hosts" is defined in terms of labels and written $Q_1 \cap Q_2 \vdash I(\tau)$. In general, $H \vdash I$ denotes that a set of hosts H can provide integrity guarantee for data replicated in it up to level I. It is defined by the following rule:

$$\text{(Q2)}\quad \frac{H \neq \emptyset \quad \forall H' \subseteq H,\ I \leq I_{\sqcup}(H') \texttt{ or } I \leq I_{\sqcup}(H - H')}{H \vdash I}$$

Rule (Q2) essentially says that $H \vdash I$ iff either the set of compromised hosts in H or the set of correct hosts in H have a combined integrity as high as I. In other words, either the adversary has the inherent capability to compromise data of integrity label I, or the correct hosts in H have a combined integrity label as high as I so that if they all agree on the value of some data, then that value has integrity I.

3.1.1 Masking quorum systems

The label-based security constraints for quorum replication can be instantiated to derive masking quorum system [9], a quorum construct that tolerates failures specified as a fail-prone system $\mathcal{B}$ (a collection of host sets $\{B_1, \ldots, B_n\}$ such that all the failed hosts are contained in some B_i). A quorum system $\mathcal{Q}$ is a masking quorum system with respect to $\mathcal{B}$ if it satisfies the following two properties:

- M-Consistency: $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) - B_1 \not\subseteq B_2$
- M-Availability: $\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : B \cap Q = \emptyset$

First, we construct a label model consistent with the fail-prone system. Let label l be a collection of host sets $\{H_1, \ldots, H_n\}$, meaning the underlying security property is compromised if and only if all the hosts in some H_i are compromised. Then $l_{\mathtt{A}} = \{B_1, \ldots, B_n\}$. Let $l_H = \{H \mid \forall B_i\ H \not\subseteq B_i\}$. In a fail-prone system $\mathcal{B}$, $l_{\mathtt{A}}$ represents a low security level, and l_H represents a high security level. For each host h, we have $C(h) = I(h) = A(h) = \{\{h\}\}$. Given two labels l_1 and l_2, $l_1 \leq l_2$ if for any H in l_2, there exists H' in l_1 such that $H' \subseteq H$.

With this label model, we can prove that if $\mathcal{Q}$ is secure to store data with high integrity and availability labels, then $\mathcal{Q}$ is a masking quorum system.

Theorem 3.1. If $\mathcal{Q} \vdash \texttt{int}_{\{l, l_H, l_H\}}$, then $\mathcal{Q}$ is a masking quorum system.

Proof. By contradiction. Assume M-consistency does not hold. Then there exist $Q_1, Q_2 \in \mathcal{Q}$ and $B_1, B_2 \in \mathcal{B}$ such that $(Q_1 \cap Q_2) - B_1 \subseteq B_2$. Therefore, there exists a subset H of $Q_1 \cap Q_2$ such that $H \subseteq B_1$ and $Q_1 \cap Q_2 - H \subseteq B_2$, which imply $l_H \not\leq I_{\sqcup}(H)$ and $l_H \not\leq I_{\sqcup}(Q_1 \cap Q_2 - H)$. Contradict $Q_1 \cap Q_2 \vdash l_H$.

Assume M-Availability does not hold. Then there exists $B \in \mathcal{B}$ such that B intersects with every Q in $\mathcal{Q}$. Based on the label model, we have $\bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q)) = \{H \mid H \subseteq |\mathcal{Q}|, \forall Q \in \mathcal{Q} : H \cap Q \neq \emptyset\}$. Thus, $B \in \bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q))$. However, for any H in l_H, $H \not\subseteq B$, which contradicts $l_H \leq \bigsqcup_{Q\in\mathcal{Q}}(A_{\sqcap}(Q))$. □

3.2 Typing rules

Let Γ represent a typing assignment, mapping references and variables to types. A typing judgment of Qimp has the form $\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau$, meaning that expression e evaluated in quorum system $\mathcal{Q}$ has type τ with respect to Γ and the program counter label *pc*. Note that the program counter label captures the sensitivity of control flow. For simplicity, a component in the typing environment of a typing judgment may be omitted if the component is irrelevant. For example, in rule (INT), the type of n has nothing to do with the typing environment, and thus the typing judgment is simplified as $\vdash n : \texttt{int}_{\ell}$. If expression e (such as a `remote` expression) is not evaluated in a quorum system, then the quorum system

(INT) $$\vdash n : \mathtt{int}_\ell$$

(UNIT) $$\vdash () : \mathtt{unit}_\ell$$

(VAR) $$\frac{C(\Gamma(x)) \le C_\sqcap(\mathcal{Q})}{\Gamma\,;\mathcal{Q} \vdash x : \Gamma(x)}$$

(LOC) $$\frac{\Gamma(m) = \tau^{\mathcal{Q}} \quad \mathcal{Q} \vdash \tau}{\Gamma \vdash m : \tau^{\mathcal{Q}}\ \mathtt{ref}_\ell}$$

(SV) $$\frac{\Gamma \vdash v : \tau}{\Gamma\,;\mathcal{Q} \vdash v \cdot t : \tau^{\mathcal{Q}}}$$

(ADD) $$\frac{\Gamma\,;\mathcal{Q} \vdash e_i : \mathtt{int}_{\ell_i} \quad i \in \{1,2\}}{\Gamma\,;\mathcal{Q} \vdash e_1 + e_2 : \mathtt{int}_{\ell_1 \sqcup \ell_2}}$$

(DEREF) $$\frac{\Gamma\,;pc \vdash e : \tau^{\mathcal{Q}}\ \mathtt{ref}_\ell \quad \ell \sqsubseteq \tau}{\Gamma\,;\mathcal{Q}\,;pc \vdash !e : \tau^{\mathcal{Q}}}$$

(ASSIGN) $$\frac{\Gamma\,;\mathcal{Q} \vdash v : \tau^{\mathcal{Q}}\ \mathtt{ref}_\ell \quad \Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau \quad pc \sqcup \ell \sqsubseteq \tau}{\Gamma\,;\mathcal{Q}\,;pc \vdash v := e : \mathtt{unit}_{\{\bot,\,\top,\,A(\tau)\}}}$$

(IF) $$\frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \mathtt{int}_\ell \quad \ell \sqsubseteq \tau \quad \Gamma\,;\mathcal{Q}\,;pc \sqcup \ell \vdash e_i : \tau \quad i \in \{1,2\}}{\Gamma\,;\mathcal{Q}\,;pc \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

(WHILE) $$\frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \mathtt{int}_\ell \quad \Gamma\,;\mathcal{Q}\,;pc \sqcup \ell \vdash e' : \mathtt{unit}_{\ell'} \quad l \le I(pc) \sqcap I(\ell) \sqcap A(\ell') \sqcap A(\ell)}{\Gamma\,;\mathcal{Q}\,;pc \vdash \mathtt{while}\ e\ \mathtt{do}\ e' : \mathtt{unit}_{\{\bot,\,\top,\,l\}}}$$

(LET) $$\frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau \quad \Gamma, x{:}\tau\,;\mathcal{Q}\,;pc \vdash e' : \tau' \quad A(\tau') \le A(\tau)}{\Gamma\,;\mathcal{Q}\,;pc \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau'}$$

(EVAL) $$\frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau^{\mathcal{Q}} \quad C(pc) \le C(\tau) \le C_\sqcap(|\mathcal{Q}|)}{\Gamma\,;pc \vdash \mathtt{remote}\ e : \tau[\mathcal{Q}] : \tau}$$

(EVAL2) $$\frac{\forall i \in \{1,\ldots,n\}.\ \Gamma\,;\mathcal{Q}\,;pc \vdash e_i : \tau^{\mathcal{Q}} \quad C(pc) \le C(\tau) \le C_\sqcap(|\mathcal{Q}|)}{\Gamma\,;pc \vdash \mathtt{remote}\ e_1@h_1, \ldots, e_n@h_n : \tau[\mathcal{Q}] : \tau}$$

(SUB) $$\frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau \quad \tau \le \tau'}{\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau'}$$

Figure 3. Typing rules of Qimp

component of the typing environment is represented by $\emptyset$. The typing rules are shown in Figure 3.

Rules (INT), (UNIT), (ADD) are standard. Rule (VAR) adds a confidentiality check to ensure that all the hosts in $\mathcal{Q}$ have a confidentiality label as high as that of x.

Rule (LOC) checks reference values. Reference m has type $\tau^{\mathcal{Q}}\ \mathtt{ref}_\ell$ if $\Gamma(m) = \tau^{\mathcal{Q}}$. In addition, $\mathcal{Q} \vdash \tau$ ensures that $\mathcal{Q}$ is secure enough to store data of type τ.

Rule (SV) checks stamped values. If v has type τ, then $v \cdot t$ has type $\tau^{\mathcal{Q}}$.

Rule (DEREF) is used to check the dereference expression $!e$. Suppose e has type $\tau^{\mathcal{Q}}\ \mathtt{ref}_\ell$. Then $!e$ has type $\tau^{\mathcal{Q}}$. The constraint $\ell \sqsubseteq \tau$ is required because the value of $!e$ depends on the value of e.

Rule (ASSIGN) checks the assignment expression. We require $pc \sqcup \ell \sqsubseteq \tau$ so that information about the program counter and about the reference itself cannot be leaked through side effects of this expression. The availability of $v := e$ depends on the availability of e. Thus, the type of this expression is $\mathtt{unit}_{\{\bot,\,\top,\,A(\tau)\}}$.

In rule (IF), the branches e_1 and e_2 are checked with program counter label $pc \sqcup \ell$ because which branch to take depends on the value of e.

Rule (WHILE) is used to check the `while` expression $\mathtt{while}\ e\ \mathtt{do}\ e'$. The `while` expression always has a unit type. The availability of the expression depends on the availability of both e and e', and the integrity of e because the value of e determines whether the loop ends. In addition, the loop may be infinite, so whether the evaluation terminates depends on the integrity of the program counter. Therefore, the constraint $l \le I(pc) \sqcap I(\ell) \sqcap A(\ell') \sqcap A(\ell)$ is required.

Rule (LET) is used to check expression $\mathtt{let}\ x = e\ \mathtt{in}\ e'$. At run time, e' is evaluated with x being replaced by the value of e. Thus, e' is checked with x bounded to the type of e. The availability of the `let` expression depends on the availability of e, and thus $A(\tau')$ is less than or equal to $A(\tau)$.

Rule (EVAL) checks expression $\mathtt{remote}\ e : \tau[\mathcal{Q}]$. In this rule, e has type $\tau^{\mathcal{Q}}$. In practice, a value returned from a host is not necessarily a stamped value. For example, if e is an assignment expression, then the return value would be $()$. The returned values from remote evaluations are always consumed by the *resolve* function, which works the same way if non-stamped values are treated as stamped values with the smallest timestamp $\langle\rangle$. This treatment simplifies rule (EVAL) and is formalized as a subtyping rule below. The constraint $C(\tau) \le C_\sqcap(|\mathcal{Q}|)$ ensures that hosts in $\mathcal{Q}$ are allowed to receive timestamp $inc(t, C(\tau))$. The constraint $C(pc) \le C(\tau)$ ensures incrementing the timestamp at a level at least as high as $C(pc)$ so that information about the program counter is properly protected by the multilevel timestamp.

Rule (SUB) is standard. The subtyping rules of Qimp are shown as follows:

$$\text{(S1)}\quad \frac{\ell \sqsubseteq \ell'}{\beta_\ell \le \beta_{\ell'}} \qquad \text{(S2)}\quad \tau \le \tau^{\mathcal{Q}}$$

Rule (S1) is standard for a security type system. Rule (S2) is mainly for simplifying rule (EVAL).

This type system satisfies subject reduction.

Definition 3.2 ($\Gamma \vdash \mathcal{M}$)**.** $\mathcal{M}$ is well-typed with respect to Γ, written $\Gamma \vdash \mathcal{M}$, if for any $m \in dom(\Gamma)$, $\Gamma(m) = \tau^{\mathcal{Q}}$ implies that for any $h \in \mathcal{Q}$, $\mathcal{M}[h][m]$ has type $\tau^{\mathcal{Q}}$.

Definition 3.3 ($\Gamma \vdash \mathcal{D}$)**.** $\mathcal{D}$ is well-typed with respect to Γ, written $\Gamma \vdash \mathcal{D}$, if for any e' such that $\mathcal{D}[\langle e, h, t\rangle] = e'$, $\Gamma \vdash e' : \tau$.

Theorem 3.2 (Subject reduction)**.** Suppose $\Gamma ; pc \vdash e : \tau$, and $\Gamma \vdash \mathcal{M}$, and $\Gamma \vdash \mathcal{D}$, and $\langle e, \mathcal{M}, \mathcal{D}, t\rangle \longrightarrow \langle e', \mathcal{M}', \mathcal{D}', t'\rangle$. Then $\Gamma ; pc \vdash e' : \tau$, and $\Gamma \vdash \mathcal{M}'$ and $\Gamma \vdash \mathcal{D}'$.

Proof. By induction on the derivation of $\Gamma ; pc \vdash e : \tau$. □

4. Noninterference

This section formalizes the noninterference results of Qimp, which state that a well-typed Qimp program satisfies the noninterference properties with respect to confidentiality, integrity and availability.

Intuitively, confidentiality noninterference means that running a program with two inputs that are indistinguishable at the low confidentiality level will generate outputs indistinguishable at the low confidentiality level. The following definitions formalize the indistinguishability relations of memories and delayed evaluation configurations with respect to low confidentiality. The confidentiality noninterference of Qimp is formalized in Theorem 4.1.

Definition 4.1 ($\Gamma \vdash \mathcal{M}_1 \approx_{C \le l_{\mathtt{A}}} \mathcal{M}_2$)**.** For all m, if $\Gamma(m) = \tau^{\mathcal{Q}}$ and $C(\tau) \le l_{\mathtt{A}}$, then for any two quorums Q_1 and Q_2 of $\mathcal{Q}$, $v_i = \mathit{resolve}(\{\mathcal{M}_i[h][m] \mid h \in Q_i\}, \Gamma(m))$ for $i \in \{1, 2\}$, and $v_1 = v_2$.

Intuitively, $\Gamma \vdash \mathcal{M}_1 \approx_{C \le l_{\mathtt{A}}} \mathcal{M}_2$ means that for any low-confidentiality reference m, the values of m are the same in $\mathcal{M}_1$ and $\mathcal{M}_2$.

Definition 4.2 ($\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$)**.** Two delayed evaluation configurations $\mathcal{D}_1$ and $\mathcal{D}_2$ are equivalent, written $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$, if for $\{i, j\} = \{1, 2\}$, $\langle e, h, t\rangle \in \mathit{dom}(\mathcal{D}_i)$ and $C(h) \le l_{\mathtt{A}}$ imply $\langle e, h, t\rangle \in \mathit{dom}(\mathcal{D}_j)$.

Theorem 4.1 (Confidentiality noninterference)**.** Suppose $\Gamma ; pc \vdash e : \mathtt{int}_\ell$, and $C(\ell) \le l_{\mathtt{A}}$, $\mathcal{M}_1 \approx_{C \le l_{\mathtt{A}}} \mathcal{M}_2$, and for $i \in \{1, 2\}$, $\langle e, \mathcal{M}_i, \emptyset, t_0\rangle \longrightarrow^* \langle v_i, \mathcal{M}_i', \mathcal{D}_i, t_i\rangle$ without (A1) or (A2) steps. Then $v_1 = v_2$, $t_1 \approx_{l_{\mathtt{A}}} t_2$ and $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$.

Proof. See Appendix A. □

The above theorem assumes that the evaluations of e do not include (A1) and (A2) steps. Note that rules (A1) and (A2) have no confidentiality constraints, and active attack steps based on them may affect low-confidentiality data and effectively be treated as distinguishable low-confidentiality inputs. Thus, assuming the lack of such steps is a simple way to ensure low-confidentiality inputs are indistinguishable, which is the prerequisite of confidentiality noninterference. The theorem still holds if there are (A1) and (A2) steps, but those steps do not produce low-confidentiality effects.

The integrity noninterference of Qimp is formalized in Theorem 4.2.

Definition 4.3 ($\Gamma \vdash \mathcal{M}_1 \approx_{I \not\le l_{\mathtt{A}}} \mathcal{M}_2$)**.** For all m, if $\Gamma(m) = \tau^{\mathcal{Q}}$ and $I(\tau) \not\le l_{\mathtt{A}}$, then for any two quorums Q_1 and Q_2 of $\mathcal{Q}$, $v_i = \mathit{resolve}(\{\mathcal{M}_i[h][m] \mid h \in Q_i\}, \Gamma(m))$ for $i \in \{1, 2\}$, and $v_1 = v_2$.

Theorem 4.2 (Integrity noninterference)**.** Suppose $\Gamma ; pc \vdash e : \mathtt{int}_\ell$, and $I(\ell) \not\le l_{\mathtt{A}}$, and $\mathcal{M}_1 \approx_{I \not\le l_{\mathtt{A}}} \mathcal{M}_2$, and $\langle e, \mathcal{M}_i, \emptyset, t_0\rangle \longrightarrow^* \langle v_i, \mathcal{M}_i', \mathcal{D}_i, t_i\rangle$ for $i \in \{1, 2\}$. Then $v_1 = v_2$.

Proof. See Appendix A. □

In the context of distributed protocols such as quorum read/write, availability is often formulated as a liveness property: all requests eventually end under all possible failure scenarios that the protocols are designed for. In contrast, the end-to-end availability guarantee of Qimp cannot be formulated as a liveness property, because that would entail solving the halting problem. Instead we follow the same approach as in the previous work [19] and define the end-to-end availability guarantee as a noninterference property: the adversary cannot affect whether high-availability programs terminate.

Theorem 4.3 (Availability noninterference)**.** Suppose $\Gamma ; pc \vdash e : \mathtt{int}_\ell$, and $A(\ell) \not\le l_{\mathtt{A}}$, and $\mathcal{M}_1 \approx_{I \not\le l_{\mathtt{A}}} \mathcal{M}_2$, and $\langle e, \mathcal{M}_1, \emptyset, t_0\rangle \longrightarrow^* \langle v_1, \mathcal{M}_1', \mathcal{D}_1, t_1\rangle$ without (A1) or (A2) steps. Then the evaluation of $\langle e, \mathcal{M}_2, \emptyset, t_0\rangle$ always terminates, that is, $\langle e, \mathcal{M}_2, \emptyset, t_0\rangle \longrightarrow^* \langle e'', \mathcal{M}_2'', \mathcal{D}'', t_2''\rangle$ implies $\langle e'', \mathcal{M}_2'', \mathcal{D}'', t_2''\rangle \longrightarrow^* \langle v_2, \mathcal{M}_2', \mathcal{D}_2, t_2\rangle$ for some $\langle v_2, \mathcal{M}_2', \mathcal{D}_2, t_2\rangle$.

Proof. See Appendix A. □

5. Related Work

Language-based information flow control techniques [13] can enforce noninterference, including in concurrent and distributed systems [12, 14]. But this work does not address availability and assumes a trusted computing platform. The Jif/split system [16, 18] dealt with untrusted hosts and introduced secure program partitioning and automatic replication of code and data. The Swift system [2] also uses automatic replication to improve integrity. However, these systems cannot specify or enforce availability, and there is no correctness proof for their (comparatively simple) replication mechanisms. The Fabric system [8] enforces confidentiality and integrity without relying on a trusted platform, but does not support replication or address availability.

In previous work [19], we extend the decentralized label model [11] to specify availability policies and present a type-based approach for enforcing availability policies in a sequential program. This paper examines the distributed setting to permit formal analysis of the availability guarantees of quorum replication schemes.

Walker et al. [15] designed λ_{zap}, a lambda calculus that exhibits intermittent data faults, and use it to formalize the

idea of achieving fault tolerance through replication and majority voting. However, λ_{zap} describes a single machine with at most one integrity fault.

Quorum systems [3, 4, 9**?** , 10] are a well studied technique for improving fault tolerance in distributed systems. Quorum systems achieve high data availability by providing multiple quorums capable of carrying out read and write operations. If some hosts in one quorum fail to respond, another quorum may still be available. The integrity guarantee of quorum systems is usually formalized as regular semantics [6] under simple, symmetric assumptions about the number of hosts that can fail. Our work offers new capabilities. First, it allows the construction of quorum systems based on non-uniform security labels assigned to hosts. Security guarantees are formalized as noninterference properties. Second, hosts in the quorum system can provide more general computation rather than just storage. Third, we control the covert channels created by the quorum protocols themselves.

The Replica Management System (RMS) [7] computes a placement and replication level for an object based on programmer-specified availability and performance parameters. However, RMS does not consider attacks on integrity (Byzantine failures) or on confidentiality.

6. Conclusions

This paper is the first attempt to study quorum replication using a lattice-based label model and a security-typed language Qimp. It provides the first noninterference result for the commonly used technique of quorum replication: end-to-end security assurances of quorum constructs and protocols can be formalized as noninterference properties and provably enforced by the type system of Qimp. The language-based approach also enriches the understanding of quorum replication from the perspective of high-level information flow policies, unifying analysis of all three aspects of security (confidentiality, integrity and availability). The new mechanism of multilevel timestamps is essential to controlling the information channels created by keeping replicas synchronized. These results suggest that other distributed protocols may be analyzed along similar lines, supporting the secure construction of a wider range of distributed systems.

Acknowledgments

This work and its presentation here has benefited from many insightful suggestions, including from Lorenzo Alvisi, Michael Clarkson, Stephen Chong, Heiko Mantel, Danfeng Zhang, Chinawat Isradisaikul, and anonymous reviewers. This work was supported by NSF grant CCF-09644909, ONR grant N00014-13-1-0089, and MURI grant FA9550-12-1-0400, administered by the U.S. Air Force. The views and conclusions here are those of the authors and do not necessarily reflect those of any of these funding agencies.

References

[1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 40–53, January 2000.

[2] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, October 2007.

[3] D. K. Gifford. Weighted voting for replicated data. In *Proc. Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, December 1979. ACM SIGOPS.

[4] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[5] Flavio Junqueira and Keith Marzullo. Designing algorithms for dependent process failures. In *Proceedings of the Workshop on Future Directions in Distributed Computing*, pages 24–28, 2003.

[6] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[7] Mark C. Little and Daniel McCue. The Replica Management System: a scheme for flexible and dynamic replication. In *Proc. 2nd International Workshop on Configurable Distributed Systems*, pages 46–57, Pittsburgh, March 1994.

[8] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.

[9] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.

[10] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Small byzantine quorum systems. In *International Conference on Dependable Systems and Networks (DSN02)*, June 2002.

[11] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[12] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proc. 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[14] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 355–364, January 1998.

[15] David Walker, Lester Mackey, Jay Ligatti, George Reis, and David August. Static typing for a faulty lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, September 2006.

[16] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[17] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2012.

[18] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

[19] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 272–286, June 2005.

A. Noninterference proof

The noninterference result for Qimp is proved by extending the language to a new language Qimp*, which uses a bracket construct to syntactically capture the differences between executions of the same program on different inputs.

Intuitively, each evaluation configuration in Qimp* encodes two Qimp configurations. The operational semantics of Qimp* is consistent with that of Qimp in the sense that evaluation of a Qimp* configuration is equivalent to the evaluation of two Qimp configurations it encodes. The type system of Qimp* can be instantiated to ensure that a well-typed Qimp* configuration satisfies certain invariants. In particular, if the invariant represents some equivalence relation corresponding to noninterference, subject reduction of Qimp* then implies a noninterference result for Qimp. For example, if the invariant is that the low-confidentiality parts of two Qimp configurations are equivalent, the subject reduction of Qimp* implies the confidentiality noninterference of Qimp.

A.1 Syntax extensions

The syntax extension of Qimp* includes bracket constructs, which compose two Qimp terms and capture the differences between two Qimp configuration. In particular, a timestamp may also contain bracket constructs.

$$\begin{array}{rrcl} \text{Values} & e & ::= & \ldots (v,\, v) \\ \text{Expressions} & e & ::= & \ldots (e,\, e) \\ \text{Timeticks} & \eta & ::= & n \mid (n,\, n) \\ \text{Timestamps} & t & ::= & \langle \overline{l : \eta}, \eta \rangle \end{array}$$

Bracket constructs cannot be nested; their subterms must be Qimp terms. Given a Qimp* expression e, let $\lfloor e \rfloor_i$ represent the Qimp expressions that e encodes. The projection functions satisfy $\lfloor (e_1,\, e_2) \rfloor_i = e_i$ and are homomorphisms on other expression forms. A Qimp* local memory M maps references to Qimp* values that encode two Qimp values. Thus, the projection function can be defined on memories too. For $i \in \{1, 2\}$, $\mathit{dom}(\lfloor M \rfloor_i) = \mathit{dom}(M)$, and for any $m \in \mathit{dom}(M)$, $\lfloor M \rfloor_i(m) = \lfloor M_i(m) \rfloor_i$. A Qimp* global memory $\mathcal{M}$ is a pair of Qimp memories $(\mathcal{M}_1,\, \mathcal{M}_2)$. Similarly, a delayed evaluation configuration $\mathcal{D}$ in Qimp* is a pair of Qimp configurations $(\mathcal{D}_1,\, \mathcal{D}_2)$.

$$\text{(E1)}\quad \frac{M(m) = v}{\langle !m,\, M \rangle_i \longrightarrow \langle \lfloor v \rfloor_i,\, M \rangle_i}$$

$$\text{(E7)}\quad \frac{\lfloor M \rfloor_i(m) = v' \cdot t' \quad t'' = \mathit{max}(t, t') + 1 \quad M' = (\mathit{if}\ t < \lfloor t' \rfloor\ \mathit{then}\ M\ \mathit{else}\ M[m \mapsto_i v \cdot t''])}{\langle m := v,\, M,\, t \rangle_i \longrightarrow \langle (),\, M',\, t \rangle_i}$$

$$\text{(E15)}\quad \langle \mathtt{if}\ (v_1,\, v_2)\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\, M \rangle \longrightarrow \langle (\mathtt{if}\ v_1\ \mathtt{then}\ \lfloor e_1 \rfloor_1\ \mathtt{else}\ \lfloor e_2 \rfloor_1 \mid \mathtt{if}\ v_2\ \mathtt{then}\ \lfloor e_1 \rfloor_2\ \mathtt{else}\ \lfloor e_2 \rfloor_2),\, M \rangle$$

$$\text{(E16)}\quad \frac{|\mathcal{Q}| = \{h_1, \ldots, h_n\} \quad t' = \mathit{inc}(t,\, C(\tau)) \quad \mathcal{D}'_i = \mathcal{D}_i[\langle \lfloor e \rfloor_i, h_j, \lfloor t' \rfloor_i \rangle \mapsto \mathtt{nil} \mid 1 \le j \le n]\ \ i \in \{1, 2\}}{\langle \mathtt{remote}\ e : \tau[\mathcal{Q}],\, \mathcal{M},\, (\mathcal{D}_1,\, \mathcal{D}_2),\, t \rangle \longrightarrow \langle \mathtt{remote}\ \lfloor e \rfloor_1 @ h_1^1, \ldots, \lfloor e \rfloor_1 @ h_n^1, \lfloor e \rfloor_2 @ h_2^2, \ldots, \lfloor e \rfloor_2 @ h_n^2 : \tau[\mathcal{Q}],\, \mathcal{M},\, (\mathcal{D}'_1,\, \mathcal{D}'_2),\, t' \rangle}$$

$$\text{(E17)}\quad \frac{i \in \{1, 2\} \quad \exists Q_i \in \mathcal{Q}\ \forall h_j^i \in Q_i\ e_j^i = v_j^i \quad \lfloor \mathcal{D}' \rfloor_i = \lfloor \mathcal{D} \rfloor_i[\langle e, h_k, \lfloor t \rfloor_i \rangle \mapsto e_k^i \mid h_k \notin Q_i] \quad v_i = \mathit{resolve}(\{v_j^i @ h_j^i \mid h_j^i \in Q_i\},\, \tau) \quad v = (\mathit{if}\ v_1 = v_2\ \mathit{then}\ v_1\ \mathit{else}\ (v_1,\, v_2))}{\langle \mathtt{remote}\ e_1^1 @ h_1^1, \ldots, e_n^2 @ h_n^2 : \tau[\mathcal{Q}],\, \mathcal{M},\, \mathcal{D},\, t \rangle \longrightarrow \langle v,\, \mathcal{M},\, \mathcal{D}',\, t \rangle}$$

Figure 4. The operational semantics of Qimp*

Since a Qimp* term effectively encodes two Qimp terms, the evaluation of a Qimp* term can be projected into two Qimp evaluations. An evaluation step of a bracket expression $(e_1,\, e_2)$ is an evaluation step of either e_1 or e_2, and e_i can only access the corresponding projection of the memory. Thus, the configuration of Qimp* has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a bracket term, and if so, which branch of a bracket the term belongs to. For example, the configuration $\langle e,\, M,\, t \rangle_1$ means that e belongs to the first branch of a bracket, and e can only access the first projection of M. We write "$\langle e,\, M,\, t \rangle$" for "$\langle e,\, M,\, t \rangle_\bullet$", which means e does not belong to any bracket.

The operational semantics of Qimp* is shown in Figure 4. It is based on the semantics of Qimp and contains new evaluation rules (E15)-(E17) for manipulating bracket constructs. Rules (E1) and (E7) are modified to access the memory projection corresponding to index i. The rest of the rules in Figure 2 are adapted to Qimp* by indexing each configuration with i. In rules (E9)-(E11), the index i is in $\{1, 2\}$, as rules (E16) and (E17) cover the $\bullet$ case. The following adequacy and soundness lemmas state that the operational semantics of Qimp* faithfully encodes the evaluation of two Qimp terms:

Lemma A.1 (Soundness)**.** Suppose $\langle e,\, \mathcal{M},\, \mathcal{D},\, t \rangle \longrightarrow \langle e',\, \mathcal{M}',\, \mathcal{D}',\, t' \rangle$ in Qimp*. Then there exists evaluation $\langle \lfloor e \rfloor_i,\, \lfloor \mathcal{M} \rfloor_i,\, \lfloor \mathcal{D} \rfloor_i,\, \lfloor t \rfloor_i \rangle \longrightarrow^* \langle \lfloor e' \rfloor_i,\, \lfloor \mathcal{M}' \rfloor_i,\, \lfloor \mathcal{D}' \rfloor_i,\, \lfloor t' \rfloor_i \rangle$ for $i \in \{1, 2\}$ in Qimp.

Proof. By inspection of the evaluation rules. □

Lemma A.2 (Adequacy)**.** If there exists Qimp evaluation $\langle e_i, \mathcal{M}_i, \mathcal{D}_i, t_i\rangle \longrightarrow^* \langle v_i, \mathcal{M}'_i, \mathcal{D}'_i, t'_i\rangle$ for $i \in \{1, 2\}$, and there exists e and t in Qimp* such that $\lfloor e\rfloor_i = e_i$ and $\lfloor t\rfloor_i = t_i$. Then $\langle e, (\mathcal{M}_1, \mathcal{M}_2), (\mathcal{D}_1, \mathcal{D}_2), t\rangle \longrightarrow^* \langle v, \mathcal{M}', \mathcal{D}', t\rangle$ such that $\lfloor v\rfloor_i = v_i$ and $\lfloor t'\rfloor_i = t'_i$.

Proof. By induction on the structure of e. □

A.2 Typing rules

The type system of Qimp* includes all the typing rules in Figure 3 and has an additional rule for typing the bracket expression. Essentially, a bracket expression represents the distinguishable parts of two evaluations. Therefore, the security label of this expression and the program counter label must not satisfy the indistinguishability constraint, which we call the ζ-invariant. For confidentiality, the ζ-invariant is low-confidentiality, implying low-confidentiality parts are indistinguishable. For integrity, the ζ-invariant is high-integrity. An ζ-invariant must satisfy the condition that $\zeta(\ell')$ and $\ell \sqsubseteq \ell'$ imply $\zeta(\ell)$.

$$\text{(BRACKET)}\quad \frac{\Gamma\,;\mathcal{Q}\,;pc \vdash e_i : \tau \quad i \in \{1,2\} \qquad \neg\zeta(\tau) \quad \neg\zeta(pc)}{\Gamma\,;\mathcal{Q}\,;pc \vdash (e_1, e_2) : \tau}$$

A.3 Subject reduction

Definition A.1 ($\Gamma \vdash M$)**.** M is well-typed with respect to Γ, written $\Gamma \vdash M$, if $dom(\Gamma) = dom(M)$ and $\forall m \in dom(\Gamma).\ \Gamma\,;\mathcal{Q} \vdash M(m) : \Gamma(m)$.

Lemma A.3 (Local subject reduction)**.** Suppose $\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau$, and $\Gamma \vdash M$, and $\langle e, M, t\rangle_i \longrightarrow \langle e', M', t\rangle_i$, and $i \in \{1, 2\}$ implies $\neg\zeta(pc)$. Then $\Gamma\,;\mathcal{Q}\,;pc \vdash e' : \tau$ and $\Gamma \vdash M'$.

Proof. By induction on the derivation of $\langle e, M, t\rangle_i \longrightarrow \langle e', M', t\rangle_i$. Most cases are straightforward.

- Case (E15). Since e' is a bracket expression, we just need to prove $\neg\zeta(\tau)$. Because $\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau$, we have $\neg\zeta(\tau)$ by rules (BRACKET) and (IF).

□

Suppose $\mathcal{M}$ is a Qimp memory. Let $\mathcal{M}(m)$ denote the resolved value of m based on all the local values of m in $\mathcal{M}$.

Definition A.2 ($\Gamma \vdash \langle \mathcal{M}, \mathcal{D}, t\rangle$)**.** Suppose $\mathcal{M}$ is $(\mathcal{M}_1, \mathcal{M}_2)$ and $\mathcal{D}$ is (D_1, D_2). $\langle \mathcal{M}, \mathcal{D}, t\rangle$ is well-typed with respect to Γ, written as $\Gamma \vdash \langle \mathcal{M}, \mathcal{D}, t\rangle$, if the following conditions hold.

- For any m such that $\Gamma(m) = \tau^{\mathcal{Q}}$, $\Gamma \vdash \mathcal{M}_i[h][m] : \Gamma(m)$ holds, and $\zeta(\tau)$ implies $\mathcal{M}_1(m) = \mathcal{M}_2(m)$.
- If $\langle e, \mathcal{M}, \mathcal{D}, t\rangle_i \longrightarrow \langle e, \mathcal{M}', \mathcal{D}', t\rangle_i$, then for any $m \in dom(\Gamma)$, $\mathcal{M}_i(m) = \lfloor\mathcal{M}'\rfloor_i(m)$. In other words, no delayed evaluations can change the resolved value of a memory location.
- If $\zeta(\tau)$ is $C(\tau) \le l_{\mathtt{A}}$, then $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$, and $\lfloor t\rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor t\rfloor_2$.

Theorem A.1 (Subject reduction)**.** Suppose $\Gamma\,;\mathcal{Q}\,;pc \vdash e : \tau$, and $\Gamma \vdash \langle \mathcal{M}, \mathcal{D}, t\rangle$, and $\langle e, \mathcal{M}, \mathcal{D}, t\rangle_i \longrightarrow^* \langle e', \mathcal{M}', \mathcal{D}', t'\rangle_i$ where e and e' are not expanded `remote` expressions, and $i \in \{1, 2\}$ implies $\neg\zeta(pc)$. Then $\Gamma\,;\mathcal{Q}\,;pc \vdash e' : \tau$ and $\Gamma \vdash \langle \mathcal{M}', \mathcal{D}', t'\rangle$.

Proof. By induction on the first step and length of evaluation $\langle e, \mathcal{M}, \mathcal{D}, t\rangle_i \longrightarrow^* \langle e', \mathcal{M}', \mathcal{D}', t'\rangle_i$.

- Case (E9). e is `remote` $e'' : \tau[\mathcal{Q}]$. Since e' is not an expanded `remote` expression, the last step of the evaluation must use rule (E11). In this case, we have $i \in \{1, 2\}$, and thus $\neg\zeta(pc)$. So if m is updated during the evaluation, then $\neg\zeta(\Gamma(m))$. Therefore, the first two conditions for $\Gamma \vdash \langle \mathcal{M}', \mathcal{D}', t'\rangle$ immediately hold. Suppose $\zeta(\tau)$ is $C(\tau) \le l_{\mathtt{A}}$. Then $\neg\zeta(pc)$ is $C(pc) \not\le l_{\mathtt{A}}$. Therefore, $C(h) \not\le l_{\mathtt{A}}$ holds for any host h in $\mathcal{Q}$. Thus, $\lfloor\mathcal{D}'\rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor\mathcal{D}'\rfloor_2$. Moreover, $\lfloor t'\rfloor_i = inc(\lfloor t\rfloor_i, C(\tau))$ increments at label $C(\tau)$, which satisfies $C(\tau) \not\le l_{\mathtt{A}}$. Therefore, $\lfloor t\rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor t\rfloor_2$ implies $\lfloor t'\rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor t'\rfloor_2$. By Lemma A.3 and induction on the evaluation length, $\Gamma\,;\mathcal{Q}\,;pc \vdash e' : \tau$.
- Case (E12). By $\Gamma \vdash \langle \mathcal{M}, \mathcal{D}, t\rangle$.
- Case (E13). By Lemma A.3.
- Case (E14). By induction.
- Case (E16). e is `remote` $e'' : \tau[\mathcal{Q}]$. As in case (E9), the last step of the evaluation uses rule (E17). There exists a quorum Q_i in each of the two encoded Qimp evaluations such that all the hosts in Q_i complete evaluating $\lfloor e''\rfloor_i$. So we can consider only the evaluations in Q_1 and Q_2. The goal is to prove that e' is a non-bracket value v if $\zeta(\tau)$ holds. To prove that, we construct a Qimp* evaluation out of the local evaluations at Q_1 and Q_2. The key is to construct a Qimp* memory that captures the local memories of Q_1 and Q_2. First, we construct a Qimp memory M_i out of the local memories at Q_i. For any m such that $\Gamma(m) = \tau^{\mathcal{Q}}$, we have

  $$M_i(m) = resolve(v@h \mid h \in Q_i, \tau)$$

  Then M is constructed as follows,

  $$M(m) = \begin{cases} M_1(m) & \text{if } M_1(m) = M_2(m) \\ (M_1(m), M_2(m)) & \text{if } M_1(m) \neq M_2(m) \end{cases}$$

  It is clear that M is well-typed, because $\mathcal{M}$ is well-typed, which implies that for any m such that $\zeta(\Gamma(m))$, the resolved values of m in $\mathcal{M}_1$ and $\mathcal{M}_2$ are the same.
  By Lemma A.2, we have $\langle e'', M, t\rangle \longrightarrow^* \langle v, M', t\rangle$. By Lemma A.3, $\Gamma\,;\mathcal{Q}\,;pc \vdash v : \tau$.
  For any m that is updated during the evaluation, all the hosts in Q_1 and Q_2 update m with value $\lfloor M'\rfloor_i(m)$. If $\zeta(\Gamma(m))$, then $M'(m) = v'$ is not a bracket value, that is $\lfloor v'\rfloor_1 = \lfloor v'\rfloor_2$. By $\mathcal{Q} \vdash \Gamma(m)$, we have $\mathcal{M}'_1(m) = \mathcal{M}'_2(m) = v'$.
  Let $\mathcal{D}_i = \lfloor\mathcal{D}\rfloor_i$ and $\mathcal{D}'_i = \lfloor\mathcal{D}'\rfloor_i$. Suppose $\zeta(\ell)$ is $C(\ell) \le l_{\mathtt{A}}$. For any $h \in |\mathcal{Q}|$, if $C(h) \le l_{\mathtt{A}}$, then $C(pc) \le C(\tau) \le$

$l_{\mathtt{A}}$, and $C(\Gamma(x)) \leq l_{\mathtt{A}}$ for any x appearing in e, which imply $\lfloor e \rfloor_1 = \lfloor e \rfloor_2$ and $\lfloor t' \rfloor_1 = \lfloor t' \rfloor_2$. By rule (E16), $\mathit{dom}(\mathcal{D}_i') = \mathit{dom}(\mathcal{D}_i) \cup \{\langle \lfloor e \rfloor_i, h_i, \lfloor t' \rfloor_i \rangle \mid h_i \in |\mathcal{Q}|\}$, and thus $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$ implies $\mathcal{D}_1' \approx_{l_{\mathtt{A}}} \mathcal{D}_2'$.
By rule (E17), $\mathcal{D}_i' = \mathcal{D}_i[\langle e, h_k, \lfloor t' \rfloor_i \rangle \mapsto e_k^i \mid h_k \notin Q_i]$. For any new delayed evaluation at e_k^i, if it steps forward by $\langle e_k^i, M_k, \lfloor t' \rfloor_i \rangle_i \longrightarrow \langle e_k', M_k', \lfloor t' \rfloor_i \rangle_i$, then $M_k'(m)$ is either the same as or has a smaller timestamp than $M'(m)$. Therefore, steps of new delayed evaluations will not affect the resolved value of any reference.
Since $t' = \mathit{inc}(t, C(\tau))$, it is clear that $\lfloor t \rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor t \rfloor_2$ implies $\lfloor t' \rfloor_1 \approx_{l_{\mathtt{A}}} \lfloor t' \rfloor_2$. So we have $\Gamma \vdash \langle \mathcal{M}', \mathcal{D}', t' \rangle$.

□

A.4 Confidentiality noninterference

Theorem A.2 (Confidentiality noninterference)**.** Suppose $\Gamma ; \mathit{pc} \vdash e : \mathtt{int}_\ell$, and $C(\ell) \leq l_{\mathtt{A}}$, and $\mathcal{M}_1 \approx_{C \leq l_{\mathtt{A}}} \mathcal{M}_2$, and for $i \in \{1, 2\}$, $\langle e, \mathcal{M}_i, \emptyset, t_0 \rangle \longrightarrow^* \langle v_i, \mathcal{M}_i', \mathcal{D}_i, t_i \rangle$ without (A1) or (A2) steps. Then $v_1 = v_2$, $t_1 \approx_{l_{\mathtt{A}}} t_2$ and $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$.

Proof. Let $\zeta(\ell)$ be $C(\ell) \leq l_{\mathtt{A}}$, and $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2)$. By Lemma A.2, we have $\langle e, \mathcal{M}, \emptyset, t_0 \rangle \longrightarrow^* \langle v, \mathcal{M}', \mathcal{D}', t \rangle$ such that $v_i = \lfloor v \rfloor_i$ and $t_i = \lfloor t \rfloor_i$ for $i \in \{1, 2\}$. Since $\mathcal{M}_1 \approx_{C \leq l_{\mathtt{A}}} \mathcal{M}_2$, we have $\Gamma \vdash \langle \mathcal{M}, \emptyset, t_0 \rangle$. By Theorem A.1, $\Gamma \vdash v : \mathtt{int}_\ell$ and $\Gamma \vdash \langle \mathcal{M}', \mathcal{D}', t \rangle$. By $\zeta(\ell)$, we have $v_1 = \lfloor v \rfloor_1 = \lfloor v \rfloor_2 = v_2$. $\Gamma \vdash \langle \mathcal{M}', \mathcal{D}', t \rangle$ implies $t_1 \approx_{l_{\mathtt{A}}} t_2$ and $\mathcal{D}_1 \approx_{l_{\mathtt{A}}} \mathcal{D}_2$. □

A.5 Integrity noninterference

Lemma A.4 (Subjection reduction with A1)**.** Let $\zeta(\ell)$ be $I(\ell) \not\leq l_{\mathtt{A}}$. Then the subject reduction of Qimp* still holds with evaluation rule (A1), which formalizes integrity attacks.

Proof. With evaluation rule (A1), we just need to reconsider the case of a remote evaluation that begins with (E16) and ends with (E17). There still exist quorums Q_1 and Q_2 that complete the evaluation. However, some low-integrity hosts in Q_1 and Q_2 may be compromised and invoke rule (A1) during the evaluation. So instead of constructing a Qimp* memory using local memories of all hosts in Q_1 and Q_2, we just consider the high-integrity hosts in Q_1 and Q_2. Suppose H_i are the set of high-integrity hosts in Q_i, then we construct the Qimp* memory M just using local memories in H_1 and H_2. The key point is that H_1 and H_2 are enough to resolve any reference replicated on $\mathcal{Q}$. Based on rule (Q1), the intersection between H_1 and any quorum Q contains enough high-integrity hosts. Therefore, M is still well-typed. Similarly, H_1 and H_2 can ensure that any reference updated during the evaluation can be resolved to the correct value. So the rest of the subject reduction proof just holds. □

Theorem A.3 (Integrity noninterference)**.** Suppose $\Gamma ; \mathit{pc} \vdash e : \mathtt{int}_\ell$, and $I(\ell) \not\leq l_{\mathtt{A}}$, and $\mathcal{M}_1 \approx_{I \not\leq l_{\mathtt{A}}} \mathcal{M}_2$, and $\langle e, \mathcal{M}_i, \emptyset, t_0 \rangle \longrightarrow^* \langle v_i, \mathcal{M}_i', \mathcal{D}_i, t_i \rangle$ for $i \in \{1, 2\}$. Then $v_1 = v_2$.

Proof. Let $\zeta(\ell)$ be $I(\ell) \not\leq l_{\mathtt{A}}$. By Lemma A.4 and the same argument as in the proof of Theorem A.2. □

A.6 Availability noninterference

Theorem A.4 (Availability noninterference)**.** Suppose $\Gamma ; \mathit{pc} \vdash e : \mathtt{int}_\ell$, and $A(\ell) \not\leq l_{\mathtt{A}}$, and $\mathcal{M}_1 \approx_{I \not\leq l_{\mathtt{A}}} \mathcal{M}_2$, and $\langle e, \mathcal{M}_1, \emptyset, t_0 \rangle \longrightarrow^* \langle v_1, \mathcal{M}_1', \mathcal{D}_1, t_1 \rangle$ without (A1) and (A2) steps. Then the evaluation of $\langle e, \mathcal{M}_2, \emptyset, t_0 \rangle$ always terminates.

Proof. In Qimp, there are two ways that an evaluation may not terminate. First, there is an infinite loop. Second, a remote evaluation does not terminate because not enough hosts in the quorum system are available.

Like in the Aimp language [19], the typing rules of Qimp ensure that low-integrity inputs cannot affect availability. Therefore, by $\mathcal{M}_1 \approx_{I \not\leq l_{\mathtt{A}}} \mathcal{M}_2$ and that e terminates when being evaluated with $\mathcal{M}_1$, we have that e cannot get into an infinite loop while being evaluated with $\mathcal{M}_2$.

Assume $\langle e, \mathcal{M}_2, \emptyset, t_0 \rangle$ does not terminate. Then it must be the case that some remote evaluation does not terminate. Suppose $\langle e, \mathcal{M}_2, \emptyset, t_0 \rangle \longrightarrow^* \langle E(\mathtt{remote}\ e' : \tau[\mathcal{Q}]), \mathcal{M}_2', \mathcal{D}', t' \rangle$, and $\langle \mathtt{remote}\ e' : \tau[\mathcal{Q}]), \mathcal{M}_2', \mathcal{D}', t' \rangle$ does not terminate. By subject reduction, $\mathtt{remote} e' : \tau[\mathcal{Q}]$ is well-typed. By rules (LOC), (DEREF) and (EVAL), $\mathcal{Q} \vdash \tau$ must hold. By induction, we can prove that the availability label of any sub-expression of e must be at least as high as the availability label of e. Therefore, we have $A(\tau) \not\leq l_{\mathtt{A}}$. So there exists a quorum Q in $\mathcal{Q}$ such that $A(h) \not\leq l_{\mathtt{A}}$ for any host h in Q. So the remote evaluations of e' on Q will terminate. By rule (E11), $\langle \mathtt{remote}\ e' : \tau[\mathcal{Q}], \mathcal{M}_2', \mathcal{D}', t' \rangle$ terminates, which results in a contradiction. So the original assumption does not hold, and $\langle e, \mathcal{M}_2, \emptyset, t_0 \rangle$ always terminates. □